Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

How to Deploy Security Mechanisms Online (Consistently)

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Sebastian Elmar Roth

Saarbrücken, 2023

Tag des Kolloquiums:	28 April 2023
Dekan:	Prof. Dr. Jürgen Steimle

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Karl Bringmann
Berichterstattende:	Dr.-Ing. Ben Stock
	Prof. Dr. Christian Rossow
	Prof. Dr. Matteo Maffei
Akademischer Mitarbeiter:	Dr. Adrian Dabrowski

Zusammenfassung

Um eine Vielzahl von Angriffen im Web zu entschärfen, unterstützen moderne Browser clientseitige Sicherheitsmechanismen, die über sogenannte HTTP Response-Header übermittelt werden. Um jene Sicherheitsfeatures anzuwenden, setzt der Betreiber einer Web site einen solchen Header, welchen der Server dann an den Client ausliefert.

Wir haben gezeigt, dass das konfigurieren eines dieser Mechanismen, der Content Security Policy (CSP), einen enormen technischen Aufwand erfordert, um auf nicht triviale Weise umgangen werden zu können. Daher ist jenes feature auf vielen Webseiten, auch Top Webseiten, falsch konfiguriert.

Aufgrund der Fähigkeit von CSP, auch Framing-basierte Angriffe abzuwehren, überschneidet sich seine Funktionalität darüber hinaus mit der des X-Frame-Options Headers. Wir haben gezeigt, dass dies zu inkonsistentem Verhalten von Browsern, aber auch zu inkonsistentem Einsatz in realen Webanwendungen führt. Nicht nur überladene Verteidigungsmechanismen sind anfällig für Sicherheitsinkonsistenzen. Wir haben untersucht, dass aufgrund der Struktur des Webs selbst, falsch konfigurierte Ursprungsserver, oder CDN-Caches, die von der geographischen Lage abhängen, unerwünschte Sicherheitsinkonsistenzen verursachen können.

Um die hohe Anzahl an Fehlkonfigurationen von CSP-Headern nicht außer Acht zu lassen, haben wir uns auch den Erstellungsprozess eines CSP-Headers genauer angesehen. Mit Hilfe eines halbstrukturierten Interviews, welches auch eine Programmieraufgabe beinhaltete, konnten wir die Motivationen, Strategien und Hindernisse beim Einsatz von CSP beleuchten. Aufgrund der weiten Verbreitung von CSP werden drastische Änderungen allgemein jedoch als unpraktisch angesehen. Daher haben wir ebenfalls untersucht, ob eine der neuesten und daher wenig genutzten, Web-Sicherheitsmechanismen, namentlich Trusted Types, ebenfalls verbesserungswürdig ist.

Abstract

To mitigate a myriad of Web attacks, modern browsers support client-side security policies shipped through HTTP response headers. To enforce these policies, the operator can set response headers that the server then communicates to the client.

We have shown that one of those, namely the Content Security Policy (CSP), requires massive engineering effort to be deployed in a non-trivially bypassable way. Thus, many policies deployed on Web sites are misconfigured.

Due to the capability of CSP to also defend against framing-based attacks, it has a functionality-wise overlap with the X-Frame-Options header. We have shown that this overlap leads to inconsistent behavior of browsers, but also inconsistent deployment on real-world Web applications. Not only overloaded defense mechanisms are prone to security inconsistencies. We investigated that due to the structure of the Web itself, misconfigured origin servers or geolocation-based CDN caches can cause unwanted security inconsistencies.

To not disregard the high number of misconfigurations of CSP, we also took a closer look at the deployment process of the mechanism. By conducting a semi-structured interview, including a coding task, we were able to shed light on motivations, strategies, and roadblocks of CSP deployment. However, due to the wide usage of CSP, drastic changes are generally considered impractical. Therefore, we also evaluated if one of the newest Web security features, namely Trusted Types, can be improved.

Contributions of this Dissertation

This dissertation is based on the papers mentioned below. I contributed to four papers as the main author (*P1, P3-5*) and one as the second author (*P2*). Notably, one of the papers was an extension of the work conducted for my master thesis (*P1*).

[P1] Roth, S., Barron, T., Calzavara, S., Nikiforakis, N., and Stock, B. Complex Security Policy? – A Longitudinal Analysis of Deployed Content Security Policies. In: *Network and Distributed System Security Symposium (NDSS)*. 2020.

[P2] Calzavara, S., Roth, S., Rabitti, A., Backes, M., and Stock, B. A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web. In: *USENIX Security Symposium (USENIX Security)*. 2020.

[P3] Roth, S., Calzavara, S., Wilhelm, M., Rabitti, A., and Stock, B. The Security Lottery: Measuring Client-Side Web Security Inconsistencies. In: *USENIX Security Symposium (USENIX Security)*. 2022.

[P4] Roth, S., Gröber, L., Backes, M., Krombholz, K., and Stock, B. 12 Angry Developers – A Qualitative Study on Developers' Struggles with CSP. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021.

[P5] Roth, S., Gröber, L., Baus, P., Krombholz, K., and Stock, B. Trust Me If You Can – How Usable Is Trusted Types In Practice? In: *Currently under Submission.* 2023.

Further Contributions of the Author

I was also able to contribute to the following papers as a main or co-author. One work was done in the process of my work as a student helper at CISPA (*S1*), another one was part of my master thesis (*S2*), and the others one (*S3*) was created by a student that I advised during his Bachelor thesis in our research group.

[S1] Musch, M., Steffens, M., Roth, S., Stock, B., and Johns, M. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2019.

[S2] Roth, S., Backes, M., and Stock, B. Assessing the Impact of Script Gadgets on CSP at Scale. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2020.

[S3] Stolz, P., Roth, S., and Stock, B. To hash or not to hash: A security assessment of CSP's unsafe-hashes expression. In: *SecWeb Workshop co-located with IEEE Security and Privacy 2022 (SecWeb)*. 2022.

Acknowledgments

First of all, thanks to all the awesome people at the CISPA Helmholtz Centre for Information Security who helped me grow as a person and as a researcher. From the moment I started as a student assistant back in 2015 until today, you all always supported me, and you have all contributed to making my work as a researcher so much fun and so rewarding!

My biggest thank you, goes to my supervisor Ben. You more than live up to the German description "Doktorvater"! Thank you for always having an open ear, for giving me the freedom to grow, to make mistakes and to learn from them, but also for guiding me when necessary and keeping me away from bad decisions. Thanks for all the fruitful discussions, your valuable feedback, and your support during all those years. It was (and will hopefully also be in the future) an indescribable privilege to work with you.

I want to thank all my co-authors Ben, Katharina, Lea, Stefano, Moritz, Alvise, Michael, Nick, and Timothy for their valuable comments, fruitful discussions, and their contributions to the research that we conducted together.

As mentioned above, all people at CISPA always helped and supported me, but I want to specifically thank:

- Katharina and her entire research group (Lea, Matthias, Carolyn, Alexander, Divyanshu, Adrian, Simon, and Dañiel) for always supporting me, adopting me as an *honorable group member*, and renewing my interest in usable security research. Without all of you, my time at CISPA would have been less successful and for sure less fun. Thank you!
- my office mates Shubham, Florian, Jannis, and especially my former colleagues Marius and Aurore, for creating an environment in our office that enabled fruitful discussions, and room for shenanigans.
- my former supervisors Sascha and Michael for giving me the opportunity to start working in research early on during my bachelor's and for igniting my passion for information security.
- our HiWis, especially Moritz, for their helpful contributions to several research and teaching projects.
- to Sebastian, Tobias, Annabelle, Janine, Lea, Oliver, and the whole Corporate Communication Department for always presenting our research in a way that brought even more attention.
- to Michael, Rafael, and the whole Empirical Research Support team for explanations, discussions, and their support.
- to Bettina, Mazi, the people from the Front-Office, and the IT department for helping whenever there was a problem at CISPA that needed to be solved.

In addition to that, I want to thank my friends, especially Niklas, Anna, Hendrik, Lea, Christoph, Jannick, Jonathan, and Felix; my family, especially my sister Alina; and my girlfriend Kerstin, for always supporting me and helping me retain my sanity during stressful times.

Also thanks to all the people from the saarsec CTF team, especially Ben, Simeon, Alex, Daniel, Johannes, and Markus; and all the people from the Jugger team *Keulen Eulen*, for giving me compensation for the rather theoretical and stressful work.

Finally, I would like to thank all the reviewers of the papers in this dissertation for their insights and constructive feedback throughout the peer review process and all of the participants of my studies for their valuable time and insights that ultimately made those projects possible.

Contents

1 Introduction **1**
1.1 State of the Art & Motivation . . . 3
1.2 Contributions of this Thesis . . . 4
1.2.1 Commitment to Open Science . . . 6
1.3 Outline . . . 6

2 Background & Related Work **7**
2.1 Basic Technologies . . . 9
2.1.1 Hypertext Markup Language . . . 9
2.1.2 JavaScript . . . 9
2.1.3 Same Origin Policy . . . 10
2.1.4 Hypertext Transfer Protocol . . . 11
2.2 Web-based Attacks . . . 12
2.2.1 Clickjacking . . . 12
2.2.2 Cross-Site Scripting . . . 13
2.2.3 Cross-Site Request Forgery . . . 15
2.2.4 Network Attacks . . . 15
2.3 Client-side Security Mechanisms . . . 16
2.3.1 X-Frame-Options . . . 16
2.3.2 Content Security Policy . . . 16
2.3.3 Trusted Types . . . 18
2.3.4 Cookie Security attributes . . . 19
2.3.5 HTTP Strict Transport Security . . . 19
2.4 Qualitative Methods . . . 20

3 Historic Deployment of Security Headers In-The-Wild **21**
3.1 Motivation . . . 23
3.2 Methodology . . . 24
3.2.1 Dataset Construction . . . 25
3.2.2 Threats to Validity . . . 26
3.3 Historical Evolution of CSP . . . 27
3.3.1 Adoption and Maintenance of CSP . . . 27
3.3.2 Use Cases for CSP . . . 28
3.4 CSP for Script Content Restriction . . . 29
3.4.1 Insecure Practices Die Hard . . . 29
3.4.2 Allowed Sources . . . 31

3.4.3 Longitudinal Case Studies 32
3.5 CSP for TLS Enforcement 35
3.5.1 Evolution of TLS Enforcement 36
3.5.2 Leveraging CSP for TLS Migration 37
3.6 CSP for Framing Control 38
3.6.1 Evolution of CSP for Framing Control 38
3.7 In-Depth Analysis of Deployed CSPs 40
3.7.1 Reasons for Giving Up on CSP 40
3.7.2 Investigating Insecure Policies 43
3.7.3 Analyzing Secure Sites 43
3.8 Framing Control Notification 44
3.8.1 Insights from Initial Responses 44
3.8.2 Follow-Up Survey 45
3.8.3 Limitations and Additional Survey 46
3.9 Discussion 46
3.9.1 Summary of CSP's Use Cases 47
3.9.2 Complex Security Policy? 48
3.9.3 Quo Vadis, CSP? 49
3.10 Summary 51

4 Inconsistent Deployment of Security Headers In-The-Wild 53
4.1 Motivation 55
4.2 Data Collection Framework 57
4.2.1 Scope of the Study 57
4.2.2 Challenges and Design Choices 58
4.3 Formalizing Inconsistencies 59
4.3.1 Consistency 60
4.3.2 Compatibility Relations 60
4.3.3 Equivalence Relations 63
4.4 Measuring Inconsistencies 66
4.4.1 Overview of the Findings 66
4.4.2 Intra-Test Inconsistencies 68
4.4.3 Inter-Test Inconsistencies 71
4.4.4 Disclosure 73
4.5 Special Case: XFO vs. CSP frame-ancestors 74
4.5.1 Motivation 75
4.5.2 Analyzing Framing Control Policies 75
4.5.3 FrameCheck Results 78
4.5.4 Recommendations & Countermeasures 84
4.5.5 Summary of Framing Control Inconsistencies 88
4.6 Discussion 88
4.6.1 Limitations 88
4.6.2 Overall Security Impact 88
4.7 Summary 89

5 The Developers' Struggle with CSP Deployment 91
5.1 Motivation 93
5.2 Methodology 94
5.2.1 Recruitment and Participants 94
5.2.2 Screening Survey 96
5.2.3 Interview 98
5.2.4 Coding Task 99
5.2.5 Pre-Study 100
5.2.6 Data Analysis 100
5.2.7 Ethical Considerations 101
5.3 Results 101
5.3.1 Participant Demographics 102
5.3.2 Motivation for CSP 102
5.3.3 Roadblocks of CSP 105
5.3.4 Deployment Strategies 108
5.4 Discussion 112
5.4.1 Relations of Roadblocks and Strategies 112
5.4.2 Drawing Task 114
5.4.3 Reflections on Methodology 115
5.4.4 Limitations 117
5.5 Summary 117

6 Is Trusted Types repeating the mistakes made for CSP? 119
6.1 Motivation 121
6.2 Methodology 122
6.2.1 Recruitment and Participants 122
6.2.2 Screening Survey 124
6.2.3 Interview 124
6.2.4 Coding Task 125
6.2.5 Pre-Study 127
6.2.6 Data Analysis 128
6.2.7 Ethical Considerations 128
6.3 Results 129
6.3.1 Participant Demographics & Background 129
6.3.2 Roadblocks for Trusted Types and Where to Find Them 130
6.3.3 Client- vs. Server-side XSS 133
6.3.4 Perceptions on XSS and Trusted Types 134
6.4 Discussion 135
6.4.1 Sourceless iframe Problem 135
6.4.2 Standardized & Customizable Sanitazation 137
6.4.3 Sanitizing JavaScript 137
6.4.4 Impact of Information Sources 138
6.4.5 Limitations 140
6.5 Summary 141

7 Conclusion **143**
7.1 Summary of Contributions 145
7.2 Future Work 146
7.3 Concluding Thoughts 147

Appendix **149**
A Appendix: Complex Security Policy 149
B Appendix: The Security Lottery 152
C Appendix: 12 Angry Developers 153
D Appendix: Trust me If you Can 163

List of Figures

2.1 Clickjacking example Web site that shows an image of our family cat. . 13
2.2 Example for a MITM attack on a client that loads example.com 16

3.1 Usage of CSP only, CSP-RO only, or both 27
3.2 Maintenance of CSPs over time 28
3.3 Classified policies over time 29
3.4 Overall adoption of content restriction and insecure practices 30
3.5 Boxplot showing the number of elements in script allow-list 31
3.6 Average historic Alexa rank of allowed domains 32
3.7 TLS Enforcement Strategies 36
3.8 Evolution of `X-Frame-Options` and `frame-ancestors` 38

4.1 Histogram and CDF of the similarity values for syntactically different header values. 62

5.1 Poster that was used by the OWASP to advertise our study. 95
5.2 Advertisement that was used in our LinkedIn campaign. 96
5.3 The landing page of our screening questionnaire. 97
5.4 Template for XSS/CSP Drawing Task 98
5.5 Categories of Motivations 103
5.6 Categories of Roadblocks 105
5.7 Categories of Strategies 109
5.8 Developer decision tree to create a sane CSP. 114

6.1 Recruitment Flyer posted on Twitter 123
6.2 Diagram of the deployment workflow based on our participants. 130
6.3 Starting point of the interactive roadmap for successful Trusted Types deployment. 140

List of Tables

3.1 Number of sites per year restricting script content, and using hashes, nonces, and `strict-dynamic`. 30
3.2 From report-only to enforced policies 34
3.3 Number of sites per year that set a frame-ancestors directive, as well as number and fraction of sites that set policies not expressible by X-Frame-Options 39

4.1 Selected client conditions that might influence the received security headers 57
4.2 Example observations upon crawling 61
4.3 False positive & false negative rates for similarity 63
4.4 Detected intra-test and inter-test inconsistencies by factor (321 sites in total). We present the numbers with and without page similarity for HSTS to highlight the impact of this choice on the measurement. 67
4.5 Browsers considered in the present study 76
4.6 Framing control semantics of popular browsers 77
4.7 Defenses used in the collected policies 79
4.8 Practices in unduly inconsistent policies (classes might overlap) 80
4.9 Presence and distribution of vulnerable policies 82
4.10 Simplification of multiple XFO directives into a single one (adoption at origin o) 85

6.1 Top 10 third parties by inclusions 126
6.2 Usage of JS frameworks 127

7.1 Union of all intra-test inconsistencies snapshots. 153
7.2 Overview of overlap with additional snapshots of our analysis 154
7.3 Demographics of the 11 participants that completed the survey. (One participant only took part in the interview.) 159
7.4 Drawing task results with concepts and number of participants. (One participant did not draw an XSS attack.) 160

1

Introduction

1.1 State of the Art & Motivation

The Web has improved our ways of communicating, collaborating, teaching, and entertaining us and our fellow human beings. We use it on a daily basis for social media, health care, bank transactions, and improving the knowledge. This importance for our everyday life makes the Web one of the primary targets of malicious actors.

As Web sites are delivered via a Web server, the code running on the server side is a natural target for attacks. SQL Injection (SQLi) attacks smuggle code that can delete, modify, or extract information from the database system of the server. In the case of a Remote Code Execution (RCE), attackers have the capability to execute their own program code on the targeted server, which can lead to a complete takeover of the server. However, not only the server-side code of a Web application can be attacked by different kinds of malicious actors, but also the client side of Web applications can face a plethora of different attacks.

The Website of the airline British Airways faced a *Cross-Site Scripting (XSS)* vulnerability in one of their third-party JavaScript libraries, in 2018: The attackers were able to redirect the customers with their data to an attacker-controlled server with a domain name similar to the one of the airline. Thereby, the attackers managed to get the credit card data of more than 380,000 transactions [15]. Also the multiplayer game Fortnite lost the data of millions of users due to an XSS in an orphaned and insecure page in their application [60]. In 2014 the Enterprise Manager in the McAfee Vulnerability Manager faced multiple *Cross-Site Request Forgery* vulnerabilities, that allowed attackers to send authenticated requests to the application causing state-changing actions such as the modification of content.[82] The social network Facebook faced *Clickjacking* attacks in 2011. Here, malicious actors used invisible iframes, which contained the actual Web site of the network as an overlay over malicious content. This way the victims clicked on like buttons or sent friend requests without even noticing [119]. In addition to that, the network connection between a Web server and the client can also be attacked. Here, a *Manipulator-in-the-Middle (MITM)* attacker can drop, inject, or modify traffic and thus content ad libitum [92].

To mitigate all those different Web-based attacks, modern Web browsers support client-side security mechanisms shipped through HTTP response headers. The browser then enforces these defenses, according to the rules set by the Web sites operator via the server-side response header. New subtypes of XSS are constantly showing up [109, 106, 67, 58, 111, 50, 51, 59, 49, 75, 123]. Also, XSS has a sheer permanent presence in the OWASP Top 10 Web Application Security Risks [90]. In theory, a properly configured *Content Security Policy (CSP)* header can be used to mitigate the effect of different kinds of XSS. However, research has shown that the vast majority of policies in the wild are trivially bypassable [137, 136, 19]. CSP has also replaced the non-standardized *X-Frame-Options (XFO)* header to defend against Clickjacking attacks. However, the adoption of CSP for this purpose, is too low to defend the majority of users [19]. In order to defend against MITM attacks, the *HTTP Strict Transport Security (HSTS)* header can be used to enforce the usage of secure network connections. However, this header is only deployed by 29% of all Websites [94], although services like Let's Encrypt [56] are offering free and easily obtainable TLS certificates for HTTPS connections.

Placing the burden of properly configuring (also complex) security mechanisms on the developers of Web applications opens the floodgates to misconfiguration and misconceptions. Moreover, the mindset, problems, and strategies that developers face when deploying client-side security mechanisms lack a proper scientific analysis. Even if the developer manages to configure a secure policy for a Web application, users may access the same site in multiple different ways. This unforeseeable user behavior, might lead to inconsistent protection of users. Also, the fact that multiple security mechanisms are protecting against the same threat model, but are configured and supported differently, might cause inconsistent behavior.

In this thesis, we evaluate the security of deployed security mechanisms configurations, the complexity of their deployment process, and the impact of client configurations on their consistency. With the results, we hope to influence the standardization committees to change their modus operandi to a more developer-centric evaluation of planned mechanisms. We also aim to guide developers in *how to consistently deploy security mechanisms for their Web applications* in order to defend the users of Web applications against malicious actors.

1.2 Contributions of this Thesis

As explained above, our goal is to elaborate on *How can developers consistently deploy security mechanisms for their Web applications?*. Therefore, we split our overarching question into four sub-problems that need to be solved. First, we present our longitudinal measurement of deployed Content Security Policies, where we want to tackle the questions *How has the quality of the Content Security Policy configurations and its use-cases changed over the course of time?* (RQ1). Second, we deal with the issue *If all users of a Web site get the same level of security?* (RQ2) by analyzing the prevalence of inconsistencies in the security policies of top sites across different client characteristics and by assessing inconsistencies in framing control policies across different browsers. To address the problem of *What are the root causes for insecure practices in CSP deployment?* (RQ3) we conducted a qualitative study involving 12 real-world Web developers including a drawing and coding task. Finally, we use the results of a coding interview with ten security-aware developers to deal with the question: *Is Trusted Types repeating CSP's mistakes?* (RQ4).

RQ1: How has the quality of the Content Security Policy configurations and its use cases changed over the course of time?

To answer our first research question about CSP usage over the course of time, we leverage the unique vantage point of the Internet Archive to conduct a historical and longitudinal analysis of how CSP deployment has evolved for a set of 10,000 highly ranked domains. In doing so, we document the long-term struggle site operators face, when trying to roll out CSP for content restriction. Moreover, we shed light on the usage of CSP for other use cases, in particular, TLS enforcement and framing control. Here, we find that CSP can be easily deployed to fit these two security scenarios, but both lack widespread adoption. Specifically, while the underspecified and thus

inconsistently implemented X-Frame-Options header is increasingly used on the Web, CSP's well-specified and secure alternative cannot keep up. To understand the reasons behind this, we run a notification campaign and subsequent survey, concluding that operators have often experienced the complexity of CSP (and given up), utterly unaware of the easy-to-deploy components of CSP. Hence, we find the complexity of secure, yet functional content restriction gives CSP a bad reputation, resulting in operators not leveraging its potential to secure a site against the non-original attack vectors.

RQ2: Do all users of a Web site get the same level of security?

Users of the Web may access the same site in variate ways, e.g., using different User-Agents, network access methods, or language settings. All these usage scenarios should enforce the same security policies, otherwise, a security lottery would take place: depending on specific client characteristics, different levels of Web application security would be provided to users (inconsistencies). To investigate if all users get the same level of security, we formalize security guarantees provided through four popular mechanisms. We then apply these formal definitions to measure the prevalence of inconsistencies in the security policies of top sites across different client characteristics. Based on our insights, we investigate the security implications of both deterministic and non-deterministic inconsistencies and show how even prominent services are affected by them. Also, we formally study the problem of inconsistencies in framing control policies across different browsers and we implement an automated policy analyzer based on our theory, which we use to assess the state of click-jacking protection on the Web.

RQ3: What are the root causes for insecure practices in CSP deployment?

Research has shown that the vast majority of all Content Security Policies in the wild are trivially bypassable. To find the root causes behind this omnipresent misconfiguration of CSP, we conducted a qualitative study involving 12 real-world Web developers. By combining a semi-structured interview, a drawing task, and a programming task, we were able to identify the participant's misconceptions regarding the attacker model covered by CSP as well as roadblocks for secure deployment or strategies used to create a CSP.

RQ4: Is Trusted Types repeating CSP's mistakes?

To see whether Trusted Types meets the same fate as CSP, we uncover roadblocks that occur during the deployment of Trusted Types. Moreover, by conducting a semi-structured interview including a coding task with ten security-aware Web developers we were able to investigate strategies on how developers can circumvent those problems. Our work also identifies key weaknesses in the design and documentation of Trusted Types, which we urge the standardization committee to fix before Trusted Types faces the destiny of CSP. Also, we hope that this work provides a scientific foundation for the ongoing discussion regarding the future of Trusted Types [81].

1.2.1 Commitment to Open Science

Because a transparent and open scientific process increases not only the verifiability of our results but also the accessibility of our observations, and because I understand science as a duty for the benefit of society, I am committed to open science. Therefore, I open-source the dataset, the data-gathering scripts, the tools, and the analysis scripts whenever possible in all projects.

Thus, the overview of normalized policies [A5], which we gathered for [P1], is publicly available. We open-sourced our analysis library [A2] as well as the server-side proxy [A3] for [P2]. The crawling and analytics scripts for [P3] are publicly available [A4]. Also, the Web applications used for [P4] are also available [A1] on GitHub.

1.3 Outline

The rest of this thesis is structured as follows: First, Chapter 2 explains the Web and its technologies including the aforementioned client-side attacks and security mechanisms. Furthermore, the chapter presents details about related work in the field of security headers on the Web. Chapter 3 evaluates *how the quality of the Content Security Policy configurations and its use-cases have changed over the course of time* (RQ1). The small survey regarding framing control conducted throughout the work presented in Chapter 3 then inspired us to have a closer look into security inconsistencies in deployed security mechanisms and what caused them (Chapter 4), such that we know *if all users of a Web site get the same level of security* (RQ2). Chapter 3 also revealed that there is (and has always been) a sheer omnipresent misconfiguration of CSPs in real-world Web applications. To provide a scientific foundation for the educated guesses of the reasoning behind these misconceptions, we then extended the appraisals based on empirical data with qualitative data from a developer-centric interview study (Chapter 5). This way we were able to find *the root causes for insecure practices in CSP deplyoment* (RQ3). However, as CSP is too widely deployed to introduce the changes that Chapter 5 suggests, we then focused on a novel, not yet standardized, and not widely deployed security mechanism: Trusted Types. With the roadblocks, strategies, and improvement suggestions that we investigate in the semi-structured interview study that includes a coding task (Chapter 6) *we can then hopefully have a positive impact on the mechanism before it is too late and Trusted Types faces CSPs legacy* (RQ4). Finally, Chapter 7 summarizes all contributions of this thesis and elaborates on future work, before concluding this thesis.

2
Background & Related Work

Contributions of the Authors

This chapter explains the technical and methodological background of this work and gives an overview of the work that relates to this thesis. The content of this chapter is a mixture and variation of the background and related work sections from the main papers for this thesis [P1, P2, P3, P4, P5].

2.1 Basic Technologies

All of the papers covered by this dissertation tackle the Web and its technologies. Thus, this section explains the basic technologies of the Web, such as HTML, JavaScript, or the Web's transfer protocol HTTP.

2.1.1 Hypertext Markup Language

The Hypertext Markup Language (HTML) is the semantic representation of the appearance of a document that a Web browser renders into a multimedia Web page. Some HTML tags define structure such as `body` or `div` tags, others specify how content is displayed e.g. the `table` tag, and again others can be used to display media content like pictures (`img` tag). In order to produce dynamic and interactive Web applications, a developer needs to also use Cascading Style Sheets (CSS) to change the visual design of HTML elements, as well as JavaScript (JS) to enable interactive user experience via Browser executed programs delivered in HTML `script` tags. With the release of HTML 4 [105], the tags support so-called event handlers as attributes. Those can be used to trigger JavaScript execution on certain events such as clicking an HTML element, pressing a key, moving the mouse, or changing a value. Thus, HTML `script` tags are now no longer the only tag that is capable of executing JavaScript programs. Also, version 4 added support for the inclusion of frames via the `frame`, `iframe`, or `frameset` HTML tag. Those frames enable developers to display multiple documents in a single window. While the current living HTML standard [141] deprecates the `frame` and `frameset` tags, iframes are still present nowadays and used by a plethora of different attacks, such as Clickjacking, which Section 2.2.1 explains in detail.

2.1.2 JavaScript

To extend the Web platform beyond just displaying static content, JavaScript (also called ECMA Script [48]) has been introduced. Traditionally an HTML `script` tag can be used to execute JavaScript which is either loaded as an external resource using the `src` attribute or executed directly as inline JavaScript. However, also other HTML tags can execute the script via specific event attributes. For example, the `img` tag can carry `onerror` that has JavaScript code as a value. This code is then executed if an error occurs with the tag, such as network errors during loading the resources defined in the `src` value. But also plenty of other HTML tags can carry a plethora of different event attributes that are capable of executing JavaScript, with and without requiring user interaction. In order to also enable support for dynamically

Listing 1 Example for different ways to execute JavaScript.

```
<script src="https://example.com/external-script.js"></script>

<script>
    console.log("This comes from an inline script!");
</script>

<img src="https://example.com/non-existsant-url.jpg"
     onerror="console.log('This comes from an attribute!')"/>

<script>
   eval("console.log('This code" + " " + "was evaled!')");
</script>

<a href="javascript:console.log('Executed via URL!')">ClickMe</a>
```

crafting JavaScript during runtime and then execute this code, string-to-code conversating functions, most prominently `eval('<code>')`. But also other JavaScript APIs e.g, `settimeout`, `setinterval`, or the `Function` constructor. Also, URLs that are using the `javascript:` or `data:` scheme can execute JavaScript if they are loaded by the browser. Examples of the previously mentioned ways to execute JavaScript are shown in Listing 1. Notably, those are only examples and not an exhaustive list, because there are multiple other ways to execute JavaScript code in Browsers.

In this work, we focus on JavaScript's original use case as a client-side scripting language executed in Web browsers. However, it is also possible to use JS on the server side with local interpreters like `node.js`. Hence the importance of JavaScript and its security considerations goes beyond the context of client-side Web security, although this work only focuses on code executed by a Web browser.

2.1.3 Same Origin Policy

Nowadays, Web sites can include other Web sites for example in so-called HTML `iframe` tags, or get a handle to other pages programmatically opened in a new tab or browser window via the `window.open()` JavaScript API. In order to compartmentalize those pages into different scopes the concept of a Web Origin [7] has been introduced. A Web Origin is an execution scope that is bound to the protocol, hostname, and port of the loaded document. Alongside with this scoping the RFC 6454 [7] introduces the most basic security mechanism for the Web, the Same Origin Policy (SOP). This policy ensures that only documents with the same Web Origin can access each other, such that operators of malicious Web sites can not interfere with benign Web sites.

Notably, there is an exemption from the SOP. If you load JavaScript, even from a third party, it is executed in the context of the including Web site. Therefore, the operator of the application has to trust the external party, as their scripts have the

Listing 2 Example of a JSONP data exchange across origins

```
<script>

function processData(data) {
  // JS Code that is processing third-party data
}

</script>
<script src="//example.com/data?callback=processData"></script>
```

same capabilities as scripts that originate from the own site. However, even if the third party itself is trusted, it can happen that their Web server gets compromised. Thus, all scripts that are loaded from this infected third party are now possibly malicious scripts, which are executed in the context of the including Web site.

As mentioned above third-party JavaScript that is loaded into a Web application is exempt from the SOP. However, data that is loaded from third parties might not be readable by the application. To still allow this data sharing across origins developers came up with JSON with Padding (JSONP) to encapsulate the data into callback functions that are loaded as JavaScript. The example displayed in Listing 2 shows code that prepares a function to process the third-party data and loads this data from `example.com` via a URL that provides a callback parameter. On the server-side of `example.com` the value of this is then used to create JavaScript that encapsulates the data (e.g. `processData(raw_data)`) as a response.

2.1.4 Hypertext Transfer Protocol

The first version (1.0) of the Hypertext Transfer Protocol (HTTP) was standardized back in May 1996 [10] by the Internet Engineering Task Force (IETF). The protocol, operating application layer, is designed to enable distributed and collaborative information systems. Because the protocol itself was built in a generic way it allows for a plethora of different applications, most prominently the transition of HTML documents (Web sites). One difference between the non-standardized version 0.9 and the first standardized version which is especially important for this work is the introduction of the HTTP header field, which can be used to carry meta-information about the request such as the purpose or the context of the request. The first version of the protocol, however, still had room for improvement. In version 1.1 [42], improvements like persistent connections and caching [43] of responses have been introduced in order to speed up loading time and reduce bandwidth consumption. Also, its later versions 2.0 [9] and 3.0 [11] introduce multiple improvements, however, those are not covered by this work as they do not interfere with the security mechanisms that are the topic of this dissertation. Notably, HTTP itself is sent in plain or only encoded, which means that it does not provide any connection security per default. Therefore, nowadays HTTP traffic is usually encapsulated in a secure TLS tunnel (HTTPS).

2.1.4.1 HTTP Headers

HTTP headers [44] are additional pieces of information that are part of an HTTP request and also of HTTP responses. Request headers, which are sent from the client to the server, can provide the server with the context of this request, such as the origin of the request or if the client is authenticated. The headers present in responses can be used to provide additional information about the content type or the size of the response. But, as mentioned in Section 2.1.4 the support for HTTP headers is especially important for this work as we tackle security mechanisms that are communicated via HTTP headers from the server to the client side in all primary works presented in this dissertation. Details of each of the HTTP security headers and the mechanisms that they enforce are described in Section 2.3. Syntax-vise HTTP headers consist of a case-insensitive name followed by a colon and then followed by the value of the header and are placed between the HTTP status line and the actual content of the package.

2.1.4.2 Cookies

One of the most essential HTTP headers for modern Web applications is the `cookie` header [6], as it is nowadays used to go beyond the stateless construct of HTTP to store information across multiple HTTP requests, such that sessions, and thus logins to Web applications, are possible. For example, the Web server (on successful login) generates a secret session identifier, that is sent via the `set-cookie` response header. The browser will save that cookie in its cookie jar. Until the expiry time (defined by the `Expires` keyword) has been reached this cookie is now added as `cookie` header to each request that matches the configuration of the `Domain` and the `Path` rules [6].

Notably, cookies are not only used for session management, but also to personalize the content of a Web site (e.g. language settings), or they are used for tracking users to enable personalized advertisements. The usage of cookies for storing user-specific information and session information of the user are the reason why cookies are considered a valuable target for attackers. Thus, multiple additional keywords for cookies have been introduced to harden them against malicious actions. Details of those cookie security mechanisms are explained in Section 2.3.4.

2.2 Web-based Attacks

This section gives an overview of the different Web-based attacks that this work covers. In addition to the explanation of the attack or the group of attack, the current state of science around those mechanisms is described.

2.2.1 Clickjacking

As mentioned in Section 2.1.1, developers can use the HTML iframe tag to integrate a frame of another Web site in their Web application. This feature is for example frequently used by advertisements or social media widgets. However, a developer can freely choose the style of such an iframe, in particular, its opacity. Thus, attackers can trick their victims into clicking elements in another Web application, and thus

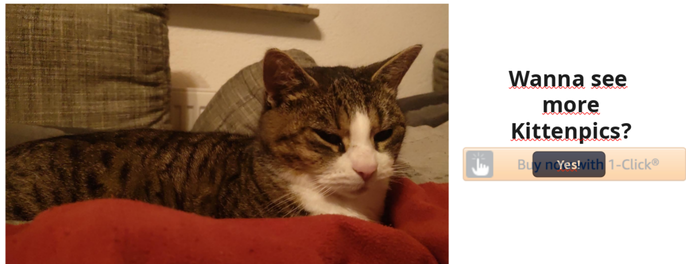

Figure 2.1: Clickjacking example Web site that shows an image of our family cat.

issuing state-changing actions, without the victim even noticing. Malicious actors can place opaque iframes that contain like/retweet buttons or also instant-buy buttons over seemingly benign buttons, such that users, if authenticated to the social network or the payment provider, buy certain products or follow specific content. Figure 2.1 shows how seemingly benign content of a Web site that provides access to cat pictures can be used to trick the user into clicking on an Amazon 1-Click buy button (with opacity set to 66%) that overlays the ”Yes” button.

2.2.2 Cross-Site Scripting

As mentioned in Section 2.1.3 the SOP protects Web Origins from accessing one another, such that e.g., JavaScript can only interact with the DOM of the Website that executes it. If now however an attacker manages to execute JavaScript in the origin of a vulnerable Web site, this script has the same capabilities as a script that is served legitimately by the page itself. Thus, the attacker’s script can steal sessions or login credentials, impersonate user actions, or change the page content to distribute fake news. Since its initial discovery back in 1999 [109] numerous publications and blog posts investigated the different types of Cross-Site Scripting (XSS), such as XSS through content sniffing [106], client-side (sometimes called *DOM-based*) XSS [67], universal XSS [58], self XSS [111], scriptless XSS [50], mutation-based XSS [51], expression-based XSS [59], Same-Origin method execution [49], gadget-based XSS [75], or persistent client-side XSS [123]. In general, however, XSS can either be persistently stored or only reflected per request, while the vulnerable code is either located on the server or the client side. The plethora of different types of XSS as well as their omnipresence in annually published OWASP Top 10 Web Application Security Risks [90] shows that XSS is a problem that is still lacking a proper solution.

2.2.2.1 Server-Side XSS

In the dimension of server-side XSS, the vulnerable code is located on the server side. In this case, parts of the request such as the URL parameters (reflected) or stored user content such as profile descriptions (persistent) are mixed up with the developer-specified markup in the response. This way an attacker can inject malicious markup such as *script* tags that are then delivered via the server of the vulnerable application,

Listing 3 Example of PHP code vulnerable to a reflected server-side XSS

```
// Processing of requests to //example.com/welcome?user=<username>
...
echo "<p>Welcome " + $_GET['user']; +  "!</p>";
...
```

which causes the script to run in the context of this Web application. An example of vulnerable PHP code that has a reflected server-side XSS is displayed in Listing 3.

Notably, server-side XSS can be mitigated or even defended against by enforcing Type safety on the backend such that a string (or XSS payload) can never end up in a response [65]. Also many popular frameworks, such as the Python Django Framework, automatically encode, and thus sanitize, every string that is programmatically passed to the template rendering engine [35]. Thus, a developer needs to explicitly use the `mark_safe` method in order to cause a server-side XSS vulnerability. Also, academic solutions, such as Kirda et al.'s Noxes [66], can be used to prevent XSS. Noxes uses a set of rules to detect XSS attempts. For example, it requires user interaction for all dynamically crafted links to enable the user to take either temporary or permanent security decisions.

2.2.2.2 Client-Side XSS

Another dimension of XSS attacks is the client side where the vulnerable code is not reflected by the server but included and executed fully on the client side within the victim's browser. As mentioned in Section 2.2.2, an injection can happen either happen in a reflected fashion, in this case for example by using data from the URL via `location.href`, or it can originate from a persistent client-side storage such as cookies or the `localStorage` API. Either way, the vulnerability is that the attacker code from these locations at some point ends up a JavaScript sink that can execute code such as *innerHTML* or *eval*, which is exemplified by the code in Listing 4.

For reflected client-side XSS multiple works [76, 78, 127] used a modified browsing engine to taint dangerous flows from a URL into executing sinks, to show that client-

Listing 4 Example of JS code vulnerable to a reflected client-side XSS

```
window.onload = funciton() {
  let url = decodeURI(location.href);
  let html_el = document.getElementById('error_msg');
  html_el.innerHTML = "Loaded URL (" + url + ") was not found!";
}
```

side XSS is a prevalent problem even in the Top most visited Web applications. Steffens et al. [123] also used taint-tracking to show the prevalence of persistent client-side XSS vulnerabilities in the Top 5,000 Web sites. They showed that 21% of the sites that make use of data originating from client-side storages are vulnerable to persistent client-side XSS.

2.2.3 Cross-Site Request Forgery

In a Cross-Site Request Forgery (CSRF) attack, a malicious actor performs actions on behalf of a victim. This action can either be a login CSRF where the victim has logged in at a Web application with the credentials of an attacker-controlled account, also called session-fixation-attack, or an authenticated CSRF where victims are already logged in with their account and is thus authenticated to the vulnerable service. This attack works because browsers may not only add cookies to the requests that are performed by the user, but also to those that happen programmatically in the background. Thus in the case of a login CSRF, the victims might get confused to be logged into their own account and entering sensitive information into the account of the attacker. In the case of an authenticated CSRF, a malicious Web site starts programmatically issuing state-changing requests e.g. a bank transaction to a vulnerable banking service, the browser adds the authentication cookie of victims to this request if they are logged in. Thus, the attacker can impersonate the user and perform arbitrary actions in the vulnerable Web application. If the victim is logged in as an administrative account, this can be even more severe, as a CSRF attack can then compromise the entire Web application. Pellegrino et al. [100] showed that authenticated CSRFs can be automatically detected when we have access to the source code of a Web application, while Sudhodanan et al. [129] provides an extensive overview of login CSRF vulnerabilities.

While there are server-side mechanisms to mitigate the issue such as Anti-CSRF Tokens [8] or checking of the Fetch Metadata request headers [143], this dissertation is focusing on client-side security mechanisms. One of them, the Cookie Security Attribute `SameSite`, can potentially mitigate CSRF attacks, which Section 2.3.4 explains in detail.

2.2.4 Network Attacks

A Network attacker extends the capabilities of a Web attacker with full control of the unencrypted HTTP traffic. Figure 2.2 shows an example of a Machine-in-the-middle (MITM) attack on a client that visits example.com. By proxying the traffic between the two parties, the MITM can, for example, read packages to steal credentials, drop packages to deny service, or modify content in order to create seemingly trustworthy fake news or distribute malware. There are several methods how an attacker can get into the middle of a communication, such as ARP spoofing [107], DNS poisoning [118], fake wireless access points [2], physically tapping into a network line, etc. Those kinds of attacks are particularly dangerous because they often go undetected, as the two parties may not be aware someone tampers with their communication.

Although the Web is fast progressing towards full usage of transport layer encryption through TLS, browsers do not automatically upgrade all connections to HTTPS to avoid

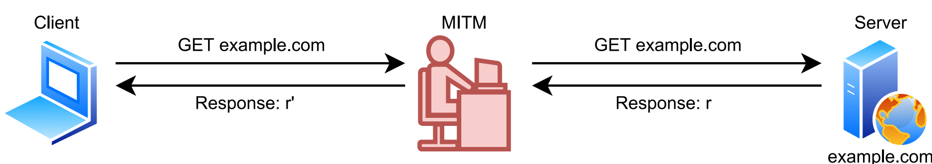

Figure 2.2: Example for a MITM attack on a client that loads example.com

breakage. This introduces the danger of attackers who force a victim's browser to make a request towards the HTTP version of a site, thus allowing the attacker to perform impersonation attempts, e.g., for phishing or to sniff non-Secure cookies. In order to harden a Web application against this kind of attack a Web site's operator can deploy a `Strict-Transport-Security` header, which Section 2.3.5 explains in detail.

2.3 Client-side Security Mechanisms

Given that the main papers presented by this dissertation are either measurements of client-side security mechanisms or qualitative analyses of their real-world deployment roadblocks, this section explains the different mechanisms as well as the current state of science around those.

2.3.1 X-Frame-Options

In order to reduce the attack surface for Clickjacking attacks (Section 2.2.1) developers can deploy the X-Frame-Options (XFO) header [108]. By configuring this header, developers can control who can load their Web application in an iframe. The header itself can have three different modes. With `SAMEORIGIN` the framing is restricted to the origin of the Web application itself. The `ALLOW-FROM` value followed by a specific origin can be used to allow framing from one specific origin. And the `DENY` value prevents any framing of the Web site.

However, support and the implementation of the different modes are inconsistent, as for example, only some browsers support the `ALLOW-FROM` mode. And even if supported this mode only enables developers to allow one single origin. Due to these limits and the inconsistent support, the header should be replaced by the corresponding CSP directive, which Section 2.3.2.1 explains in detail.

2.3.2 Content Security Policy

The Content Security Policy (CSP), initially proposed by Stamm et al. [121], ought to mitigate the damage an XSS attacker can cause and be later extended to cover additional threats. The policies themselves are deployed by the server side, either as HTML meta tags or via HTTP response headers, and enforced on the client side by the browser [139]. It contains one or multiple directives that are followed by a list of source expressions, which can either be keywords or allowed sources for the corresponding directive.

Listing 5 Example Content Security Policy Header

```
content-security-policy:
      script-src 'self' cdn.com *.partner.net;
      frame-ancestors 'self' partner.net;
      Upgrade-insecure-requests;
```

In order to mitigate XSS attacks the site's operator has to set a *script-src* (or alternatively a *default-src*) directive. By default, these directives forbid the execution of inline JavaScript such as inline script tags or inline event handlers and also disables the usage of JS functions that are doing a string-to-code conversion such as *eval*. However, even when those dangerous practices are disallowed, a malicious actor can still include scripts from an attacker-controlled domain. Therefore, a CSP directive such as *script-src* is followed by a list of allowed sources that can be used to include JavaScript.

As shown by Steffens et al. [122] the curation of those lists of trusted domains is a hard task, as a common practice of third-party scripts is to dynamically include other third-party scripts. To address the problem of ever-changing inclusion trees of third parties, Weichselbaum et al. [136] proposed the *'strict-dynamic'* source-expression. If present, the trusted script propagates the trust to scripts that are programmatically added by them. Also, in the presence of *'strict-dynamic'* in the CSP it is only allowed to trust scripts via the usage of *nonces* or their content hashes.

A *nonce* is a secret that is random for each request, such that an attacker can not guess it. If present in a CSP allow list, only script tags that carry the same nonce are allowed to be included and executed. Because *nonces* are only attached to trusted scripts, they also mitigate the attacker's capability to bypass the CSP via side-loading script-gadgets [75] or abusing open-redirects in one of the allowed domains [S2].

Although there are approaches to automate the generation of a CSP [37, 98], research has shown that the vast majority of policies in the wild remain insecure [137, 136, 19], a work by Steffens et al. [122] investigated potential technical problems of CSP.

2.3.2.1 CSP frame-ancestors

Given the problems of the underspecified XFO header, framing control has been incorporated into CSP as `frame-ancestors` directive. This solution has a clear advantage over XFO due to its standardized support and additional expressiveness, which is why browsers supporting CSP Level 2 must ignore the XFO headers in the presence of a CSP that includes a `frame-ancestors` directive. However, XFO is still the only way for a Web application to control framing in legacy browsers.

In contrast to XFO, CSP `frame-ancestors` performs the origin check for framing based on *the whole chain* of nested frames (ancestors) between the top-level browsing context and the framed page, which offers the strongest security guarantees by ruling out double framing. Moreover, CSP is strictly more expressive than XFO, since it can take advantage of the full CSP syntax, which allows one to allow an arbitrary (possibly

Listing 6 Trusted Types Example

```
let TT = trustedTypes.createPolicy('default', {
    createHTML: function(rawHTML) {
        return sanitizeHTML(rawHTML);
    },
    createScript: function(rawJS) {
        return sanitizeJS(rawJS);
    },
    createScriptURL: function(scriptURL) {
        return sanitizeURL(scriptURL);
    }
});
```

empty) list of origins. For example, the DENY directive of XFO can be simulated by setting the `frame-ancestors` directive to `'none'`, while the SAMEORIGIN directive can be simulated by setting it to `'self'`. Even better, CSP can be easily used to allow all subdomains of given domains, e.g., `frame-ancestors *.foo.com *.bar.com`, which cannot be expressed through XFO. To mimic this behavior in XFO a developer would need to implement logic on the server side that checks the HTTP `Referer` header sent by the browser (known as *Referer Sniffing*) and deploy XFO in DENY or ALLOW-FROM mode depending on the value of the `Referer` header.

2.3.3 Trusted Types

Although the problem of client-side XSS and its prevalence in Web applications is well understood, a proper defense or mitigation technique is still missing. As mentioned in Section 2.2.2.1 Google engineers enforce type safety on the backend such that dangerous markup can never end up in the body of a response [65]. To transfer this defense idea to the client side, Google suggested Trusted Types [69] to filter values before they flow into dangerous JS sinks such as *eval* or *innerHTML*. To deploy Trusted Types a developer needs to first set the corresponding CSP `trusted-types` directive which defines the sanitizer policy names that are allowed, then enforce the usage of those policies via the `require-trusted-types-for` directive with the `'script'` value, and finally create the actual Trusted Types policy including the sanitizer functions using the `trustedTypes.createPolicy` JavaScript function, as exemplified in Listing 6.

When deployed correctly, Trusted Types enforces that *all* dangerous JavaScript APIs that might lead to code execution can only be used with a Trusted Types Object. To create those objects the developer either needs to explicitly call the corresponding Trusted Types sanitizer function e.g. `TT.createHTML(input)` or define the sanitizers as *default* policy, such that they are inherently called whenever a dangerous sink is used. In a 2021 document [68] reported on Google's success in deploying Trusted Types on 130 of their services. Furthermore, the report mentions that Google faced no client-side XSS in their Trusted Types protected applications. The report also men-

tioned that new tools and better framework support can ease the deployment. In fact, Wang et al. [134] showed how Trusted Types can be incorporated into Web frameworks such as AngularJS. So, if Trusted Types works as it ought to, it could potentially be the solution to effectively mitigate the problem of client-side XSS.

2.3.4 Cookie Security attributes

As explained in Section 2.1.4.2 cookies are quite important for features that many Web sites are offering nowadays (e.g. login). However, despite their importance for modern Web applications, cookies suffer from a range of security problems in their default configuration. A Web site's operators are thus recommended to improve the level of protection of their cookies by marking them with appropriate *security attributes*.

The *HttpOnly* attribute ensures that cookies are not accessible from JavaScript, which prevents cookie theft through malicious scripts, e.g., injected through XSS. The *Secure* attribute guarantees that cookies are never sent in plain HTTP requests, but only over encrypted HTTPS connections, which rules out network sniffing attempts. The combination of both keywords significantly raises the confidentiality guarantees of cookies, which is particularly important to protect against session hijacking [18, 38].

The latest addition to the set of attributes is called *SameSite*, which is meant to protect against CSRF attacks. If the attribute is set to Lax, cookies are only sent on cross-site requests when the main frame is navigated using a safe method like GET. If instead the attribute is set to Strict, cookies are never sent across sites. Some modern browsers like the latest versions of Chrome automatically promote all cookies to SameSite Lax, hence site operators can opt-out from protection by setting the SameSite attribute to None for compatibility reasons.

Notably, setting all authentication cookies as SameSite would mitigate the issue of CSRF (Section 2.2.3), as those cookies are not sent on cross-site sub-requests. However, setting authentication cookies to SameSite *Lax* or even *Strict* is not always possible for services as their business model might rely on being integrated as a third-party iframe into other services for example social media integration such as Twitter timelines or Facebook like buttons.

2.3.5 HTTP Strict Transport Security

In order to prevent the issues explained in Section 2.2.4, *HTTP Strict Transport Security* (HSTS) was introduced. Once this mechanism is set for a specific HTTPS host, via the `Strict-Transport-Security` header, the browser will only connect to this application via HTTPS for the duration specified in the max-age attribute (or until an HSTS header with max-age set to 0 is received). Optionally, HSTS can specify the includeSubDomains directive, which extends protection to all the subdomains of the host that deploys the security header. This is important to defend against network attackers who could otherwise forge cookies from HTTP subdomains and to prevent the exfiltration of domain cookies lacking the Secure attribute.

Due to its design, HSTS faces a Trust-On-First-Use (TOFU) problem because network attacks can be performed before TLS connections are enforced via the Strict-Transport-Security header in the first response. In order to get rid of this issue, hosts

supporting HSTS can also ask for inclusion in the HSTS preload list [29], which is a public list of known hosts where browsers should activate HSTS by default. To be accepted into the HSTS preload list, a host must serve a valid HSTS header with max-age set to at least one year, have enabled includeSubDomains, and include the preload directive. Since preloading implies a fully functional HTTPS setup for all of a site's subdomains, which might cause problems in practice, the preload list offers a feature for removal. For this removal request to go through, a site has to serve a valid HSTS header (i.e., at least specifying a max-age value) *without* the preload directive [30]. After this, the site is removed from the preload list without further notice to the Web site's operator.

2.4 Qualitative Methods

Two of the main publications in this thesis explore the origins of behaviors and misconceptions of a security mechanism [P4, P5]. Here a "bottom-up" approach of a qualitative study design [61] is what we rely upon. In the area of IT Security, the qualitative analysis of semi-structured interviews was already used to better understand problems, misconceptions, and mindsets of developers [72], users [57, 64], or administrators [71] of IT Systems. While this is suitable for exploring a topic, and investigating concepts and trains of thought of participants on topics which they are already familiar with, some, especially the investigation of previously unknown scenarios need a more controllable study setting, namely lab studies. Here the researcher can adjust the settings of the study to observe if certain changes have possible effects on the investigated scenario. In order to get qualitative data out of those lab tasks, the participants can be asked to think aloud such that the process sheds light on, for example, the motivations behind certain decisions.

In one of the primary works in this dissertation [P4], we used also incorporated a drawing task. Those tasks can be powerful to visualize participants' understanding of a system or concept [12]. If needed, for example, to reduce the drawing effort in an online setting, the participant is given a template to complete. Participants then explain their understanding of the system as they draw. In security and privacy research, drawing tasks have, among other things, already been used to explore user's understanding of the internet [64], and end users' and administrators' understanding of HTTPS [71].

3

Historic Deployment of Security Headers In-The-Wild

Contributions of the Authors

This chapter is based on *"Complex Security Policy? – A Longitudinal Analysis of Deployed Content Security Policies"* [P1]. Sebastian Roth, Ben Stock, Stefano Calzavara, and Nick Nikiforakis conceived the study. All authors contributed to the design of the study. Sebastian Roth conducted a pre-study with a smaller dataset during his master's thesis to get preliminary results. Afterwards, he gathered the large-scale dataset and analyzed the gathered data. Timothy Barron and Nick Nikiforakis investigated how allowed domains in a CSP can be abused, therefore the subsection that presents those results is removed from this thesis. Sebastian Roth and Ben Stock created and sent out the email notifications as well as the CSP survey. Sebastian Roth created the public overview of normalized policies [A5]. Ben Stock, Stefano Calzavara, Nick Nikiforakis supervised the study. All authors commented and contributed text to individual sections.

3.1 Motivation

Given the complexity of modern Web applications and the number of different attacks that Web sites and their users can be exposed to, browser vendors have long supported a wide range of additional opt-in security mechanisms.

The Content Security Policy (CSP) is one such mechanism that was introduced in 2010 and was initially designed to grant Web developers more control over the content loaded by their Web sites [121]. Unlike other security mechanisms that have a limited number of configuration options, CSP allows Web developers to author complex, allow-list-based policies to precisely specify the sources that are trusted for a wide range of content including JavaScript, images, plugins, and fonts.

Though the (in)effectiveness of CSP has been analyzed and debated in several research papers [137, 19, 136, 21], CSP is still under active development and is routinely adopted by more and more Web sites. The, at the time of conducting this study, most recent study [21] observed an increase of one order of magnitude in CSP deployment in the wild between 2014 and 2016. Notably, though, virtually all papers have focused on CSP as a means to restrict content and have treated its newly added features (such as TLS enforcement and framing control) as side notes. To close this research gap and holistically analyze CSP it is important to take a critical look at how CSP deployment has evolved over time, so as to understand for which purposes developers use CSP and how it affects security.

To this end, we leverage the Internet Archive to obtain the historical CSP headers for 10,000 highly ranked domains from 2012 to 2018. Leveraging this unique vantage point for a long-term study, we first investigate CSP's evolution for content restriction and specifically highlight numerous case studies that document the struggle in rolling out a CSP. Among the insights, we find that 56% (251/449) of Web sites that test CSP for content restriction with report-only, presumably due to its complexity, refrain from ever enforcing *any* policy. Similarly, we find numerous examples of sites trying to keep on top of their allow lists for months, before eventually giving up.

Notably, though, our longitudinal lens over seven years allows us to show that CSP's

primary use case is gradually shifting away from the original goal of content restriction, with 58% (720/1,233) of Web sites deploying CSP for other purposes. In particular, we document the increase in the adoption of TLS enforcement and how the security mechanism is used in practice for *utility* purposes. Moreover, we find that while CSP is well-equipped to rid the Web of the `X-Frame-Options` (XFO) header, many sites still rely on this deprecated header incapable of providing CSP's fine-grained framing control. To understand the reason behind this lack in CSP adoption, we conduct a notification campaign of sites running XFO. Based on the over 100 responses we received, we find that CSP's full potential is scarcely documented and the perceived complexity of CSP shies operators away from even the easy-to-deploy parts. Furthermore, using an in-depth analysis of violated policies as well as sites that are either trivially insecure or are able to sustain a secure policy, we propose CSP extensions to ease adoption and improve security in practice.

The rest of this chapter is structured as follows:

- We leverage the Internet Archive to conduct the longest study of the CSP mechanism to date (2012–2018), and discuss how our insights can generalize (Section 3.2).
- We document the evolution of CSP and its use cases over time, showing its gradual move away from content restriction to other security goals (Section 3.3).
- Based on the collected CSPs for content restriction, we document the long-term struggles site operators face in the deployment of CSP (Section 3.4).
- We shed light on the previously dismissed CSP use cases of TLS enforcement (Section 3.5) and framing control (Section 3.6), and highlight their growing importance.
- To untangle the reasons behind CSP's failed adoption, we conduct an in-depth analysis of sites that gave up on CSP, ran insecure CSPs, and those that managed to run a strict CSP. Based on this, we identify characteristics that are blocking sites from CSP deployment (Section 3.7).
- Given that the adoption of CSP for framing controls lags behind the underspecified XFO header, we conduct a notification of sites at risk and report on the insights gathered from the field (Section 3.8).
- Based on the insights gained throughout our analysis, we highlight how CSP's evolution has affected its prevalence and usage, and propose three actionable steps to improve the adoption of CSP (Section 3.9).

3.2 Methodology

Contrary to prior work that mostly quantified the insecurity of CSP with respect to XSS mitigation, our main focus is to investigate how CSP's changing role affected its deployment. To this end, the Internet Archive (IA) [55] provides a glimpse into the past since, starting from 1996, it archives both content and headers of popular Web sites and makes the information publicly available. Here, we describe how we arrive at a representative dataset of 10,000 popular sites, followed by how we crawl the IA for data related to CSP's evolution.

3.2.1 Dataset Construction

Since widespread support for CSP landed in browsers in 2012 [25], we focus on analyzing the deployment and usage of CSP from 2012 to the end of 2018. However, merely using a current list of highly-ranked domains does not suffice, as many domains may not have been registered (let alone be popular) during the entire period of our study. Moreover, the reliability of lists such as Alexa was recently challenged for exhibiting massive fluctuations on a day-to-day basis [112]. Therefore, we used the following selection strategy for sites: using Alexa lists collected over time [112], for each month between January 2012 and December 2018 we extracted the set of sites that were in the top 50,000 on at least a single day within that month. We then calculated the intersection of these monthly sets, sorting the sites by their average rank (over the entire time).

To limit the overall number of queries to the IA, we opted to study the 10,000 most prolific sites for the given timeframe. To arrive at this set, for each entry in the sorted list of sites, we queried the IA to determine if it contained any archived version of the site. This check is necessary since the IA removes any previously stored content when a site blocks the IA crawler through `robots.txt` [124, 77]. To understand potential biases towards any period within the seven years, for each month we computed the average top 10,000 sites, and compared them to the list we finally chose. This showed that at least 4,960 and at most 6,199 entries overlapped for any given month, averaging at 5,738 domains. Hence, we argue our dataset is well-balanced, allowing us to draw general conclusions on the evolution of deployed CSPs over time.

Given this set of sites, we queried the IA for their daily snapshots from January 1, 2012, until December 31, 2018. Selecting one snapshot per day (where available), we followed any HTTP redirection to reach the final landing page for users accessing the site at archival time. Note that we only used HEAD requests to reduce the load on the IA. Once all redirects were followed, we compared the originally visited site to the final URL. In total, we queried the IA for 10,414,975 snapshots, out of which 10,316,325 yielded final URLs still on the same site. For each of the same-site snapshots, we collected all variants of CSP headers historically used (i.e., `Content-Security-Policy`, `X-Content-Security-Policy`, `X-WebKit-CSP`, and `Content-Security-Policy-Report-Only`), as well as XFO and `Strict-Transport-Security`. The IA prefixes archived headers with `X- Archive-Orig-`, making them easy to differentiate from other response headers [124]. Note that the IA removes CSP directives in `<meta>` elements to avoid interference with the CSP of the IA itself. The CSP deployment occurred after previous research had identified the possibility of maliciously tampering with historical results through the use of externally-hosted scripts [77]. This filtering has a minor effect on our experiments since few sites deploy CSP through a `meta` tag (Section 3.2.2).

Naturally, not all sites have daily snapshots in the IA. On average, a site had snapshots for 1,031 of the 2,557 days in our analysis timeframe. Hence, whenever there is no snapshot for a given day d_i, we use d_{i-1} instead as a basis for our analysis, in a recursive fashion. This means that for every gap in the snapshot data, we use the last entry before that gap as the data point. This approach suffers from a certain level of imprecision, as it might be unclear exactly when a change in a CSP has occurred. This loss of fine-grained information, however, does not significantly affect the class

of observations pursued in our work. For each collected CSP policy, we normalized all randomized elements as nonces are meant to used be just once, and the violation reporting URL may also contain random strings. For those, we removed the random parts, allowing us to properly analyze actual changes. All normalized policies are made publicly avaiable [A5].

3.2.2 Threats to Validity

Given that our analysis uses the IA to extract information, it is prone to the following threats to the validity of the results.

3.2.2.1 Incorrect Archival Data

It is not clear to what extent the data collection process in the IA might influence our results, since a specific browser version might yield a different server response. To determine this IA-specific influence, we chose a second archive service to corroborate the IA's data. In particular, Common Crawl (CC) [31] has been collecting snapshots of popular sites since 2013. For each date on which we found a CSP in the IA, we queried the CC API for a matching snapshot. Overall, we found 38,129 overlapping snapshots for 940 sites. Out of these, 729 (1.9%) on 127 sites were inconsistent between the two archives. For 96 cases the difference was the lack of `block-all-mixed-content` or `upgrade-insecure-requests` in the CC data. Further investigation showed that in the IA, these directives were separated from the remaining CSP with a comma instead of a semicolon. This likely relates to the IA joining headers with the same name with a comma. For those pages, we could always only find a *single* CSP header in the CC response. Moreover, starting from August 2018, these sites still used the aforementioned directives in the IA data, but CC returned two CSP headers (one including only those directives). Hence, we speculate this relates to a bug in CC, which was fixed around August 2018. 23 cases showed evidence for a difference in crawling time; e.g., taking the IA policy from the following day matched the CC. Additionally, 29 differences can be attributed to allowing different edge CDNs based on the crawler's IP. For the remaining 581 cases, the exact cause of the difference was not detectable. Notably though, in only 238/38,129 cases (0.6%) did those policies have a different use case (see Section 2.3.2). Overall, this confirmation from a second source gives us high confidence in our utilized dataset.

3.2.2.2 CSP Through Meta Tags

As mentioned, CSP can also be deployed via HTML meta tags which are currently removed by the IA. To understand the potential impact of this drawback, we crawled the live main page of all 10,000 Web sites from our dataset on June 10, 2019. We collected CSP headers and also checked the content for CSP meta elements. In this experiment, a total of 1,206 sites deployed CSP, and, of those, 78 (6.4%) sites set their policy *only* through a meta element. Of the 1,147 sites that sent a CSP header, 19 *also* set the meta element. Notably though, only 3 with both meta and HTTP header CSPs

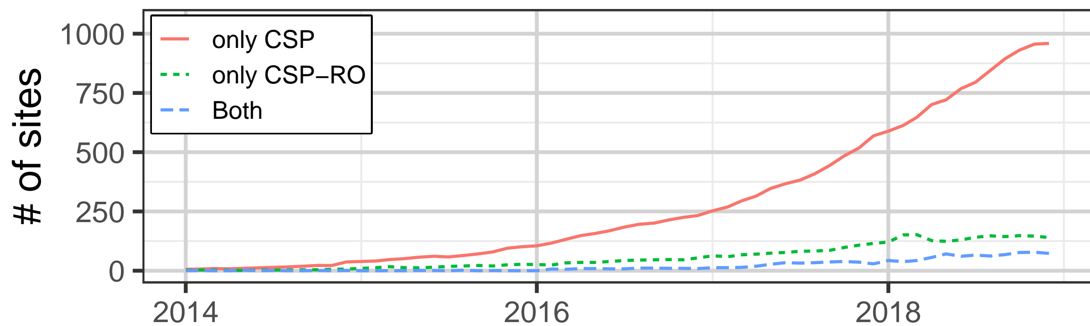

Figure 3.1: Usage of CSP only, CSP-RO only, or both

had policies which differed in their use case (see Section 3.3.2). Hence, we are confident the archived headers provide a valid dataset for our large-scale historical analysis.

3.3 Historical Evolution of CSP

In this section, we provide an overview of how CSP deployment evolved, studying the adoption and maintenance of CSPs, and the changing use cases we observed over time.

3.3.1 Adoption and Maintenance of CSP

Figure 3.1 shows the number of Web sites utilizing CSP in enforcement mode, in report-only mode (CSP-RO), and in both modes in parallel. The figure does not include the CSP adoption from 2012 to 2014, since only 8 different sites deployed CSP before 2014. In our dataset, `lastpass.com` and `adblockplus.org` were the first Web sites to adopt CSP in January 2012, while the other six sites joined in 2013. In total, we find that 1,233 out of the 10,000 sites in our dataset used CSP in enforcement mode for *at least a single day* in our analysis period. Notably though, in the last month of our analysis, only 1,032 domains enforced a CSP.

We draw two main observations based on the plot. First, even though CSP offers the report-only mode to enable developers to experiment with policies before deployment, this mode is not nearly as popular as the enforcement mode. This means that most developers are rolling out policies without having a chance to test them on their user base. We suspect that this is one of the main reasons why CSPs in the wild are so relaxed since they have not been appropriately evaluated and the developers eventually opted for utility over security (see Section 3.4). Second, we observe even fewer Web sites utilizing CSP in enforcement and report-only mode at the same time, which is the preferred way of gradually testing and deploying more restrictive policies. These two observations together suggest that developers are likely confused about the role of report-only and therefore do not take advantage of it.

The plot also shows that the overall adoption of CSP is consistently increasing over time. Given that our list of Web sites remains stable, we can attribute the increased CSP adoption to Web developers deciding to use it, rather than CSP-capable Web sites suddenly climbing in Alexa popularity. Specifically, in 2017 and 2018 anywhere

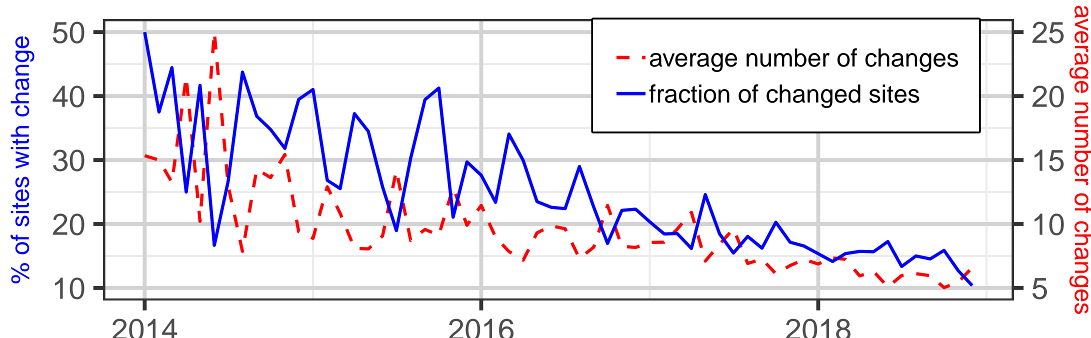

Figure 3.2: Maintenance of CSPs over time

between 18 and 65 Web sites in our dataset were adopting CSP in enforcement mode for the first time every single month. Considering the ever-increasing complexity of Web sites and their deployed JavaScript code [124], the rising adoption of CSP seems to be a positive sign for security. Given the low adoption of report-only, the next sections focus primarily on enforcement mode.

Since CSP is one of many security mechanisms deployed by servers and enforced by browsers (like HSTS [91]) one may think that similar to other mechanisms, once a policy is curated, that policy can be deployed and used for a prolonged period. Unfortunately, CSP — especially for its original use case of script content restriction — is way more complicated than other security mechanisms and requires constant maintenance to ensure that the appropriate sources are allowed so that the site remains operational and secure. Figure 3.2 quantifies the burden of keeping deployed CSPs up to date. The dashed red line shows that, in many cases, sites needed to modify their policies tens of times each month. Even though we see the average number of changes going down towards a steady state, we later show that this is rather caused by CSP being used for non-traditional reasons (such as for TLS enforcement) than by stabilized allow-lists. The blue line shows the fraction of sites with changed policies. We still observe the need to regularly maintain CSPs since by the end of our analysis, still roughly 10% of sites changed their policy at least monthly.

3.3.2 Use Cases for CSP

Even though CSP was initially developed as a measure to mitigate the impact of script injection, the multitude of directives available nowadays allows site operators to control much more than included content. Specifically, we classify policies in the following (overlapping) categories: **Script Content Restriction** (policies using `script-src` or `default-src`); **TLS Enforcement** (policies using `upgrade-insecure-requests`, `block-all-mixed-content`, or allow-listing only HTTPS sources); and **Framing Control** (policies using `frame-ancestors`). Figure 3.3 shows the number of sites applying CSP for the identified use cases over time. When comparing the numbers to Figure 3.1, we find that the increase in the deployment of CSP starting from 2015 coincides with the increased usage of framing control. Similarly, the increase in the overall usage of CSP from 2017 onwards aligns with the increased enforcement

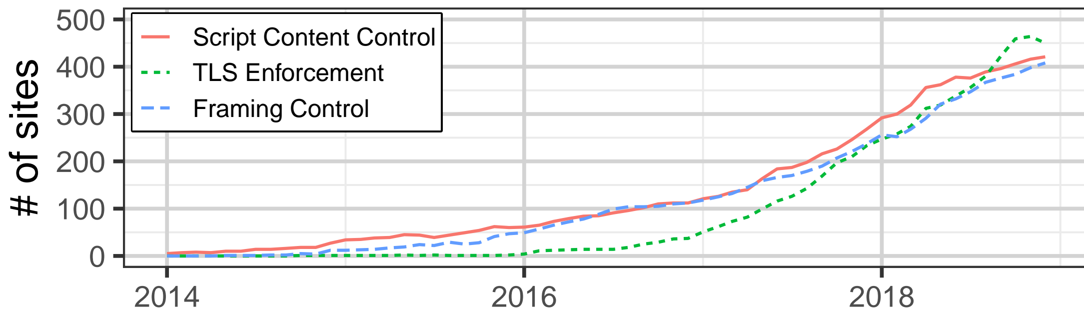

Figure 3.3: Classified policies over time

of TLS connections, mostly through `upgrade-insecure-requests`. Moreover, the decrease in maintenance shown in Figure 3.2 is evidence of easily deployable mechanisms like TLS enforcement, rather than reduced effort to keep policies up to date. This clearly indicates that while CSP was meant as a tool to mitigate script injection, new additions to the set of the CSP directives have shifted CSP into new use cases. In the following, we analyze each category separately, discuss its evolution, and highlight key insights.

3.4 CSP for Script Content Restriction

In this section, we analyze how CSP has evolved with respect to its content restriction capabilities. This not only allows us to confirm findings of prior work through the longitudinal lense of the IA, but also to highlight unknown trends in the increasing trust of operators into lower-ranked domains, to investigate the success of newly introduced CSP features, and to identify previously unexplored attacks related to hijacking allowed domains. Finally, given the unique vantage point of an archival analysis, we conduct a number of case studies which document the long-lasting struggle of Web sites to deploy an effective policy.

The first observation we make is that, out of the 1,032 sites in the dataset that enforced a CSP by the end of our analysis period, only 421 sites shipped policies aimed at restricting script content. This clearly shows that although CSP was initially meant to mitigate script injections, this is only attempted by about 41% of the deployed policies. We now present insights gained from the deployed policies.

3.4.1 Insecure Practices Die Hard

Figure 3.4 shows the evolution in the number of sites using CSP for content restriction, and how many of them have been using various unsafe practices therein. The two most popular unsafe practices are the use of `unsafe-inline` (without the use of hashes/nonces) and the use of `unsafe-eval`. While `unsafe-eval` must be considered the lesser evil, given that its presence does not immediately nullify CSP's security, we observe almost all the policies deployed in 2014 and 2015 made use of `unsafe-inline`. We attribute this to the inflexibility of early versions of CSP. However, it

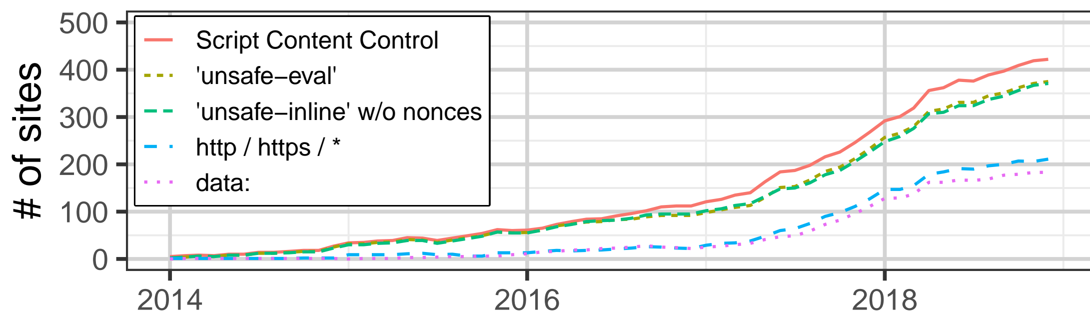

Figure 3.4: Overall adoption of content restriction and insecure practices

is noteworthy that even at the end of our analysis period, this trivially insecure directive is contained in almost 90% of the policies. While we can only speculate about the exact reason for this trend, we opine that it is likely due to event handlers which cannot be allowed with hashes or nonces. Checking merely the start pages for the 378 sites which deployed `unsafe-inline` in December 2018, 180 (48%) of them carried event handlers. The actual number of sites making use of them is likely even higher, but we could not confirm this without adding a significant load on the IA by crawling sub-pages. The bottom two lines of Figure 3.4 refer to the allow-listing of entire schemes. In particular, the first line shows that developers are declaring that any HTTP/HTTPS origin is permitted, which obviously voids security. The second line represents the allow-listing of the `data` scheme, which can be used to add arbitrary code, e.g., through `data:;alert(1)` [84].

Our analysis also indicates that the numerous features added to CSP to ease its secure deployment are not successful. Table 3.1 reports on the adoption of hashes, nonces, and `strict-dynamic` on a yearly basis. Note that, although `strict-dynamic` has recently been shown to be bypassable through *Script Gadgets* [75], we still treat it as an improvement since it should ease CSP deployment. The table highlights that while the usage of CSP to control scripts has constantly grown, neither hashes nor nonces have gained significant adoption. We also find that in both 2017 and 2018, at most 8 sites made use of `strict-dynamic`. While we are only checking start pages and might therefore miss wider-spread deployment, this still highlights that the new directive is not widely used. Overall, we can conclude that insecure practices are present in 90% of policies, whereas secure practices like nonces or hashes, reach less than a 5% adoption

Year	Controls script	Hashes	Nonces	`strict-dynamic`
2014	27	0 (0%)	1 (4%)	0 (0%)
2015	75	1 (1%)	2 (3%)	0 (0%)
2016	135	1 (1%)	3 (2%)	0 (0%)
2017	296	4 (1%)	14 (5%)	7 (2%)
2018	478	6 (1%)	24 (5%)	8 (1%)

Table 3.1: Number of sites per year restricting script content, and using hashes, nonces, and `strict-dynamic`.

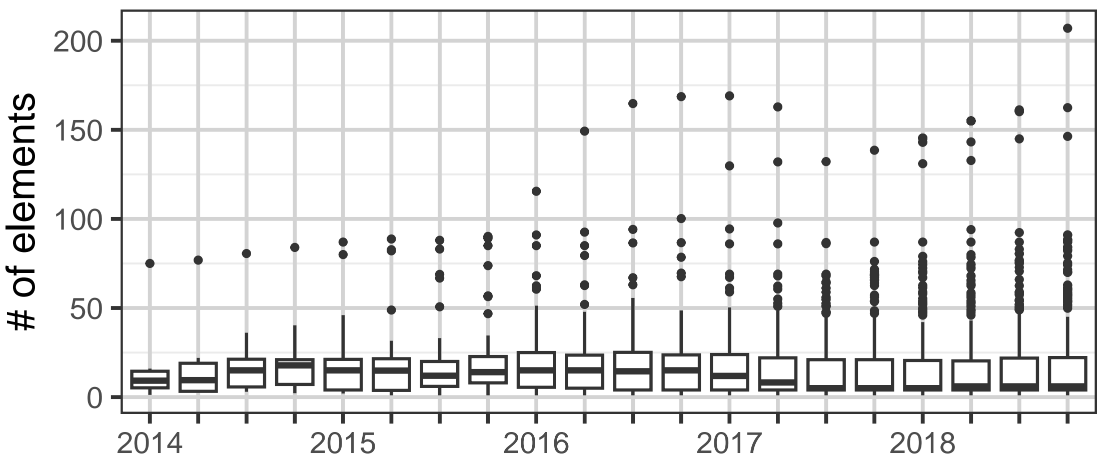

Figure 3.5: Boxplot showing the number of elements in script allow-list

rate.

3.4.2 Allowed Sources

We now complement the findings of Weichselbaum et al. [136] regarding the insecurity of allow-lists by quantifying the evolution of the number of allowed script sources over time (Figure 3.5). We observe that even though the median remains relatively stable, the number and magnitude of outliers expand year after year with some Web sites allow-listing over 200 unique sources for their scripts. The low median must be interpreted in light of the unsafe practices described earlier. allow-listing an entire scheme (such as `https`) and allowing `unsafe-inline` may result in short policies (in terms of the number of entries) which are, however, still more vulnerable than *explicitly* trusting hundreds of remote third parties.

The ranking of allowed sources is an important dimension of CSPs, since site popularity is often used as a proxy for security. This stems from the reasonable assumption that, on average, the developers of more popular Web sites have more know-how and resources to help secure their code. For example, Van Goethem et al. [133] discovered that more popular Web sites tend to utilize more security mechanisms than less popular ones. To analyze this, Figure 3.6 shows information about the ranking of Web sites that are allowed as script sources. In particular, this contains all hosts that were allowed; i.e., even in the presence of `*`, we analyzed the remaining contained sites. We argue that this is useful, given that remote sources contained in the allow-list are *explicitly trusted* by the site, and the existence of `*` often is a byproduct of attempting to curate a limited allow-list (cf. Section 3.4.3.1). For this analysis, we used the publicly available historical dataset of Scheitle et al. [112], extracting the rank from each site on the day when it was allowed. To combat the fluctuation of these lists, we aggregate the results on a monthly basis. We find that starting from 2017, the average rank of trusted CSP sources increases, although the average number of elements in allow-lists does not (cf. Figure 3.5). This means that developers are explicitly trusting less popular Web sites through their CSPs to host JavaScript code, thereby weakening their security.

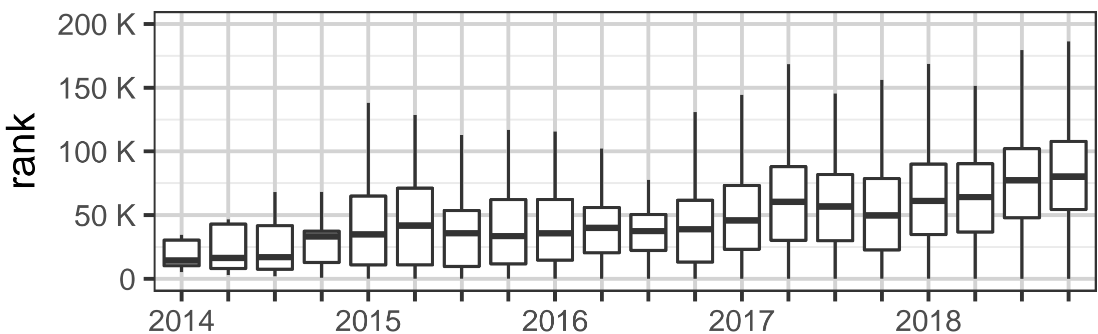

Figure 3.6: Average historic Alexa rank of allowed domains

3.4.3 Longitudinal Case Studies

Contrary to other works which focused on analyzing CSP's adoption and security from a bird's eye viewpoint and at most over a period of a few months, our archival analysis enables us to investigate how sites struggled in their deployment of content-restricting CSPs over the years. In the following, we present our insights into sites that exemplify general behavior and issues we identified in our long-term study.

3.4.3.1 Curating Limited allow-lists

To understand how popular Web sites attempt to curate their allow-lists, we use `airbnb.com` as a motivating example. The site first deployed a CSP report-only (CSP-RO) header on November 12, 2014. While initially 17 sites were allowed (and therefore allowed) to serve script resources, by March 27, 2015, the list of allowed sites had grown to 21. On the next day, however, a new policy was deployed (still in report-only mode), allow-listing the full HTTPS scheme; thereby essentially allowing any site to provide script content to Airbnb. In addition, we observed multiple changes to allowed hashes, indicating that due to changes of inline scripts, these had to be repeatedly modified. The policy was further modified until May 1, 2015, when the curated policy was deployed as an enforcement CSP. This policy was modified three more times until May 9, 2015, when CSP was fully disabled.

The policy was re-enabled on May 22, 2015, and from then up to December 8, 2017, we observed 222 changes to the deployed CSP, merely adding and removing hashes of allowed scripts, while still allowing any HTTPS host to serve scripts (i.e., not actually mitigating the risk of content injection at all). On December 8, 2017, `airbnb.com` started experimenting with changing CSP-RO headers, limiting the sites from which remote scripts could be loaded. The first variant of a blocking CSP with limited sites was deployed on January 13, 2018. However, on January 16, 2018, the policy regressed to trusting all HTTPS hosts for scripts. After another 29 modifications to CSP-RO headers, Airbnb on March 13, 2018, finally deployed a blocking CSP with limited hosts, and a different report-only policy until the end of our analysis timeframe. By then, while only two more sites were added to the enforced list, we observed almost daily changes to the policy, ensuring up-to-date hashes of the used inline scripts.

For both Airbnb and other domains exhibiting the same struggle, we attempted to

attribute changes in the CSP to changes in the page itself, so as to understand if they had been caused through the site itself adding new scripts, or third-party code (such as ads) loading additional dependencies. However, apart from obvious changes in hashes due to changed inline scripts, we could only attribute a negligible fraction of changes in the allowed sites to modified content of the start page (10/106 for Airbnb). For Airbnb, we investigated further, crawling one level of links for each of the changes we could not attribute. Doing so, we were able to attribute an additional 15 changes, but at the cost of making 106,090 GET requests to the Internet Archive. Given this small increase in coverage and our goal of not flooding the IA with requests, we decided to not follow this path of attribution any further.

Even though Airbnb was an early adopter of CSP and made significant efforts to secure their site, they had to often effectively disable their policies for long periods of time. These gaps could have been abused by attackers to launch attacks that would not normally be possible with Airbnb's CSP. Moreover, while they finally managed to curate a allow-list of 33 sites, it took them multiple years from when they started experimenting with CSP to arrive at this final policy.

3.4.3.2 Eventually Giving Up on CSP

Next to those sites which deploy CSP in an insecure variant by allowing any origin to provide script code, our dataset also contains numerous sites which tried to deploy CSP but eventually gave up. Overall, 294 sites used CSP in either mode for at least one day but not within the last month of our analysis. Of those sites that used CSP for at least one month, we find that 227 eventually gave up. Lastly, considering only those sites that made a long-term attempt at CSP (at least one year), we find a total of 79 sites which ended their deployment before December 1, 2018.

`Researchgate.net` was a prominent example of a Web site which moved from CSP to CSP-RO on September 13, 2018, before completely abandoning it on November 27, 2018. Before moving to report-only mode, we observed 29 different attempts at enforcing a working policy. This indicates that `researchgate.net` struggled to build a functional, yet secure policy. In a similar case, `zaycev.net` had deployed CSP since September 2015 and stopped deployment on April 11, 2018. Checking snapshots from March and April, the site itself appeared to be unchanged. Notably though, in the week before CSP was disabled, we observed 3 changes, seemingly to get the non-functioning CSP to stop interfering with the Web site. These examples make it clear that even though sites spend significant effort to deploy CSP, many of them eventually yield to the complexity of keeping their policies secure *and* functional, and proceed to entirely abandon the mechanism.

3.4.3.3 Failing Gracefully Through Report-Only

While prior works have found evidence that CSP report-only is used to test and then deploy policies, our longitudinal allows us to investigate the usage of report-only in a more thorough fashion. In particular, we can identify all those sites which attempted report-only within the five year period for which have extracted CSP data and understand how many of them actually made use of CSP report-only to test and then roll out

CSP-RO for Content	CSP for Anything	CSP for Content	CSP for TLS	CSP for Framing
449	216 (48%)	130 (29%)	78 (17%)	23 (5%)

Table 3.2: From report-only to enforced policies

policies. Even though report-only mode is meant to allow "developers to piece together their security policy in an iterative fashion" [139], it can also be used to determine that the deployment of CSP would cause too much interference with the operation of the application and should therefore not be rolled out; importantly before breaking the application due to blocked resources.

In total, our dataset shows that 449 sites experimented with report-only aimed at restricting content at some point throughout our analysis period. Of those, 233 never actually enforced any type of CSP afterward. This already indicates that CSP's complexity in restricting content deters developers to deploy it after seeing the results in report-only. Next, we analyze to what extent the remaining 216 sites leveraged the tested content-restricting policy in enforcement mode.

Table 3.2 shows the overview of our results. Of the 216 sites that enforced any type of CSP after testing a policy aimed at restricting content, only 130 end up deploying such a policy. The remaining 86 decided to not use CSP for its original purpose, but rather for TLS enforcement (78) and framing control (23), whereas 15 used it for both. Of those 86 sites, 79 had also tested report-only for framing or TLS control before, showing that they merely dropped the content restriction part. Two of the most prominent examples are `aliexpress.com` and `aarp.org`. AliExpress started experimenting with CSP report-only in September 2015, eventually allowing up to 35 sites as well as the usage of inline scripts and eval. In June 2016, they moved to a report-only policy permitting any origin to provide content, but later only enforced an `upgrade-insecure-requests` policy. `aarp.org` went further than this, trying to remove `unsafe-inline` and `unsafe-eval` from their policy altogether. They experimented with this for a month in November 2017 but re-enabled the insecure directives in December 2017. While this policy was still active for report-only in December 2018, the site instead adopted an enforced CSP merely used for framing control.

Overall, our results indicate that more than half of the sites which experiment with any sort of report-only policy never actually deploy one in enforcement mode. This provides clear evidence that the complexity of the mechanism is too high for many site operators. Notably though, we find that even when sites first deploy report-only to restrict content and subsequently move to an enforced policy, only 60% (130/216) keep their content restriction in place, whereas the remaining policies use CSP for the additional use cases, i.e., TLS enforcement or framing control. This shows that such sites could *fail gracefully*, in the sense of understanding that CSP for content restriction would break their site without causing any outages to an enforced policy. Overall, only 29% of all sites that tried script content policies via report-only ended up enforcing such a policy, again highlighting CSP's complexity.

3.4.3.4 Lasting Success in CSP Deployment

We now turn to sites with a long-standing track record of improving their CSP deployment. Three prime examples are `pinterest.com`, `github.com`, and `flickr.com`. Pinterest first deployed a CSP-RO in February 2014 and tested it with an increasing number of allowed hosts until July 2014. At that point, they deployed a policy with 15 allowed sites but still had to resort to `unsafe-inline`. The number of allowed sites went up to 23 sites until May 2017, at which point they deployed nonces and `strict-dynamic` (making modern browsers ignore both `unsafe-inline` and allowed hosts). Notably, even though nonces and `strict-dynamic` remained in the policy until the end of our experiment, the list of allowed hosts still grew over time. This, however, is because legacy browsers would otherwise block content.

When Github first deployed CSP in November 2013, their allow-list contained 5 sites, including a CDN of their hosting provider Fastly and Google Analytics. In October 2014, however, Github stopped using Google Analytics and moved all their assets to two of their own subdomains. By October 2015, those two subdomains were explicitly allowed; but since that time, Github consolidated all scripting resources on `assets-cdn.github.com`, which has remained the lone entry in `script-src`. Notably, Github has never used `unsafe-inline`. This behavior is in line with Github's efforts for securing their service [131, 45].

Our last example, Flickr, first used CSP in enforcement mode in October 2015. While they had a limited host allow-list, they also had to employ `unsafe-inline`. The initial policy's allowed sites grew to 12 by May 2017 at which point Flickr started experimenting with nonces (in report-only mode). After enabling nonces in their enforcement policy in July 2017, they had to go back and forth with nonces three times until finally, since August 2017, they have always deployed nonces. In March 2018, Flickr first sent a report-only policy with `strict-dynamic`. This feature, however, while still active in CSP-RO had not been ported to their enforced CSP by the end of our analysis timeframe.

All three sites have shown a long-lasting commitment to improving security with CSP. These sites, however, are objectively major players on the Web with the ability to spend considerable time and resources on deploying appropriate policies. Even then, while Github is an excellent example of how to deploy CSP in a fully secure fashion, the other two generally positive examples have shown how complicated a deployment can be, especially with regard to removing the need for `unsafe-inline`. What's more, for legacy compatibility (as, e.g., Safari and Edge don't support `strict-dynamic` [80]), even sites with modern features like `strict-dynamic` need to constantly update their host-based allow-list, meaning CSP remains a long-term maintenance burden.

3.5 CSP for TLS Enforcement

The second use case of CSP is to ensure that no content is loaded via HTTP on HTTPS sites. In this section, we first report on how the different means of achieving that goal have evolved over the course of time, and then report specifically on how they were successfully used for TLS migration.

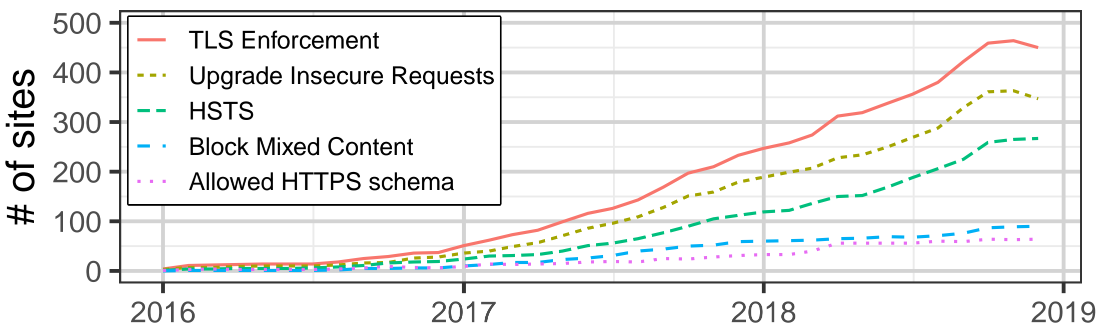

Figure 3.7: TLS Enforcement Strategies

3.5.1 Evolution of TLS Enforcement

Figure 3.7 reports on the trend of deployed CSPs for TLS enforcement. As we first saw policies specifically tailored towards TLS enforcement in 2016, we omit the data before 2016. We observe a steady increase in the usage of CSP for TLS enforcement from December 2016 onwards. Most notably, the enforcement is not pushed forward by exclusively allowing HTTPS sites, but rather by the `upgrade-insecure-requests` directive. Full support for this CSP feature first landed in browsers around March 2015 (with Chrome 43 [138]), meaning site operators did not deploy the directive until about 1.5 years after it became available. The related directive to block all mixed content has also increased in usage, but only sees adoption on a fraction of sites compared to `upgrade-insecure-requests`. Notably, we found that 51 sites make use of both directives in December 2018, even though in modern browsers `block-all-mixed-content` in the presence of `upgrade-insecure-requests` is effectively a no-op because the upgrade to HTTPS happens first. We find the decreasing number of sites deploying CSP for TLS enforcement in December 2018 is not a downward trend; a brief check for January 2019 shows that usage keeps increasing.

Figure 3.7 also compares the usage of a policy aimed at ensuring that no content can be loaded over HTTP with the usage of the HTTP Strict Transport Security (HSTS) header on those sites. The HSTS header is set to ensure that once a site has been visited over HTTPS, it cannot be loaded over HTTP until a specified timeout occured [52]. In contrast, the CSP directives ensure that any resources that are included by the original site will be loaded over HTTPS. We observe that alongside the increasing deployment of `upgrade-insecure-requests`, the fraction of sites using HSTS also rises. Notably, of the 450 sites that enforced transport security with CSP in December 2018, 267 made use of HSTS (59%). This uptake in HSTS adoption is a positive trend in line with the more widespread use of HTTPS on the Web [103].

Considering the 347 of those sites enforcing TLS explicitly via `upgrade-insecure-requests`, 194 made use of HSTS. This provides us with an interesting insight: the `upgrade-insecure-requests` directive is deployed to ensure that once the connection has been securely established, no resources or other URLs can be accidentally loaded via HTTP. HSTS, in contrast, is used to ensure that the site cannot be loaded via HTTP in the first place. While for a secure setup, both mechanisms are desirable, deploying HSTS comes with a loss of control for the operator. Once a client has

observed the HSTS header, it will refuse to connect to the site via HTTP for a set amount of time controlled by the received HSTS header. Given that activating HSTS is no more complex than shipping `upgrade-insecure-requests` we posit that its absence by half of the sites using the directive has more to do with the developers being uncomfortable fully giving up HTTP than with issues regarding HSTS setup and deployment.

3.5.2 Leveraging CSP for TLS Migration

While one of the goals of CSP is to *enforce* TLS, it also provides a meaningful aid for developers when migrating to HTTPS. When mixed active content is detected by the browser while visiting an HTTPS site, it is automatically blocked. However, `upgrade-insecure-requests` instructs the browser to gracefully upgrade the connection, rather than blocking all HTTP resources, which is particularly useful if an HTTPS site still contains references to HTTP-based active content (like scripts). To understand which sites leveraged this added benefit of CSP, for each of the 347 sites that deployed `upgrade-insecure-requests` within the last month of our analysis, we determined when each started to use the directive. Subsequently, we downloaded the snapshots for those site's main pages from the IA for 31 days after the first `upgrade-insecure-requests` deployment.

Based on the snapshots we collected *before* `upgrade-insecure-requests` was deployed, we find that 251 (72.3%) sites deployed the directive as part of a transition to HTTPS. We base this observation on the archived URL, which indicates if the site had been loaded via HTTPS by the Archive's crawlers. For those sites, we parsed the HTML of all collected snapshots once `upgrade-insecure-requests` was deployed and extracted the URLs of external scripts, images, frames, and stylesheets. On 77 sites, we found that within a month from originally deploying `upgrade-insecure-requests`, resources were still linked via HTTP. Among these sites, we found high-profile pages such as `wired.com`, `airasia.com`, and `aol.com`. For those three sites, we further investigated how much longer they linked to HTTP resources, downloading all snapshots until December 31, 2018. For Air Asia and AOL, on the last day of our experiment, there was still one HTTP resource on their main page, even though both sites moved to HTTPS in 2017. Wired, which started deploying HTTPS in June 2016, removed the last HTTP resource from the start page only in September 2017.

Overall, these results strongly indicate that `upgrade-insecure-requests` is a useful mechanism and therefore widely used when a site migrates to HTTPS, highlighted by the fact that more than 70% of sites deploying `upgrade-insecure-requests` for the first time did so as part of their move to HTTPS. In addition, 77 sites still made use of HTTP-linked resources after their move to HTTPS: `upgrade-insecure-requests` allowed these sites to function correctly due to the graceful upgrade to HTTPS.

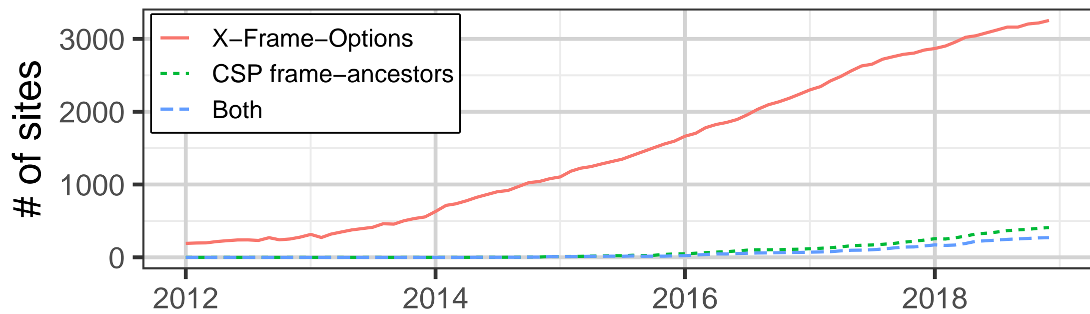

Figure 3.8: Evolution of `X-Frame-Options` and `frame-ancestors`

3.6 CSP for Framing Control

This section presents our historical findings for the third use case of CSP, i.e., framing control. In particular, we investigate to what extent CSP's `frame-ancestors` has achieved its goal of deprecating the underspecified XFO header, by analyzing sites that leverage its flexibility, sites that deploy both headers, and highlighting numerous cases in which `frame-ancestors` would be required to achieve proper protection.

3.6.1 Evolution of CSP for Framing Control

As discussed in Section 2.3.1, XFO was not standardized before it was put into browsers, leading to inconsistent enforcement, which in turn enabled double-framing attacks. This is why CSP removes any ambiguity and checks all of a frame's ancestors. The high-level overview of our analysis on the use of CSP for framing control against XFO is shown in Figure 3.8. We observe a steady increase in the usage of XFO until the end of our analysis in December 2018. In that month, 3,253 sites made use of XFO, whereas only 409 used `frame-ancestors`. Moreover, out of the sites that use CSP for framing control, 270 do so in combination with XFO. In the following, we first analyze to what extent the flexibility of `frame-ancestors` is used by sites, i.e., we determine if the same allow-list could also be expressed with XFO. Second, for those sites that make use of both CSP and XFO for framing control, we analyze how the goals of the deployed headers differ. Finally, we report on how many sites would *have to* move to CSP to achieve universal protection.

3.6.1.1 Leveraging the Flexibility of CSP

The protection offered by XFO is coarse-grained both because of the small number of configuration options, but also because of Google Chrome's decision (shared by all Chrome-based browsers) to not support the `ALLOW-FROM` directive. As part of our analysis, we wanted to determine which sites needed the additional flexibility offered by CSP's `frame-ancestors` directive. We aggregate these by year, as this shows a clearer trend compared to monthly analyses. The result is shown in Table 3.3. We find that starting from 2015 when a meaningful number of sites adopted `frame-ancestors` (shown as CSP-FA), at least 50% of them required the flexibility of CSP.

Year	Used CSP-FA	Required CSP-FA
2014	13	3 *(23%)*
2015	60	32 *(53%)*
2016	133	92 *(69%)*
2017	260	182 *(70%)*
2018	460	321 *(70%)*

Table 3.3: Number of sites per year that set a frame-ancestors directive, as well as number and fraction of sites that set policies not expressible by X-Frame-Options

For those, we find that the most common pattern is allowing all origins from the same site (104 of 321 in 2018). Moreover, manual investigation of the remaining cases showed that CSP was often used to allow-list sites from the same company (e.g., `icloud.com` allows to be framed from `*.icloud.com` and `*.apple.com`). Hence, apart from delivering different XFO headers depending on the Referer or Origin header, these sites could not run uninhibited without `frame-ancestors`.

3.6.1.2 Combining XFO and CSP frame-ancestors

When an XFO header and a CSP `frame-ancestors` directive are both set, CSP-compatible browsers are supposed to ignore XFO. However, in older browser versions, XFO was the only mechanism to protect users from clickjacking. Based on this insight, we compared the semantics of XFO and `frame-ancestors` (CSP-FA) in cases where both were present, so as to understand how site operators dealt with different browsers. Over the whole period of time, 394 Web sites made use of both XFO and `frame-ancestors` at least once on the same day. Out of those, 290 did so inconsistently at least once, i.e., XFO and CSP were not semantically equivalent (e.g., did not combine `SAMEORIGIN` and `self`). On 70 sites, `frame-ancestors` was used to relax the security boundary from the same *origin* to the same *site*, e.g., `https://site.com https://*.site.com`, while the XFO header was set to `SAMEORIGIN`. 185 sites used CSP (at least once) to relax the security boundary even further than the same site, leveraging `SAMEORIGIN` for older browsers. Notably, these sites made the best of a bad situation, given that browsers without support for `frame-ancestors` are at least more secure than when XFO is absent. Only twice was CSP used to deploy a more restrictive policy; specifically going from the invalid XFO value `ALLOWALL` (essentially disabling XFO) to allowing the site and all of its subdomains. Here, the site operators opted for an insecure solution for IE users, which is the only current browser that does not support CSP framing control. Finally, 65 sites deployed an invalid XFO header alongside CSP at least once, e.g., by using contradicting values like `SAMEORIGIN, DENY`, whose semantics on legacy browsers is unclear.

3.6.1.3 Replacing XFO with CSP-FA

It is worth noting that even though the double-framing attack is mitigated in most browsers since 2017 [86], using directives that are not supported by all browsers (e.g.,

`ALLOW-FROM` for Chrome and Safari) can have dangerous consequences. For those, setting an `ALLOW-FROM` directive in XFO results in the header being ignored completely (failing insecurely [73]). We found that in December 2018, 116 sites in our dataset made use of a non-universally supported directive, meaning the developers would have to deploy CSP's `frame-ancestors` to properly secure their sites. Notably, of the 3,253 sites that use XFO in December 2018, but not `frame-ancestors`, 362 already deploy a CSP. Hence, even though these sites would merely have to add a CSP directive to make use of a more fine-grained and consistent framing control, they still *only* resort to the deprecated security header. While one could argue that this might originate from old, outdated CSP policies, we find that 120 of these sites made use of `upgrade-insecure-requests`: a directive that was only added after support for `frame-ancestors` was enabled.

3.7 In-Depth Analysis of Deployed CSPs

We now attempt to shed light on the main reasons which may affect a successful adoption of CSP. First, we focus on sites that gave up CSP and aim to attribute this choice to its cause. Second, we highlight characteristics of sites which kept running trivially insecure policies for content restriction, before finally investigating sites which deployed effective policies.

3.7.1 Reasons for Giving Up on CSP

We start by investigating whether we can attribute changes in the deployed CSPs to violations of said policies. Specifically, we focus on those sites which tried out CSP in enforcement mode for at least one month, but gave up. We define "giving up" as not having had a policy for a specific use case in all of December 2018. This way, we lower the risk of incorrectly flagging a site for having given up just because it did not have CSP for the last few days of 2018. For each domain, we investigate the final policy snapshot for each use case, namely content restriction, TLS enforcement, and framing control. Out aim is to understand why a given domain stopped using CSP in a particular capacity.

3.7.1.1 Content Violations

CSP's initial purpose was to restrict the inclusion of content into a page. Hence, our first analysis focuses on sites that gave up this use case as per our earlier definition. For each site, we determined the exact timeframe for which the last policy was deployed. We then visited the start page on the last day of policy deployment, following all links to the same site that were archived in the timeframe of the last deployed CSP. While all of our previous analyses relied on downloading and statically parsing HTML documents, for this experiment we relied on an instrumented Chrome browser. This allowed us to not only determine if the resources statically linked in the document caused violations, but also observe if dynamically-added content interfered with CSP. We limited the analysis to the first level of links (in total, 3,347 URLs), as this already required us to make an additional 421,684 requests through loaded scripts, images, fonts, etc.

Overall, we found 63 domains which attempted content restriction, but eventually gave up. For 15, we found violations of the deployed CSP through dynamic analysis. When purely relying on the resources statically linked in the HTML document, only 7 sites indicated a violated CSP, which shows the benefits of the dynamic analysis. Unfortunately, the Archive does not always manage to correctly rewrite URLs to included resources, as shown by Lerner et al. [77]. Hence, it is likely that even the more accurate dynamic analysis missed at least a few violations, merely due to the fact that third-party code was not even executed. Of the 15 sites with violations, we analyzed which party included the violating resources. On 9 pages, the violation was caused by third-party code, whereas on 8 they were caused by first-party code, with an overlap of two domains having both first- and third-party-caused violations.

On the remaining 48 sites for which we could not find violations, we performed two analyses. First, since we cannot reason about violations that are deeply hidden in the application, we performed a live experiment on September 24, 2019, in which we crawled the live versions of all websites in our data set. For sites with a CSP on the start page, we randomly sampled 10 same-site subpages and checked their CSPs. In doing so, we found that of the 1,202 sites with CSP on the start page, 1,024 (85%) appear to have a site-wide deployment of the same policy. Hence, assuming a similar distribution of site-wide policy deployment for the archived versions, the abandoning of site-wide policies may very well be due to violations on pages other than the ones we crawled. Eventually, we resorted to manually checking the deployed policies to classify them, so as to provide an educated guess about the reason for dropping CSP. For 25 sites, we found that their policies showed increasing numbers of third-party entries (e.g., `milanoo.com` and `snai.it`). For those, it is likely that the overhead of keeping a allow-list of their dependencies was too burdensome for the operators. For another 9 sites, the policies were trivially insecure (e.g., `raspberrypi.org`) before they were dropped. Here, we argue that deploying insecure CSPs can eventually lead to their removal.

Summing up, though our archival analysis can only provide glimpses of the reasons why site operators gave up CSP, we find that more than half of the sites for which we could find violations had these caused by third parties. For the rest, even though we could not find specific violations, a significant fraction had large allow-lists with tens of third parties, indicating that reliance on third parties could well be the major reason behind the sites' decision to abandon CSP.

3.7.1.2 TLS Violations

To understand whether a TLS enforcement policy was violated, we need to check multiple angles. Trivially, if a site includes a resource via HTTP and has `block-all-mixed-content` or `default-src https://*` (or equivalent for other resource types), the CSP is violated. Contrary to this straightforward case, understanding if `upgrade-insecure-requests` could have caused incompatibility is more involved, as the Archive crawler does not honor `upgrade-insecure-requests`, i.e., would not automatically archive the HTTPS variant of an upgradable resource. As a first step, we assume that if a site was delivered over HTTPS, all resources from the same host would also be available via HTTPS (e.g., the page was `https://foo.com` and

the resource `http://foo.com/bar.png`). We do not look for violations in this case, to minimize the risk of false positives. We then leverage the observation that widely-used resources which are available over both protocols would have been loaded and archived by the crawler at least once over a secure connection, as at least *one* site would likely have included it via HTTPS. We exploit this by querying the CDX API for the HTTPS variant of the URL we want to check for upgradability, limiting the results to URLs archived within ±30 days of the HTTP resource. If we can find an HTTPS variant, we mark the resource as upgradable. We then flag all remaining resources, i.e., those with only HTTP snapshots in the IA, as non-upgradable. Though this approach might not be a perfect solution, it is the best option considering the Archive limitations. Hence, whenever a resource is non-upgradable, but is included in a site with `upgrade-insecure-requests`, we say that the site's policy is violated.

In total, 46 domains were labeled as having given up TLS enforcement. Similar to content violations, we used Chrome to crawl the first level of links beginning from the last snapshot with CSP. For 28 of the sites, we detected a non-upgradable resource on the crawled pages. In addition, for 4 domains, TLS enforcement seemed to have been dropped along with all other CSP directives, i.e., was collateral damage. For the remaining 14 domains, we could not reach a definitive conclusion. However, given the insights about site-wide deployment we discussed in the previous section, we plausibly expect that many of these sites may have had at least one non-upgradable resource on subpages we did not crawl.

3.7.1.3 Framing Control Violations

To understand if a given CSP would have caused a framing violation, we would have to retroactively investigate which other sites framed a given page. As this is not feasible, we instead resort to a heuristic to determine if the removal of `frame-ancestors` was due to framing control issues. As we have seen before, `frame-ancestors` is often used in combination with XFO. Hence, if a given site has encountered an issue related to framing control, it would likely not only remove `frame-ancestors`, but also drop or adjust XFO. Therefore, once a site has stopped using `frame-ancestors`, we check its XFO status on that same day, as well as on the next snapshot. If a site has also stopped using XFO, it is extremely likely that framing control in general proved to be a problem.

In our dataset, 69 domains used CSP for framing control, but gave up on that use case. Given our above classification, we found that for 42 sites, restricting framing in general proved to be problematic, i.e., they dropped both `frame-ancestors` and XFO at the same time. In addition, we found 7 sites which moved from explicitly allowing a hostname through both CSP and XFO to only XFO `SAMEORIGIN`. These cases are interesting, as they indicate that developers determined this to be sufficient to constrain framing (as the flexibility of CSP was not necessary); notably showing the lack of awareness of the dangers of double-framing attacks. In another 7 cases, `frame-ancestors` was removed as collateral damage, i.e., XFO was used before and after CSP's removal. Surprisingly, we also observed two sites moving from exclusively using `frame-ancestors` to XFO, indicating those operators were also not aware of the drawbacks of XFO. Overall, we find that sites do not give up on `frame-ancestors`

for reasons specific to CSP, but rather because they either find framing control too cumbersome, or altogether unnecessary.

3.7.2 Investigating Insecure Policies

As our results have indicated, around 90% of sites that tried to restrict content did so insecurely, e.g., by using `unsafe-inline` or allowing entire schemes. To understand the reasons behind this, we specifically looked at the content of all pages which deployed such insecure policies, and were never able to remove those unsafe keywords. We discovered 467 websites exhibiting this behavior. For each of the websites, we checked every snapshot from the Archive (totaling around 118K requests) for the presence of inline scripts, event handlers, and the number of third parties in the page.

Overall, 455 sites (97%) had inline scripts on the start page at least once while running an insecure policy. Moreover, 317 used event handlers (68%) and in the median, each site relied on 3 third parties (with a maximum of 26 third parties for a single site). It is worth noticing that all these numbers likely represent lower bounds, as we did not crawl the sites any further. Nevertheless, a staggering 68% of sites relied on event handlers, meaning they could not deploy a policy without `unsafe-inline` given the current CSP specification. The results also highlight the difficulty that operators face when trying to retrofit CSP; essentially, a policy that is tacked onto an existing application is virtually always insecure.

3.7.3 Analyzing Secure Sites

To complement our previous analysis, we now focus on sites which managed to deploy a secure policy, i.e., one without allowing entire schemes or using `unsafe-inline`. While prior work [136] has indicated that additional risks may originate from allowing origins with JSONP endpoints or allowing Flash to be hosted locally, we do not consider these additional factors. In total, we found that 40 sites were able to deploy a meaningfully secure policy and still have that in operation at the end of our analysis timeframe. Notably, another 7 at some point deployed a strict policy; however, they either added the unsafe keywords again or entirely disabled CSP after mere days, indicating their policy caused functionality issues. In particular, for 3 sites we found event handlers on their start pages, even though their policy did not specify `unsafe-inline`, hence definitely causing a CSP violation.

Of the 40 sites which can be counted as successfully having deployed CSP for content restriction, 2 actually run policies which interfere with scripts on their start pages as of this writing. When looking at the other 38 cases, we discovered an interesting trend. First, we found 16 adult websites, most of which deployed a strict policy without attempting a more relaxed one before. Interestingly, they all had starting days of their first CSP about 1-2 weeks apart (each). Analyzing the CSPs, we found that they were all allowing the exact same sources. Looking at the start dates of CSP deployment, we found that the operators of these sites first experimented on one site with removing the event handlers on the page, exclusively used to track users through Google Analytics. Notably, this behavior of using inline event handlers was even advocated for by Google [47]. Once they had successfully rolled out CSP for one website, they

proceeded with others. Of the remaining 22 sites, only 3 had any event handlers on the last snapshot before the deployment of the strict policy.

Overall, we find that of the few sites that were able to deploy a strict policy, virtually all either did not rely on event handlers (on their start page), or only used event handlers for a single, easy-to-change use case (such as registering event handlers programmatically for off-site links). This stands in stark contrast to the results for the sites which failed to deploy a secure CSP, where over two thirds used event handlers.

3.8 Framing Control Notification

In general, we observed that sites have a clear preference for XFO over `frame-ancestors`. Moreover, we found cases where XFO was used even though the site deployed directives only introduced after `frame-ancestors`, indicating the CSPs were updated when framing control was already possible. To understand the reason behind these findings, we decided to notify site operators running XFO to inform them about the improved support that CSP's `frame-ancestors` offers, and tried to discover their reasons for preferring XFO. To this end, we checked all live versions of the sites in our dataset starting from May 31, 2019, for their deployed XFO and CSP directives. On June 4, we notified all 2,699 sites that used XFO headers which either had a syntactically incorrect header, used the non-universally-supported `ALLOW-FROM` directive, or deployed `SAMEORIGIN`, making them prone to double-framing attacks in some browsers. We did not notify the sites that also made use of CSP's framing control, since supporting browsers ignore any XFO headers when `frame-ancestors` is present. Given the insights from prior work on Web notifications [125], we chose to send emails to generic aliases on each domain (info, security, webmaster) as well as to the WHOIS contact (where available). The template of our email can be found in Section A.1. We sent this email from one of the researchers' regular email address, ensuring that recipients could verify our identity. As expected, in line with prior work's findings [125], most emails bounced, either due to non-existing addresses or lack of appropriate MX records.

3.8.1 Insights from Initial Responses

Notably though, we received responses from 117 sites which went beyond automated confirmation emails, such as out-of-office responders or confirmation of a created ticket. By categorizing these responses, we discovered that 62 operators claimed that they would deploy `frame-ancestors` shortly. For a sample of anonymous answers we received, please consult Section A.2. Among the responses, we also found 24 answers which indicated that CSP was too complex to be deployed. In particular, they all claimed to have attempted to deploy CSP for content restriction, but either deferred it, or abandoned the attempt altogether. This is in line with the significant number of sites we discovered in our archival analysis, which either stopped deploying CSP or never moved from report-only to enforcement mode (see Section 3.4.3).

With all respondents, we exchanged further emails, indicating that CSP's `frame-ancestors` could be used without any of the other CSP functionality. In doing so, we

received emails from 16 operators stating they were not aware of any issues related to XFO, and 13 who explicitly noted they had not heard of CSP's `frame-ancestors` before. In contrast, 9 informed us in their initial response that they had already deployed the CSP directive. From the notification date and onwards, we continued our daily checks for both XFO and `frame-ancestors`. Overall, we observed an increase from 511 sites deploying `frame-ancestors` before our notifications to 554 sites by June 12, 2019. In particular, for the domains that answered to our initial message, 14 had taken action. Moreover, for the other sites, 4 belonged to a network of sites for which we had received one response. For the remaining 25 sites that rolled out `frame-ancestors` in the 8 days, we could only find two sites for which *all* our sent emails bounced. Hence, we believe that most of the sites deployed `frame-ancestors` as a result of our notification, demonstrating the ease of deployment within mere days.

Finally, in conversations with operators, several mentioned that they relied on external resources for security headers. In checking those resources, we found that they all list XFO as the only defense against framing-based attacks, whereas they advertised CSP as a means to mitigate XSS attacks [17, 40, 113, 3, 135]. Notably, even widely-used sites like `securityheaders.com` consider XFO the only viable option for framing control. Neither this service nor other resources like MDN [83] indicate that CSP can be used for this purpose.

3.8.2 Follow-Up Survey

Given the diverse responses we obtained from the notified site operators, we decided to run a more systematic survey, allowing us to ascertain the number of operators aware of issues with XFO, CSP, and the fact that `frame-ancestors` could be used in isolation. We made the survey as brief as possible and only sent it to operators who had previously answered our initial email, with the explicit goal of soliciting a high fraction of responses due to the limited effort necessary to answer the questions. In particular, as prior works have shown, unsolicited surveys have minuscule response rates [125, 34], which is why we decided to only reach out to operators to whom we had previously provided helpful information. The full questionnaire is available in Section A.3. For our survey, until June 12, 2019, we received a total of 39 answers. Out of those, two thirds (27) indicated they were not aware of the inconsistencies around XFO. When asked about why they had deployed XFO in the first place, the majority (20) said they had their own reasons to restrict framing, indicating the awareness of framing-based attacks. Moreover, 31 (79%) respondents indicated that they had been aware of CSP before our notification; yet only 12 of those claimed to have been aware of `frame-ancestors` beforehand. Of those 12, 9 claimed to know that `frame-ancestors` can be used in isolation. These reports suggest that while operators have a general understanding of CSP, they are not aware of all its directives and their security benefits.

For all respondents that indicated to have known about CSP beforehand, 23 said that their site would not work with a reasonably secure policy right away (2 claimed yes, 6 did not know). On the flip side, 29/31 operators believed that CSP could be a viable option to improve their site's resilience to XSS attacks. Additionally, when asked about the use case of TLS enforcement, 22 responded that they knew they could

operate TLS enforcement in isolation before our notification. Hence, it appears that while content restriction is clearly known as a goal of CSP, most sites are unable to deploy it due to its complexity. Operators seemed to be more aware of the fact that TLS can be enforced through CSP, but were not as knowledgeable about framing control. This, combined with the insights from resources the respondents indicated (both in the survey and the email conversations) leads us to conclude that resources on CSP critically lack details about framing control.

In terms of tool support, 36/39 respondents answered that they had used the browser console to debug and analyze their site. Hence, if there had been warnings about inconsistencies (or even lack of support for certain directives), those operators would likely have taken action. With respect to required tools, the respondents named better tools to debug CSP errors (locally), improved collection and aggregation of warnings caused in users' browsers, and in general tools to suggest appropriate security headers. As a result of these insights and separate discussions with Google engineers, we filed a Chrome feature request to issue warnings about XFO; in particular to at least warn operators about the unsupported `ALLOW-FROM` directive and suggest to deploy `frame-ancestors` instead.

3.8.3 Limitations and Additional Survey

Our notification and subsequent survey cannot be considered an in-depth analysis due to its unstructured nature (especially of the emails we received). We specifically set up the survey to be brief, so as to achieve a high response rate. However, it is not clear whether security-aware operators filled our survey. Even then, our results are indicative of operators which did not use CSP for framing control, i.e., cannot be considered experts in CSP. While we cannot account for these facts in our initial survey, to partially alleviate the identified shortcomings, we ran our survey a second time after having presented a talk about the evolution of CSP and its different use cases at an OWASP conference. For this, we are confident that professionals with a Web security background answered the questions. We received a total of 20 responses with 10/20 claiming prior knowledge of XFO's shortcomings, and 19/20 being aware of CSP beforehand. 18 of those believed CSP to be a viable option, of which 9 argued their site would be able to run a secure policy. Regarding framing control, 13/19 said they already knew about `frame-ancestors`, of which only 2 did not know that it was feasible to deploy in isolation. In contrast, 9/19 were not aware that TLS enforcement could be deployed in isolation, indicating that this is a little known use-case even among security experts. Naturally, as for our initial survey, we cannot assess incorrect reporting from operators. Nevertheless, as we stressed the anonymous nature of our survey, we expect the results to be characteristic of the participants.

3.9 Discussion

We now summarize the evolution of CSP's use cases, enabled by the unique vantage point of the IA, and highlight gathered insights. We then discuss whether CSP is too complex for site operators, and outline how it can become more useful going forward.

3.9.1 Summary of CSP's Use Cases

This section discusses how CSP for script content restriction has evolved, but more importantly also how CSP is used more frequently for purposes other than its original goal.

3.9.1.1 CSP for Script Content Restriction

Our work has confirmed a previously investigated fact [136, 137, 19, 21]: CSP is largely failing as a defense mechanism for script content restriction. Through our longitudinal analysis, we could show that although nonces and hashes have been available since 2014, they have not gained significant popularity over time. On the flip side, most policies make use of `unsafe-inline`, which makes them trivially bypassable by XSS. We argue that this is due to the complexity of deploying CSP in a secure fashion. This is evidenced by the fact that more than half of the Web sites (251/449) which experimented with report-only never switched to enforcement mode. It is also confirmed by the notification responses, in which operators regularly stated they had experimented with CSP, but felt it was incompatible with their application. This anecdotal evidence is supported by our survey, in which only 2/30 respondents familiar with CSP claimed their site could deploy it right away without breakage.

In terms of deploying limited allow-lists, we saw ample evidence in our case studies as well as the general uptick in allowed sites that curating such allow-lists is challenging. Moreover, our analysis of the allowed sites has indicated that operators are prone to add typo domains, or leave unregistered domains in their allow-list, effectively undermining the provided security guarantees (on approx. 13% of the sites with content-restricting CSPs). Of the handful of sites which managed to actually deploy a restricted allow-list, both the case studies and the feedback from our notification indicated that curating such a list takes months or even years. Hence, the overall effort of setting up and maintaining a secure policy seems unbearable to all but the biggest players.

3.9.1.2 CSP for TLS Enforcement

Previous studies mainly focussed on CSP as a means to restrict script content, treating TLS enforcement and framing control as side notes. In particular, Weichselbaum et al. [136] reported that only 3% of policies were used to enforce TLS, while Calzavara et al. [21] reported that around 0.5% of the Top 1M used `upgrade-insecure-requests`, without providing further details. Our longitudinal analysis showed that CSP is a very valuable tool for TLS enforcement, being used by about one third of the Web sites that deployed CSP. Most prominently, we observe that 347 sites make use of the `upgrade-insecure-requests` directive to automatically upgrade HTTP resources to HTTPS. We find that this feature is not only used for security purposes but when investigating those sites that deployed `upgrade-insecure-requests` as part of their migration to HTTPS, we found that 77 of 251 (31%) sites still link HTTP resources (on their start page). Here, the added benefit of `upgrade-insecure-requests` enables browsers to upgrade URLs before trying to load them, thereby avoiding mixed content warnings or blockage. Given the increasing adoption of HTTPS [103], we argue that Web sites

adopting an `upgrade-insecure-requests` policy would have an easy migration to HTTPS, while at the same time not having the burden of making their applications compliant with a strict content-restricting CSP (e.g., by removing event handlers). This fact, however, according to our survey, is less than well-known.

3.9.1.3 CSP for Framing Control

In contrast to previous studies, our findings indicate that CSP is becoming increasingly popular for framing control, now on par with content restriction (attempts) and TLS enforcement. At the same time, the adoption of CSP for framing control is not nearly as widespread as XFO: CSP with `frame-ancestors` is used in 409 Web sites, while XFO is present on 3,253 Web sites as of December 2018. However, we observe that existing Web sites are taking advantage of the additional flexibility on framing control offered by CSP. In fact, out of the 460 sites using CSP for framing control in all of 2018, 321 sites (70%) used allow-lists not expressible by XFO, which suggests that the additional expressiveness of CSP for clickjacking protection is useful in practical cases. Moreover, our notifications showed that about two thirds of respondents were not aware of the added benefit of `frame-ancestors`. While our notifications and the feedback we have received suggest that this can be easily changed for operators we could reach, the resources frequently used by the respondents lack crucial information about this fact. As an example, the Mozilla Developer Network only explicitly mentions "[..] certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks" and does not list framing control as a use case [83].

3.9.2 Complex Security Policy?

From both the evidence gathered from our longitudinal study as well as the insights provided to us through the conversations in our notification, CSP for script content restriction seems to be a failing mechanism. Even though modifications to CSP should have made it easier for sites to adopt secure policies (e.g., allowing inline scripts through nonces), we could not observe any significant uptake over time. Moreover, the responses indicated that operators still shy away from CSP for content restriction due to its perceived complexity.

One specific issue that CSP for content restriction has is the moving target it represents. As an example, `google.com` stopped using `strict-dynamic` on July 17th, 2018 even though Google engineers originally proposed the new directive [136]. Given the insights shared by the Google team in a recent presentation [135], they have since moved away from using `strict-dynamic`, favoring the explicit propagation of trust by having nonced scripts attach the nonce whenever they add additional scripts. The reasons are seemingly two-fold: first, support for `strict-dynamic` is not universal (e.g., Safari does not support it). Second, using `strict-dynamic` yields any control over which resources can be included, and opens up the potential for Script Gadget attacks [75]. The loss of control over included resources cannot be addressed through a nonce-based policy, given that any nonced script could just use the nonce to add code from elsewhere. The solution, as also proposed by the Google engineers and at least partially deployed as of now by Dropbox, is *policy composition*. There, a site sets two

CSPs, where one carries a nonce and the other carries a allow-list. Since *both* have to be fulfilled, only scripts that carry the nonce *and* are from explicitly allowed hosts can be executed. Given these advances and frequent changes in suggested best practices, it is understandable that operators feel overwhelmed by the complexity of the mechanism.

At the same time, while CSP appears to be failing as a means to mitigate XSS, it has become a successful mechanism to enforce TLS, evidenced by the uptick of this use case. Moreover, `upgrade-insecure-requests` allows sites to seamlessly migrate to HTTPS by upgrading all URLs in-flight. Notably, out of the 10K sites in our original dataset, 7,675 were archived via HTTPS. We downloaded the final snapshot for each domain from the IA and found that of the 7,328 sites without `upgrade-insecure-requests`, 435 had a least one HTTP-linked resource (6%). Given the current implementation of browsers, these sites would at the very least trigger a mixed content warning. This, in light of the results of our survey, in which 21/30 respondents claimed to be aware of the isolated usage of TLS enforcement, shows that sites are clearly not making use of CSP's full potential.

Despite the growth of CSP for framing control, unfortunately, it still lags behind the increasing adoption of XFO and more importantly, the complexity of CSP as such seems to confuse operators. This is evidenced by the fact that in December 2018 we observed 362 Web sites using CSP without `frame-ancestors` in combination with XFO. For these Web sites, it would be trivial to use CSP to enforce the same protection, but this is not done. Though this might be caused by outdated CSPs written before framing control support was added, we observed an interesting phenomenon: a third of the Web sites which deploy CSP without using `frame-ancestors` are making use of `upgrade-secure-requests`, which was introduced to CSP only later. This implies that CSP is often perceived as a complex mechanism to restrict content inclusion and not as a meaningful mechanism to control framing, which can even be used without any restriction on included content.

Overall, we find that CSP has grown from a mechanism aimed at restricting content to a multi-use measure to improve the security of Web applications. Our work has highlighted that this shows success with respect to TLS enforcement and framing control but also indicates that operators tend to shy away from deploying CSP, even though it could in many cases easily benefit their security. We believe this is caused by the ever-increasing complexity of the CSP mechanism. Apart from the already existing directives, new features for content restriction, such as more involved mechanisms for securing script code in attributes [97], are being added. In addition, with features such as `navigate-to` [96] and the signaling for Trusted Types [142], CSP is becoming a highly complex, generic Security Policy. This perceived complexity was also echoed in the notification responses, with operators explicitly naming complexity as the hurdle towards CSP deployment.

3.9.3 Quo Vadis, CSP?

Given our insights regarding CSP's (in)ability to restrict script content and the reasons we uncovered through our analysis, we propose three actionable steps which we believe can help CSP's adoption and the security of deployed policies.

3.9.3.1 `unsafe-nonced-elements`

One major roadblock to CSP adoption is the inability to use event handlers. While Chrome has recently added support for `unsafe-hashed-attributes` [95], this only enables operators to make their hash-based allow-list apply to event handlers. In practice though, we observed up to 3,344 different event handlers on a single page. While the median is only at around 5, any update to the event handlers (for all pages!) needs to be propagated to the CSP header. Instead, we propose that CSP be extended to allow for nonced elements, i.e., when an element carries a nonce, any event handlers are permitted on that element but not its children. This would remove the need of always resorting to `unsafe-inline`. This proposed changes comes with certain risks, namely nonce-reuse attacks and injections inside nonced elements. The first is acknowledged by the CSP standard authors, who proposed a fix as follows [104]: if a script tag is nonced, it will only be executed if within all of its attributes, no additional opening script tag can be found. This could be easily extended to check for other elements. To understand the feasibility of the approach, for all of the 317 sites with `unsafe-inline` which used event handlers (see Section 3.7.2) we checked each snapshot to gauge whether an element with an event handler also contained markup in any of its attributes. Assuming these were to be nonced, but contain markup, CSP would falsely block the nonced element from executing the event handlers. This analysis showed that not a single snapshot had such a case; meaning that our proposal would likely not cause incompatibilities. Second, an attacker could abuse an injection *inside* a nonced element to add additional event handlers. However, this is a significantly lower attack surface than using `unsafe-inline`, for which an injection anywhere in the page is sufficient to allow for an XSS attack. Hence, we argue that this is a viable option to make CSP more usable, while not fully sacrificing security.

3.9.3.2 Incorporate CSP into Development Cycle

As our analysis has highlighted, many operators attempted to deploy CSP to an existing application, only to either end up with a trivially bypassable policy or give up on CSP altogether. With a few exceptions, deploying CSP retroactively to an existing application does not appear to be a viable strategy. This is aggravated by the use of third parties, which are known to dynamically add additional content. Hence, we argue that CSP must be incorporated into the development cycle. In particular, we urge IDE vendors to add checks for CSP incompatible code at development time by, e.g., warning developers to not add inline scripts or event handlers, but instead proposing to externalize the desired functionality. In addition, as prior work has documented, third parties often add (script) content dynamically. We could confirm that by attributing over half the detected content violations to third parties. While prior work, such as Calzavara et al. [20] and Weichselbaum et al. [136], have proposed means to address this issue, we instead argue that third parties should be explicit about their dependencies and their impact on CSP (e.g., if they only add scripts dynamically, thereby enabling support for `strict-dynamic`). In this way, during development, a web developer could decide to incorporate another similar vendor which provides the same service with less CSP interference. This could also be incorporated into IDEs, which could

automatically analyze included parties and warn the developers about roadblocks for CSP.

3.9.3.3 Updated Informational Material for Developers

In light of our findings and survey responses, it appears that the complexity for script content restriction gives CSP a bad reputation. Given that this is *not* counteracted by widely used resources pointing out the easy-to-deploy use cases of TLS enforcement and framing control, we advocate for clear communication of the individual goals in such resources. Likewise, we argue that browser vendors are in a unique position to improve upon this situation, by warning developers through the console about inconsistently implemented mechanisms like `X-Frame-Options`, even providing a quick fix for the issue by deploying CSP. To that end, we have started discussions with both the Chrome and Firefox team on addressing this issue, with the hope of allowing more sites to leverage the easy-to-use capabilities that CSP can offer for better security.

3.10 Summary

In this chapter, we conducted a longitudinal analysis of the deployment and evolution of CSP from its beginning in early 2012 until the end of 2018. By leveraging the Internet Archive to collect the historical headers for 10,000 highly ranked websites for seven years, we identified that while CSP was initially meant as a mitigation for script injection, it has evolved into a mechanism that is equally often used to enforce TLS connections and control framing. Our longitudinal analysis allowed us to document the struggles developers face when constructing a secure and functional policy for content restriction, and highlighted that even secure CSPs are prone to bypasses through typos and expired domains. Combined with the lack of adoption of new features such as `strict-dynamic`, this leads us to conclude that the script-restricting parts of CSP are unlikely to succeed in the future. Moreover, while CSP is increasingly deployed for TLS enforcement and framing control, its adoption rate is still unsatisfactory. The insights gathered from our survey indicate that CSP has earned a bad reputation due to its complexity in content restriction, resulting in developers shying away from *any* part of CSP. Even though the alternative use cases for CSP are easy to deploy, this bad reputation, unless counteracted by tools, browser vendors, and informational material alike, significantly hampers CSP's ability to improve the Web's security.

This chapter revealed details about CSP's past and (at the time of conducting the study) presence. We evaluated real-world policy configurations and their use cases to confirm that CSPs out there are, and have always been, trivially bypassable (*RQ1*). Also, our data and the survey indicate that having two different mechanisms that aim to mitigate the same attack, is a recipe for inconsistent protections against clickjacking as developers are not aware of the issue. The following chapters evaluate if all users of a Web site get the same level of protection (*RQ2*) even in case of seemingly sane security policies, or if inconsistencies are impeding the level of security (Chapter 4). Also, we aim to find the root causes for the sheer omnipresent misconfiguration of CSPs (*RQ3*) by conducting a qualitative study involving real-world developers (Chapter 5).

4
Inconsistent Deployment of Security Headers In-The-Wild

Contributions of the Auhtors

This chapter contains the results of *The Security Lottery: Measuring Client-Side Web Security Inconsistencies* [P3] and *A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web* [P2].

With the exception of Section 4.5, this chapter explains *The Security Lottery* [P3]. For this paper, Sebastian Roth, Ben Stock, and Stefano Calzavara conceived and designed the study. Sebastian Roth and Ben Stock implemented the crawler to collect and analysed the data. Sebastian Roth sent out the disclosure notifications to the affected parties and open-sourced the crawling and analytics scripts [A4]. All authors commented on the text and contributed to individual sections.

For *Tale of Two Headers*, explained in Section 4.5, Stefano Calzavara conceived the study. Sebastian Roth, Ben Stock, and Stefano Calzavara designed the study. All authors implemented the crawling and analytics scripts. Sebastian Roth implemented the server-side proxy. Ben Stock, Michael Backes, and Stefano Calzavara supervised the study. All authors commented on the text and contributed to individual sections.

4.1 Motivation

The results presented in Chapter 3 indicate that the presence of two different mechanisms to mitigate the same attack has the potential to cause inconsistencies. Those inconsistencies between XFO and CSP's frame-ancestors are (only) the tip of the iceberg, which is described in detail in Section 4.5. Unfortunately, the attack surface against Web applications is very large and Web application security is a complicated topic that requires actions against several different types of attackers, not only against framing-based attacks.

The adoption of appropriate browser-side defenses to prevent or mitigate relevant Web threats [124] has become a frequently used layer of defense. Examples of such defenses which are quite popular on high-profile Web sites include cookie security attributes, Content Security Policy (CSP), and HTTP Strict Transport Security (HSTS). Several papers already studied (and criticized) the adoption of different client-side security mechanisms in the wild. For example, cookie security attributes turned out to be underused by site operators [130, 18], CSP is most often configured incorrectly [136, P1] and HSTS had a hard time getting traction even on top sites [79, 115]. However, these studies were performed in a fixed and often unspecified measurement setting, e.g., measuring security using a single crawler running a specific browser on a machine with a static IP.

In this chapter, we investigate the client-side security of top sites from a new angle. Our analysis starts from the observation that client-side security inherently depends on the correct communication of security policies from the server to the client. This seemingly straightforward process might hide subtleties, which can affect Web application security and its large-scale measurement [62]. In particular, we observe that there is no guarantee that two clients accessing the same URL receive the same security policies. For example, clients accessing the same Web application from different geolocations might be served by different servers, due to the existence of several localized variants

of the same site. Moreover, the same person might access the same Web application through different User-Agents, e.g., Chrome on their desktop computer and Safari on their iOS mobile device, while the same client might access the same Web application using different forms of network access, e.g., through a VPN. Finally, even the same HTTP request might receive different HTTP responses when it is sent multiple times by the same client apparently under the very same conditions, due to the DNS system resolving the same hostname to different IP addresses and the possible intervention of load balancers and HTTP middleboxes [54].

Intuitively, we (and Web users) would like all the responses served across these multiple legitimate scenarios to enforce the same security policies, otherwise, a *security lottery* would take place: depending on specific client characteristics, different levels of Web application security would be provided to users. We refer to this class of potential security flaws as *inconsistencies*. Unfortunately, our research shows that multiple client characteristics might inadvertently affect Web application security in the wild, thus leaving users of prominent services unprotected against both Web and network attacks.

In this chapter, we measure the prevalence of inconsistencies in the security policies of top sites across different client characteristics and we quantify their security implications:

1. We propose a data collection methodology tailored to our analysis and we build a dataset of 13,626,145 responses collected from the 10,000 highest-ranking sites available through HTTPS (based on Tranco [102]), while testing a number of different client characteristics. Our tests include the use of different User-Agents, network access methods, and language settings.

2. We introduce general definitions of consistency for client-side security mechanisms and we instantiate them to a set of popular defenses available in modern browsers. Our definitions are *semantics-based*, i.e., they only capture inconsistencies with a potential security import, rather than superficial syntactic differences or other types of false positives coming from the different enforcement models of different security mechanisms.

3. We apply our definitions to the collected data and we report on the key findings. Our measurement shows that a significant fraction of the analyzed Web sites suffer from different types of client-side security inconsistencies. Remarkably, the majority of them can be attributed to specific client characteristics, which identify weak spots in the security configuration, while the others can be attributed to non-deterministic factors, which may nevertheless be exploited by attackers.

4. We present a comprehensive analysis of inconsistencies in framing control policies. We proposed different recommendations against those for Web developers and browser vendors, as well as a server-side proxy designed to retrofit security to existing Web applications.

Factor	Set of tests	Tests
User-Agent	Windows client Linux client macOS client Android client iOS client	UA header: Chrome 96, Firefox 95, Edge 96, Opera 82 UA header: Chrome 96, Firefox 95, Opera 82 UA header: Chrome 96, Firefox 95, Edge 96, Opera 82, Safari 15.2 UA header: Chrome 96, Firefox 95, Opera 96 UA header: Chrome 96, Firefox 95, Edge 86, Safari 15.2
Vantage Point	VPN service Onion network	Servers from `hidemyass.com` - 1 per country (218 countries) Standard Onion client - 1 end-node per country (49 countries)
Client Configuration	Language	Accept-Language header: en, es, cn, ru, de

Table 4.1: Selected client conditions that might influence the received security headers

4.2 Data Collection Framework

Here, we discuss which types of factors we investigate to understand differences in client-side Web security guarantees. We then outline which information we use to collect data for each of the factors and we discuss our key design choices.

4.2.1 Scope of the Study

We assume the use of a modern client implementing all the security mechanisms in Section 2.3 (except Trusted Types), e.g., the latest versions of Chrome and Firefox as of January 1, 2022. We thus exclude the use of legacy clients, because it is clear that this discouraged practice may severely downgrade Web application security, e.g., because CSP is not supported by the browser. We also assume that modern clients correctly implement all the security mechanisms according to their official specifications, i.e., if two different clients receive the same security headers, we assume they enforce the same intended level of protection. Finding bugs in the implementation of client-side security mechanisms is an orthogonal issue [114].

We identify three *factors* which users may legitimately manipulate as part of their everyday Web browsing experience, without realizing that they can unintendedly affect Web application security:

1. User-Agent: users have different tastes and might prefer different browsers, e.g., due to the privacy policies they implement by default. Moreover, the same user might use different browsers on different devices, possibly running different operating systems. As long as users make use of a modern, up-to-date browser, they likely do not expect Web application security to be affected, yet it is possible that a site sets up different configurations based on the value of the User-Agent header of the incoming requests for generic reasons. This practice, known as *User-Agent sniffing*, might leave some User-Agents unprotected against specific classes of attacks. Note that we use the terms *browser* and *User-Agent* interchangeably.
2. Vantage Point: users may access a given site from different geographical vantage points. Users may not expect this practice to affect Web application security, yet it is possible that the geolocation has an impact on dynamically loaded advertisement, which in turn might require a different server-side configuration of CSP. Further, the exit nodes of the Onion network are publicly known, hence a con-

nection through the Onion network might result in a different response, possibly introducing a difference in security.
3. Client Configuration: some configuration settings might influence the way clients interact with Web sites. For example, the language of the client is normally advertised in the Accept-Language header and the site might use this information to redirect the client to a localized homepage served by a different host, possibly with a different security configuration. We only focus on this aspect (language) in our analysis for simplicity.

Table 4.1 identifies for each factor a set of possible *tests*, which can be easily simulated in a black-box fashion by a Web crawler. We identify the respective User-Agent strings for the browsers in the table via a public online repository [140].

4.2.2 Challenges and Design Choices

The discussion in the previous section does not directly yield a dataset construction procedure, due to a couple of problems we have to deal with. The first is related to the sheer *number of requests* to send to each Web site, because it is possible that security policies are influenced by a combination of multiple factors. To mitigate this, we only cover a subset of all the possible combinations by testing different factors in isolation:

1. User-Agent: when testing different User-Agents, we access the network through a local machine with a German IP, and we do not set the Accept-Language header.
2. Vantage Point: when testing different vantage points, we set the User-Agent header to Chrome 96 for Windows and we do not set the Accept-Language header. We use hidemyass VPN[1] to access the sites from 218 countries and additional 49 different countries through for the Onion network.
3. Client Configuration: when testing different language settings, we access the network through a local machine with a German IP address and we set the User-Agent header to Chrome 96 for Windows.

This way, we can measure meaningful *intra-factor* variations in the level of Web application security, e.g., by estimating the impact of the choice of a specific User-Agent when the other two factors are fixed.

The second problem is related to *non-determinism*, because the same request does not necessarily always receive the same response. For example, DNS might resolve the same hostname to different IP addresses at different times and load balancers can forward requests for the same resource to different backend servers to improve performance. In either case, there is no guarantee that all the hosts which might process the request enforce the same security policies. Security inconsistencies introduced by non-deterministic factors are in the scope of our study, yet they complicate the *attribution* of security flaws. To exemplify the problem, assume that Chrome appears to be less protected than Firefox because it did not receive any security headers at all. In this case, the User-Agent itself may not be the actual cause of insecurity because non-determinism may have played a role on the received response, e.g., the DNS resolution accidentally redirected Chrome to a poorly configured host. To mitigate this problem, each Web site is visited multiple times (five in our collection) for each test and

[1] `https://www.hidemyass.com/`

all the corresponding responses are stored, which allows us to detect non-deterministic security inconsistencies.

Our crawler takes as input a set of *Factors*, a set of *Tests* associated to each factor, a set of *URLs* to access and a number of visits n to perform for each test. For each factor $f \in Factors$ and each associated test $t \in Tests[f]$, each $u \in URLs$ is visited n times setting f to t. For the finally reached URL (after potential redirects), we resolve its origin o and save the response r in the dataset at the entry $D[u,t]$, enriched with the origin o. We refer to o as the *end origin* of the response r.

4.3 Formalizing Inconsistencies

Before presenting the formal details, we present an overview of our analysis methodology to explain its design and subtleties. A simple notion of consistent security might be as follows: all the responses collected from the same URL must enforce the same security policies. However, this intuitive definition of consistency is too strong to be useful in practice.

The first point we make is that requiring the *same* security policies for all the collected responses is overly restrictive. As a matter of fact, two security policies can be *syntactically* different, yet provide an equivalent level of protection. For example, two syntactically different CSPs may both effectively mitigate the dangers of XSS. As another example, a host may configure HSTS with tiny fluctuations in the value of the max-age attribute which do not play any role in terms of practical security. To abstract from syntactic differences without significant security implications, we define *equivalence* relations $\sim_m$ for each security mechanism m under study.

A second challenge of our analysis is related to *legitimately* different policies we might get for different client characteristics: for example, a Web site which activates CSP for desktop clients might redirect mobile clients to a static error page which requires no protection and thus enforces no CSP. We do not want to consider these cases as security inconsistencies, because the Web pages are different and legitimately require a different level of protection. To filter out false positives, we define *compatibility* relations $\bowtie_m$ for each security mechanism m under study: incompatible responses cannot lead to security inconsistencies, because m protects different objects. For the sake of generality, our framework supports different compatibility relations for different security mechanisms, because they may be based on different enforcement models, e.g., CSP operates at the page level, while HSTS operates at the host level. In our Web measurement, however, we use the same compatibility definition $\bowtie_m$ for each security mechanism m, because we want to be conservative in our findings and avoid over-reporting (see Section 4.3.2).

Finally, we observe that not all inconsistencies are equal in terms of real-world *exploitation*. Inconsistencies enabled by non-deterministic factors can be exploited by attackers who are determined (or lucky) enough to eventually stumble into them, while inconsistencies enabled by deterministic factors like the adoption of a specific User-Agent identify weak spots that knowledgeable attackers can more easily take advantage of. We discriminate these two cases by having two different definitions of consistency, as detailed below.

4.3.1 Consistency

For any security mechanism m, we assume a reflexive and transitive relation $r \lesssim_m r'$ reading as: response r configures m no more securely than response r'. We write $r \sim_m r'$ if and only if r configures m equivalently to r', i.e., we have that both $r \lesssim_m r'$ and $r' \lesssim_m r$ hold. Finally, we assume reflexive and symmetric relations $r \bowtie_m r'$ reading as: response r and r' are compatible with respect to the security mechanism m. We later instantiate $\lesssim_m$ and $\bowtie_m$ to the different security mechanisms considered in our study to capture specific security properties.

The first definition we introduce is called *intra-test consistency*. It requires all compatible responses collected within the same test to provide an equivalent level of protection. Violations to this consistency property are likely attributed to non-deterministic factors, because all the observable client conditions are the same across the received responses.

Definition 1 (Intra-Test Consistency)**.** The page with URL u satisfies *intra-test consistency* for the security mechanism m within the test t if and only if for all responses $r \in D[u, t]$ and $r' \in D[u, t]$ such that $r \bowtie_m r'$ we have $r \sim_m r'$.

The second definition of consistency which we introduce is called *inter-test consistency*. It requires all compatible responses collected within two different tests (defined for the same factor) to provide an equivalent level of protection. We require the two tests to satisfy intra-test consistency to rule out inconsistencies enabled by non-deterministic factors, e.g., occasionally missing headers on responses collected within the same test. This way, inter-test inconsistencies can be realistically attributed to specific client characteristics.

Definition 2 (Inter-Test Consistency)**.** The page with URL u satisfies *inter-test consistency* for the security mechanism m across the tests t, t', defined for the same factor and satisfying intra-test consistency, if and only if for all responses $r \in D[u, t]$ and $r' \in D[u, t']$ such that $r \bowtie_m r'$ we have $r \sim_m r'$.

We exemplify the definitions at work on a few toy examples. Let us focus on just two tests for the User-Agent factor for simplicity: Chrome 96 for Windows and Firefox 95 for Linux. Assume that pages are visited five times for each test and may be classified in two security levels: low (L) and high (H) with $L \lesssim_m H$. Consider now the example observations in Table 4.2, that we assume to be all pairwise compatible. The first row models a straightforward scenario where both intra-test consistency and inter-test consistency are satisfied. The second row models a scenario where intra-test consistency does not hold, due to the L observation for Chrome, hence inter-test consistency is undefined: this case captures a non-deterministic security downgrade. The third row represents a scenario where intra-test consistency is satisfied, but inter-test consistency is not: this captures a deterministic security downgrade occurring when the page is visited using Firefox.

4.3.2 Compatibility Relations

Arguably, certain security mechanisms such as CSP are not applicable to an origin per se, but rather to the *content* provided under a given URL. However, not every URL

Chrome 96	Firefox 95
H, H, H, H, H	H, H, H, H, H
H, H, H, L, H	H, H, H, H, H
H, H, H, H, H	L, L, L, L, L

Table 4.2: Example observations upon crawling

returns the same content on each load, in particular in the presence of errors or block pages. For such pages, which might originate from CDNs like Cloudflare, enforcing the CSP of the original page might not make sense. Hence, when we encounter an inconsistency, we need to ensure that this inconsistency is not due to different content being delivered. We leverage a *similarity score* on Web pages for this task.

Page Similarity Based on preliminary analyses of the collected data, our page similarity score takes into account four factors: first, we rely on JavaScript as a proxy to implement the pages' functionality. Therefore we created sets of the hosts from which scripts are loaded and computed their Jaccard similarity. By manually investigating the script data that we got from analyzing our responses, we encountered cases where the Jaccard similarity of the script hosts was 1, e.g., because the page only used inline scripts. Notably, the number of inline scripts differed significantly; hence, we also consider the number of scripts for each host as a second factor for our similarity. However, these first two factors do not work well for pages that only rely on a few scripts or do not even use JavaScript on their main page. To lower the impact of this, we manually investigated those pages, and we observed that the title of the page often changes in case of errors, e.g., showing just `domain.com`. We, therefore, compute the longest common substring between the titles of two documents and compute the ratio of this overlap as our third factor. In addition to that, we observed that the response size also differed between error/block pages and pages with content. Thus we define the content size of the response as our fourth factor by assigning a value between 0 and 1 (indicating the relative sizes). We finally combine our factors by computing their average. The resulting value (between 0 and 1) is then used to determine the similarity between two pages. We consider two pages as similar if their similarity score is at least 0.8.

To find the page similarity threshold, we computed the similarity score of pages where we have seen *syntactically* different security headers after normalization (e.g., normalizing CSP nonces or report URLs). Specifically, for each such case, we took the largest response as the baseline (under the assumption that content pages are larger than error pages). Then, we computed the similarity to this baseline for each of the other responses. Figure 4.1 shows the result of this, both as a histogram (bucket size 0.05), as well as the CDF for the entirety of comparisons. We find that the peak of the histogram is in the right-most bucket, i.e., similarity above 0.95. In the CDF, we observe that the similarity for most of the cases in that bucket is even beyond 0.98. Moreover, the shallow slope of the CDF around 0.8 leads to taking this value as a candidate for a threshold to distinguish between content and error/block pages. Notably, error or block pages are just one instance of a potential difference. Our notion generally discerns dissimilar pages, which could require differing levels of protection, so

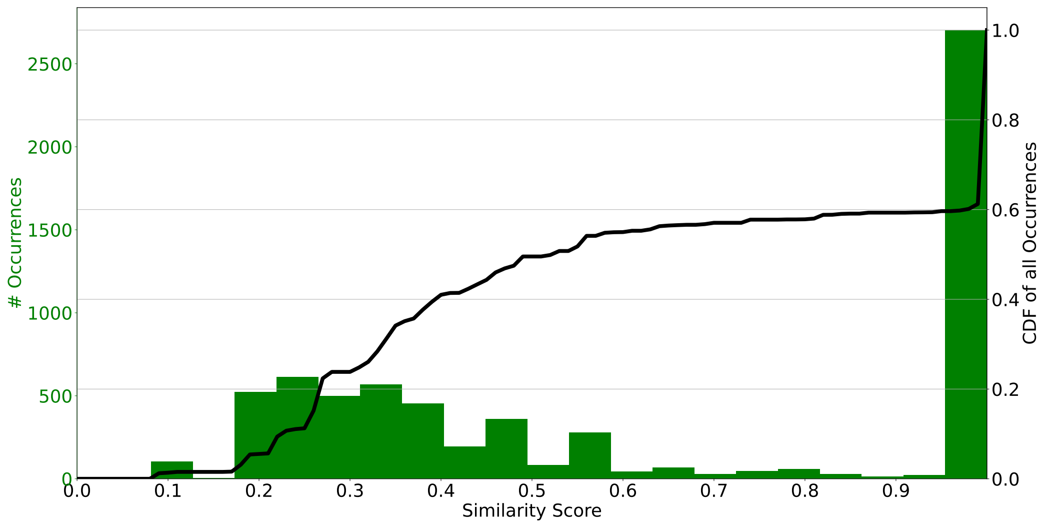

Figure 4.1: Histogram and CDF of the similarity values for syntactically different header values.

as to avoid reasoning about security inconsistencies when these are in fact legitimate.

To assess the effectiveness of the threshold, we performed analyses to identify false negatives (similar pages marked as dissimilar) and false positives (dissimilar pages marked as similar). To confirm that pages below our threshold are indeed no content pages, we spawned Chromium instances to render those pages *below* the threshold and take screenshots (after 5s) for further analysis. We then manually looked at those 1,939 cases: for the very vast majority, the pages were clearly error pages or block pages showing information about robot detection / CAPTCHAs. Among the edge cases that were close to the threshold, we found one site where the page skeleton looked like the actual content pages for the domain, but without any content. We confirmed this case as a true negative, because the page was under the threshold and seemed different in terms of the content.

Another case that was close to the threshold was a domain that randomly showed a slightly different page that additionally included items in sale. Therefore their title changed from *Mercado Libre Argentina* to *Hot Sale 2021*, and due to the additional content, the file size increased, bringing the similarity down to 0.78. Nevertheless, both pages belonged to the same application and were not empty, error, or block pages, and hence we consider this case as a false negative. Notably, we only faced this one false negative in our dataset.

In order to also assess the number of false positives, we investigated the similarity of pages above the threshold and below 0.95. For the remaining cases beyond 0.95, we are confident to have no error pages in there, given the significant overlap through all four metrics. By looking at those positive (similar) pages, we have seen one single false positive. This case had a similarity score of 0.82, although one HTML file was clearly an error page. This happened due to the error page also including the scripts from the content page. In addition, the pages were similar in size, yet different in the title

(*Hosting Platform of Choice* vs *404 Error / cPanel*). Because this is clearly an error page, we manually removed this error case from our results. Except for this one case, we could not spot any false positives in the pages above 0.8.

Our experiments show that the chosen threshold is appropriate to reason about inconsistencies, because it might suffer from occasional false negatives, but produced just one false positive and one false negative in our experiments (see Table 4.3). This means that we might lose some inconsistencies, but we are confident not to incorrectly report on inconsistencies where there are, in fact, none (because the content to be protected is different). We now use this *page similarity* notion, to define the following compatibility relations:

Compatibility Given two responses r and r', we let $r \bowtie_m r'$ if and only if the end origins of r and r' are the same and, additionally, the page similarity between r and r' reaches the stipulated threshold. Note that we use the same compatibility relation for each security mechanism m. This may be overly conservative for host-based security mechanisms like HSTS, because different pages under the same host may enforce different HSTS policies for the same object (host), hence one may legitimately disregard page similarity in the compatibility relation for HSTS. However, we empirically noticed that this weaker compatibility notion leads to over-reporting inconsistencies for HSTS. In particular, for sites hosted by CDNs, depending on our vantage point or frequency of requests, we received block or CAPTCHA pages. For Cloudflare, these lacked HSTS. However, it can be argued that this has no significant security implications. In particular, if the site normally uses HSTS, the browser will likely be aware that communication should be performed over HTTPS and the lack of the HSTS header does not deactivate HSTS. For this reason, we prefer to be conservative in our analysis and reuse the same compatibility relation (with page similarity) for all the security mechanisms to avoid potential over-reporting.

4.3.3 Equivalence Relations

We now define the $\lesssim_m$ relations ("no more secure than") for the different security mechanisms considered in this chapter, leading to corresponding security equivalence relations $\sim_m$. These definitions are motivated by the semantics of the security mechanisms under study.

Cookie Security Attributes Defining inconsistencies for cookie security attributes is straightforward, because the HttpOnly and Secure attributes require no configuration, while the SameSite attribute has three different configurations with increasing level of protection: None, Lax, Strict.

False Negatives	False Positives
1/1,939 (0.05%)	1/93 (1.08%)

Table 4.3: False positive & false negative rates for similarity

We identify cookies with the triple including their name, Domain and Path, as mandated by the corresponding RFC [6]. Formally, we let $r \lesssim_{ck} r'$ if and only if all cookies c occurring in both r and r' satisfy the following conditions:

1. If c is marked as HttpOnly in r, then c is marked as HttpOnly also in r'.
2. If c is marked as Secure in r, then c is marked as Secure also in r'.
3. If c is marked as SameSite in r, then c is marked as SameSite also in r' with at least the same level of protection, e.g., if the SameSite attribute of c is set to Lax in r, then it must be set to Lax or Strict in r'.

Content Security Policy

Defining inconsistencies for CSP is more complicated, since it is an expressive security mechanism, which supports many use cases and can thus be analyzed from multiple angles. To address this, we build multiple equivalence relations for CSP to cover different use cases [P1].

A first use case for CSP is XSS mitigation, which we study by leveraging a definition of *safe CSP* for CSP Level 3 [22]. This definition ensures that the CSP puts some meaningful restrictions against XSS: policies which do not comply with the definition can be trivially bypassed by an attacker upon any content injection.

Definition 3 (Safe CSP [22]). A CSP is *safe* if and only if it contains a script-src directive (or a default-src directive in its absence) bound to a value v satisfying both the following conditions:

1. v does not contain the 'unsafe-inline' source-expression, unless nonces or hashes are also present in v.
2. v does not contain the wildcard * or any full scheme from the following: http:, https:, data:, unless 'strict-dynamic' is also present in v.

We let $r \lesssim_{csp-xss} r'$ if and only if, whenever r sets a safe CSP, then also r' sets a safe CSP.

The second use case for CSP is framing control. To define an equivalence relation for this use case, we divide responses in four classes based on the enforced framing restrictions:

1. Framing is allowed on all origins.
2. Cross-origin framing is allowed only on selected origins.
3. Only same-origin framing is allowed.
4. Framing is not allowed on any origin.

We then let $r \lesssim_{csp-frm} r'$ if and only if the class of r is less than or equal to the class of r'.

The last use case for CSP is TLS enforcement. Its equivalence relation is defined by having $r \lesssim_{csp-tls} r'$ if and only if, whenever r activates upgrade-insecure-requests or block-all-mixed-content, then also r' does it. In other words, when r forbids the use of HTTP, then also r' enforces the same security restriction.

X-Frame-Options

We just focus on SAMEORIGIN and DENY as possible values of XFO, since ALLOW-FROM is unsupported by the modern clients considered in the present study. This implies that responses can be categorized in just three different classes:

1. Framing is allowed on all origins.
2. Only same-origin framing is allowed.
3. Framing is not allowed on any origin.

We then let $r \lesssim_{xfo} r'$ if and only if the class of r is less than or equal to the class of r'.

Strict Transport Security Defining inconsistencies for HSTS requires some care, due to possible differences in the max-age attribute which arguably have little to no impact in terms of real-world security. Our choice is discriminating four classes of responses, as follows:

1. Responses with max-age set to 0, thus forcing HSTS deactivation for their host.
2. Responses without any HSTS header. These responses do not activate HSTS, but do not forcibly deactivate it.
3. Responses with max-age enforcing protection for less than one year. This practice can be useful, but does not comply with the minimal required duration for inclusion in the HSTS preload list.
4. Responses with max-age enforcing protection for at least one year, qualifying the host for preload list inclusion.

We then let $r \lesssim_{hsts} r'$ iff all the following conditions hold:

1. The class of r is less than or equal to the class of r'.
2. If r sets the includeSubDomains directive, so does r'.
3. If r sets the preload directive, then also r' sets it.

Handling Multiple Headers Careful readers may have noticed that the above definitions assume responses to contain at most one header of each type, yet real-world responses might violate this assumption because headers can be set multiple times. Our dataset still contains at most a single header of each type, because the Requests library used in our data collection folds multiple headers into a single header set to a comma-separated concatenation of their values. For handling of multiple headers, we follow specifications where possible:

- If the same cookie (identified by name, Domain and Path) is set in multiple headers, the last one should be prioritized [6]. We thus normalize the collected headers to reflect this behavior within a single header.
- If a response contains multiple CSP headers, all of them should be enforced at the same time [139]. We thus normalize the collected headers by replacing them with a single header enforcing the conjunction of all CSPs.
- If a response contains multiple XFO headers, the correct browser behavior is undefined in the specification. Since prior research showed that different clients handle multiple XFO headers quite differently [P2], we check for syntactic differences if multiple values are present.
- If a response contains multiple HSTS headers, the first one should be prioritized [53]. We thus clean our data to keep just the first HSTS header.

4.4 Measuring Inconsistencies

We use the data collection framework in Section 4.2 to collect data from live Web sites and apply the formalization in Section 4.3 to measure inconsistencies. The focus of our study is to understand inconsistencies caused by the servers of highly ranked sites. To ensure that any data we collect could not be tainted through network proxies or firewalls, we decided to only include sites which were served through HTTPS. Specifically, we visited each of the sites in the Tranco list [102] through `https://site.com` and `https://www.site.com`, disregarding those which were not accessible through HTTPS. Further, for each final URL, we determined if this was still under the same eTLD+1 as the originally visited one and not a localized version, e.g., `https://site.eu`, so as to avoid selecting a site which is actually not highly ranked. As a result, this process yields the list of the 10,000 highest-ranked sites available over HTTPS.

Based on this methodology, we arrived at the set of top 10,000 HTTPS sites based on the Tranco list of January 1, 2022[2]. We ran our first crawl, on which we report in the following, from January 2 through January 4, 2022. To ensure that our measurement was not merely a single measurement which is not repeatable, we ran three more confirmation crawls (January 6, 10, and 14, 2022). For each crawl, we collected between 13,626,145 and 13,742,760 responses. While we focus on the results of the first crawl, the appendix lists the overlap in findings between the first and the respective follow-up crawls (Section B.2), which highlights that our results can be confirmed over multiple crawls within 12 days. To ease the confirmation and reproducibility of our findings we made our crawling and analytics pipeline publicly available [A4].

In the following, we outline the key results supported by the analysis of the collected data. We first present a high-level overview of the findings and then discuss security inconsistencies introduced by different factors, as well as the security implications of the inconsistencies.

4.4.1 Overview of the Findings

Usage Statistics To give an overview of the deployed security mechanisms in the wild, we computed the number of sites which activated a specific security mechanism at least once across our data collection. Table 4.4 shows the resulting usage statistics for each of the selected security mechanisms in the second column. Note that this is an aggregate over all different tests, i.e., it combines checks for different User-Agents, vantage points and languages. In total, 8,174 sites made use of any of the security mechanisms. The most widely used mechanism was X-Frame-Options with 5,692 occurrences. HSTS was used on 4,562 sites, whereas at least one cookie was configured with any of the security attributes on 3,876 sites. The vast majority of these cases stem from the usage of HttpOnly or Secure attributes, with only 788 sites making use of SameSite cookies. The least widely used header was Content-Security-Policy with 1,998 sites which deployed it. Notably, the vast majority of sites used CSP for framing control rather than for its original purpose of XSS mitigation [121]. It is worth noting that for XSS mitigation, we

[2] Available at `https://tranco-list.eu/list/XVWN`

Mechanism	Usage	# Sites w/ intra-test inconsistencies					# Sites w/ inter-test inconsistencies					# Sites w/ only inter-test inconsistencies				
		UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**
Content Security Policy	1,998	12	11	31	23	36	15	-	29	18	47	15	-	11	3	28
- *for XSS mitigation*	360	1	-	1	1	3	9	-	1	1	10	9	-	1	-	10
- *for framing control*	1,288	6	5	15	9	16	2	-	16	5	20	2	-	9	1	12
- *for TLS enforcement*	661	7	7	19	14	22	4	-	12	12	17	4	-	1	2	6
X-Frame-Options	5,692	20	18	43	22	50	7	-	29	13	37	7	-	9	5	20
Strict-Transport-Security	4,562	15	13	28	23	38	8	-	23	16	35	8	-	12	5	22
- *w/o page similarity*	-	*42*	*33*	*148*	*593*	*693*	*19*	*2*	*576*	*218*	*643*	*17*	*2*	*524*	*20*	*552*
- *preload*	920	3	3	6	6	10	-	-	9	4	10	-	-	6	-	6
↳ *w/o page similarity*	-	*5*	*6*	*20*	*113*	*124*	*1*	*1*	*124*	*48*	*137*	*1*	*1*	*117*	*2*	*119*
Cookie Security	3,876	10	9	11	12	16	150	1	13	8	167	149	1	9	2	160
- *Secure attribute*	2,937	4	4	5	6	8	144	-	8	3	152	144	-	7	1	151
- *SameSite attribute*	788	5	5	5	6	7	6	1	4	4	14	6	1	2	-	9
- *HttpOnly attribute*	3,104	1	-	2	2	3	2	-	3	2	6	2	-	2	1	5
Any	8,174	51	44	103	75	127	177	1	82	49	267	174	1	34	12	194
Any (incl. HSTS w/o similarity)	*8,174*	*77*	*64*	*222*	*634*	*765*	*188*	*3*	*631*	*252*	*833*	*183*	*3*	*541*	*26*	*429*

Table 4.4: Detected intra-test and inter-test inconsistencies by factor (321 sites in total). We present the numbers with and without page similarity for HSTS to highlight the impact of this choice on the measurement.

only count those cases which have a policy that is not trivial to bypass (Definition 3). Since our definition of inconsistency revolves around such policies, any site that did not have any meaningful XSS mitigation is not counted. Note that the number of sites for the subclasses of CSP and cookie security do not add to the overall usage, since a site may, e.g., configure a CSP that both mitigates XSS and enforces TLS.

Detected Inconsistencies In total we detected some inconsistency in 321 sites. Table 4.4 further shows three groups of columns: intra-test inconsistent, inter-test inconsistent, and *only* inter-test inconsistent sites. For the final column group, we removed all those sites for which we found an intra-test inconsistency for the given mechanism. This is to ensure that a site exhibiting a non-deterministic behavior is not accidentally flagged as suffering from inter-test inconsistencies, as required by our formal definition. Hence, the last column is a confident lower bound for the number of sites affected by inter-test inconsistencies. Overall, our crawl detected 127 sites which have some type of intra-test inconsistency and from 194 to 267 sites with inter-test inconsistencies. Notably, our confirmation crawls exhibited two interesting phenomena, i.e., the instability of intra-test inconsistencies and the stability of inter-test inconsistencies. Considering the union of all sites that suffered from intra-test inconsistencies at least once in our crawls, we found a total of 210 sites (Appendix Table 7.1). This likely means that the actual dangers of non-deterministic intra-test inconsistencies is more severe than what we could measure through our five observations. Conversely, the confirmation crawls showed that the number of sites with inter-test inconsistencies is stable over time (Appendix Table 7.2).

4.4.2 Intra-Test Inconsistencies

Intra-test inconsistencies come with particular security risks, as an attacker can abuse these to attack users opportunistically. In the following, we present case studies of intra-test inconsistencies for every mechanism and explain the corresponding security implications.

Cookie Security If a cookie may non-deterministically lack the HttpOnly attribute, an attacker could steal the cookie via malicious JavaScript by performing an XSS attack multiple times until access to the cookie succeeds. One of the three sites where we encountered this kind of inconsistency sets its authentication cookie named *auth-cookie_loggedIn* sometimes with and sometimes without the HttpOnly attribute.

A non-deterministically missing Secure attribute, as it happened on eight sites, allows a network attacker to steal the corresponding cookie. One site for example non-deterministically set their *csrfToken* cookie as Secure or not. Thus, attackers can steal this cookie and perform CSRF attacks because they know the anti-CSRF token. A similar issue happens on another site, for which the Secure attribute is inconsistently set on the session identifier *JSESSIONID*, thus potentially leading to session hijacking.

In seven sites, we found intra-test inconsistent deployments of the SameSite attribute, which is sometimes set to Lax and sometimes missing. This behavior might not be a problem for modern browsers, because Chromium-based browsers default to

Lax in case of a missing SameSite attribute. However, all Safari-based browsers still face the problem that cross-site attacks such as CSRF are possible due to this misconfiguration. One site, for example, sometimes set their *ASP.NET_SessionId* cookie with SameSite attribute set to Lax, and sometimes the SameSite attribute was not set, which enables an attacker to perform attacks such as CSRF.

Content Security Policy Overall, we found three sites for which XSS mitigation was enforced non-deterministically. For example, the responses from one site sometimes did not have any CSP for clients from Germany or Australia. Thus, an attacker can succeed by performing the attack multiple times until one of the responses does not carry a CSP.

For framing control, we found a total of 16 inconsistent sites across our tests. Note that the majority of inconsistencies were detected in the VPN crawl. This is because, in the VPN crawl, we test from 218 vantage point (compared to 49 tests for Onion, and 20 and five respectively, for User-Agent and language), which increases the chances of eventually getting an inconsistent response. For example, a site non-deterministically deployed frame-ancestors or not, hence an attacker can perform the attack multiple times (or load the target in multiple iframes) until the attack succeeds.

Finally, TLS was inconsistently enforced on 22 sites. For example, one site in our dataset deploys a CSP that aims to enforce TLS. However, irrespective of the factors that we checked, this CSP is not present in some responses. Nowadays, Chromium-based browsers auto-upgrade mixed content [27], whereas Firefox and Safari merely block it. Therefore, the security implications of missing TLS enforcement is limited. However, inconsistencies in this feature can lead to functionality issues. In 2020, Roth et al. [P1] showed that 77/251 sites which use CSP for TLS enforcement have HTTP resources linked from their front page. Thus, these inconsistencies might lead to essential resources being blocked in Firefox and Safari.

X-Frame-Options The most common intra-test inconsistency for X-Frame-Options was alternating between a deployed header and not deploying XFO at all (41 sites). This behavior enables an attacker to attempt the attack multiple times (or load the target in multiple iframes) until it succeeds. For the other nine cases, the Web applications alternate between a valid XFO header and a malformed one (e.g. sometimes prepending a `:` to its XFO header) or one using an unsupported feature such as ALLOW-FROM. As with omitting the header, an attacker can opportunistically exploit this.

Strict Transport Security Of the 38 sites with intra-test inconsistencies on HSTS, only six are present in the preload list, for which the issue has no implication on the client's security. For 23 sites the inconsistency is related to headers which are sometimes entirely omitted. For those not preloaded, this is problematic since the non-deterministic absence of the header might prolong the time frame where an attack is possible due to the trust-on-first-use problem of HSTS.

While inconsistencies for preloaded sites have no direct impact on a client, they nevertheless pose a threat. In our dataset seven hosts deploy an HSTS header sometimes with, sometimes without the preload directive. Here an attacker can remove the affected

site from the HSTS preload list by asking for removal of the site [30]. If the HSTS preload crawler hits a case without preload being present in the HSTS header, the site will be removed without the operator even noticing it. According to our tests with an author-owned preloaded site, there seems to be no rate-limiting in place to stop such abuse.

An intra-test inconsistent deployment of the HSTS includeSubdomains directive can also lead to problems, as it happened for four sites. For example, a payment service provider showed this behavior on several islands (e.g., Falkland Islands, Antigua and Barbuda, Bahamas, and Bermuda). Here an attacker could, in case of a lacking includeSubDomains directive, abuse the subdomains to attack the main domain (e.g., using subdomains to inject cookies into the top-level domain).

The most critical inconsistency in terms of exploitability is a host that randomly alternates between enabled and disabled HSTS, or has multiple enabled and disabled HSTS headers in random order (only the first entry is processed). Three sites have this kind of intra-test inconsistency. They, for example, alternate between `max-age=0, max-age=15768001` and `max-age=15768001`. Since the HSTS specification mandates adhering to the first observed HSTS header, the first case always deactivates HSTS. Similarly, one site non-deterministically disabled HSTS in certain (mostly eastern Europe) countries such as Lithuania, Moldova, Romania, and Ukraine. In both cases, an attacker could perform the attack several times until it succeeds because HSTS has been disabled in the last response. We also identified one site that alternate between a short max-age (<= 5 min) and a properly configured header. In those cases, the site allows an attacker to abuse the HSTS TOFU problem at a higher frequency because if the last HSTS header received was a short one, the next visit after a short time period (e.g., 5 min) will be vulnerable again, because the browser no longer enforces TLS.

Reasons for Intra-Test Inconsistencies In order to find the cause behind the intra-test inconsistencies, we took a closer look into the gathered data from the responses, such as peer IP addresses or cache headers. Here we noticed that indeed some of the different response headers were caused by caching, because all misconfigured header values also had a different `cache-control` header (18 sites) or a different `x-cache` header (five sites). For two sites, our data indicate that depending on the geolocation, we were redirected to a different end URL (under the same origin), which then causes the inconsistency. In the case of five sites, we were also able to attribute the inconsistency to certain peer IP addresses, which indicates that misconfigured origin servers might be the underlying problem. This hypothesis is also supported by one answer from the notification campaign, which indicates that “one of the origin servers seems to be configured differently”. However, here we observed that this inconsistency does not depend on the peer IP address, which indicated that what we see as the peer IP might only be a load balancer, sending our request to different origin servers on a back channel. Notably, the probability of getting a different origin server is much higher in the case of different geolocations. This, together with the fact that the number of crawls for VPN and onion are much higher than in the case of user-agent and language settings, explains the comparatively higher number of intra-test inconsistencies.

One inconsistency that is standing out, due to its prevalence in our dataset, was the inconsistent setting of SameSite for the *ASP.NET_SessionId* cookie. According to Microsoft, the framework does not support ".NET versions lower than 4.7.2 for writing the same-site cookie attribute"[4]. Thus, if some of the origin servers have a new version of .NET while others still use the old version, the cookie would show exactly the behavior we observed, which is why we believe this to be a contributing factor.

4.4.3 Inter-Test Inconsistencies

This section sheds light on the inter-test inconsistencies, i.e., for a single deterministic factor such as the User-Agent, our crawls revealed different security guarantees (see middle column of Table 4.4).

Cookie Security The vast majority of sites (144/150) that have inter-test inconsistencies for cookie security are those that deterministically gave back cookies without the Secure attribute to some User-Agents. Notably, the cause of this inconsistency is in most cases (130), special handling for the User-Agent for Firefox on iOS. For example, one site set their *sid* cookie Secure for all clients except Firefox on iOS, leaving those clients unprotected against network attackers. Other sites gave non-Secure cookies to a group of User-Agents that visited their page, another site for Safari-based clients, one for mobile clients, and another one for all iOS clients.

Two sites inconsistently deployed HttpOnly cookies for their clients. In one case, a site delivered *CM_SESSIONID* without HttpOnly attribute to clients that use Firefox on iOS. In another case a site only gave out HttpOnly cookies to non-Safari-based clients. In both cases, attackers can steal or manipulate cookies via an XSS attack and eventually perform state-changing actions on behalf of the user.

For inconsistencies of the SameSite attribute, we found 14 cases where sites either send cookies with the attribute or do not set it at all. One site, for example, only gives SameSite cookies if the Accept-Language header of the client is *not* set to English. As mentioned in the intra-test inconsistencies, this behavior might not be a problem in Chromium-based browsers, because those browsers default to Lax in case of a missing SameSite attribute. However, all Safari-based browsers and Firefox still face the problem that cross-site attacks such as CSRF are possible due to this misconfiguration.

Content Security Policy XSS mitigation as the original use-case of CSP also faced inter-test inconsistencies in ten cases. In general, if a site's CSP alternates between a safe policy and a trivially bypassable one based on some client characteristics, an attacker can specifically target the affected user population. Due to the (at the time of writing) porous support for the 'strict-dynamic' source-expression, some sites had inter-test inconsistencies that only deployed a CSP with this source-expression to clients that actually support it. Numerous sites removed 'strict-dynamic' from their CSP for all Safari (and thus all iOS) clients. The problem here is that `https:` is also present in the policy, i.e., clients without support for 'strict-dynamic' would allow script inclusion from any HTTPS host, which is insecure. Removing 'strict-dynamic' is a bad practice, because the CSP design is backward compatible and unknown source-expressions are

just ignored by browsers. Importantly, Safari recently announced support for 'strict-dynamic' and already supports it in its technology preview [24], hence dropping 'strict-dynamic' may unduly leave Safari users unprotected. Other sites dropped their entire CSP for XSS mitigation for all Safari clients, while again others did not send a CSP at all for Android clients. One Web site only deployed XSS mitigation to some countries (like Russia, Spain, or Sweden), but did not deploy CSP for others (e.g., US, Pakistan, or South Africa).

CSP for framing control is also used inter-test inconsistently across different clients (two sites) and geolocations (18 sites). For example, one site did not send a CSP controlling framing via frame-ancestors to all iOS clients, leaving those users unprotected against framing-based attacks.

Like for the case of intra-test inconsistent deployment of CSP for TLS enforcement, the inter-test inconsistent deployment of this CSP feature does not have a security impact but a functionality impact. However, while it is a randomly occurring problem for the intra-test inconsistencies, the problem deterministically occurs for parts of the user-base on 17 sites.

X-Frame-Options An inter-test inconsistent deployment of X-Frame-Options exposes a part of the user base to framing-based attacks. In seven out of 37 cases, this type of inconsistency occurred due to specific operating systems or browsers are getting different configurations. Some sites deployed XFO for desktop clients, but mobile browsers got no protection at all, making them vulnerable to framing-based attacks. In other cases specific browsers were excluded from the protection: one site did not deploy XFO for Opera clients, while another excluded Firefox browsers. This behavior was also present against users of a specific operating system, as some sites only gave XFO to non-iOS clients. In addition to that, 13 sites (Onion) and 29 sites (VPN) decided to exclude specific geolocations from the protection against framing based-attacks.

Strict Transport Security In case of inter-test inconsistencies in HSTS it makes no difference if HSTS is disabled (max-age=0) or not present because the affected clients/-countries will deterministically get the same insecure configuration. While cross-checking the inconsistent sites with the HSTS preload list, we observed that only five out of the 35 inter-inconsistent sites are actually preloaded.

There are eight Web sites that handle browsers differently. For example, one site only gives enabled HSTS to desktop clients but not to mobile clients, another does not send HSTS to Firefox and Safari-based clients, which exposes parts of the user-base to possible network attacks. In addition to that, 30 sites deploy HSTS inconsistently depending on the geolocation. Another site deploys a proper HSTS for all countries except for clients from India, which do not get an HSTS header. Also, six sites have the inter-test inconsistent deployment of HSTS with/without the includeSubdomains directive. One site, for example, deployed an HSTS header with the directive for clients from some countries such as Hungary or Ireland, but not for others such as Germany or Japan. Here an attacker could abuse subdomains to attack the main domain (e.g., using subdomains to inject cookies into the parent domain).

Reasons for Inter-Test Inconsistencies Inter-test inconsistencies are naturally attributed to deterministic factors, however a few observations are interesting. In many cases flawed User-Agent parsing or wrong handling of the parsed browser information seem to be a problem. Surprisingly many inter-test inconsistencies happened specifically for Firefox on iOS. Therefore we tested this User-Agent in different parsing libraries. All of them showed Firefox with version 40 (released August 11, 2015) as output for our Firefox iOS User-Agent string. In the case of Firefox, the version numbers for the iOS client are different from other operating systems, possibly due to the fact Firefox is based on WebKit instead of Gecko on iOS, so the User-Agent is incorrectly recognized as a legacy client. Notably, the Firefox iOS version number recently jumped from 40.2 to 96.0 on January 18, 2022 [87].

Not only the version number of Firefox, but also the iOS version number present in the User-Agent was the reason for some of the inconsistencies. The User-Agent from an online repository used in our crawler had an old iOS version number (12.1, October 2018). However, with the same Firefox and WebKit version, but a newer iOS version (15.2, December 2021), these inconsistencies were not present, although they are still concerning for a specific user population. Indeed, users may not have control of their OS version due to hardware restrictions.

Also, as mentioned in Section 4.4.3, some sites deliver a CSP without the 'strict-dynamic' expression to Safari-based clients. During our notification campaign, a videotelephony service confirmed that they are doing this because those clients lack support for this CSP feature. In either case, none of those special handling for browsers is actually necessary; unknown cookie attributes and unknown CSP source-expressions are simply ignored by browsers. Furthermore if certain features are going to be supported in future release (like 'strict-dynamic' in Safari's current Technology Preview), the special handling for certain browsers might cause security issues because the browser switches are not updated or removed. This highlights that having browser switches for security mechanisms is a dangerous practice, at least if the provided level of security differs.

In case of network related inter-test inconsistencies, possible reasons are similar to those from the intra-test inconsistencies. If misconfigured origin servers are only used for requests from specific countries, or if CDNs cache responses for certain countries longer than for others, we can observe inconsistent deployment of security mechanisms depending on the geolocation. We detected three sites with different peer IP adresses that seem to cause the issue, ten sites with different `cache-control` header, and two with different `x-cache`. For example, based on the `x-cache` header sent by one site we hypothesize that for certain countries like France they have a cache in place, because all requests from there produced a cache hit, while other requests for example from Australia only produces cache errors/misses.

4.4.4 Disclosure

Our findings imply that certain users of the sites under test might be at risk; either because an attacker can target them based on certain properties (e.g., their User-Agent) or can opportunistically exploit the non-determinism of the server. To enable site operators to fix the inconsistencies, but also to gain knowledge about the root cause

of the inconsistencies, we attempted to disclose the issues to all sites using *security@* and *webmaster@* aliases. The email that we sent contained information about our institutions and us, as well as a detailed description of the individual inconsistent headers and how they were collected. Also, we informed site operators that we are interested in the reason for the inconsistency such that we can better help others that face similar issues and offered them our assistance and further information.

In total, we sent out 256 emails (see Appendix B.1 for the template). For 197 domains, we received an email delivery failed message. Notably, we sent the email to both aliases (security@ and webmaster@), so we might have received a failure message, although one of the two addresses received our email. Research has shown that scaling up notifications is a known problem [126, 125], also due to the low availability of generic aliases [120]. In addition to that, only 25 out of the 256 domains hosted a `security.txt`, with 7 of those setting their contact email to *security@*. Thus, we only got 21 answers that were more than just an automatic response message. Seven operators asked us to provide more details, like the IP addresses of the servers that we connected to. One of those even asked us to provide a demo video that shows the inconsistency problem. In all cases, we were happy to provide them with more detailed data in order to ease their search for the reason behind the issue. Additional seven claimed that they can confirm the issue and will get right back to us, which nearly none of them have done so far. The other seven answered us that they confirmed and fixed the issue or explained to us that this is out of their control, e.g., because they are not self-hosting their sites in some countries.

Many of those that answered instructed us to contact HackerOne to report vulnerabilities. Notably, our message did not include the word “vulnerability” or similar words like “exploit”. Therefore, we answered those emails that we were not interested in any bug bounty, because we only wanted to help and raise attention for the inconsistent behavior such that all clients can be secured consistently. Notably, none of the notified parties answered that this issue is not present in their option, which further strengthens our confidence in the results. The previous *Reasons for* subsections have outlined some of the answers from our disclosure campaign that we used to reason about the inconsistencies in some of the case studies. To increase remediation rates, we tested the problematic sites again in May 2022. Here, we found that 184 still contained the issues we attempted to disclose before. By manually investigating those sites, we were able to find 105 email addresses. In this second round, only four of the manually curated email addresses responded with a failure message.

4.5 Special Case: XFO vs. CSP frame-ancestors

This section explains our work ”*A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web*” [P2], that evaluates the inconsistencies between X-Frame-Options and CSPs frame-ancestors.

4.5.1 Motivation

Clickjacking protection on the modern Web can be enforced via the X-Frame-Options header or the `frame-ancestors` directive of the Content Security Policy. Even if each of those mechanisms individually is deployed consistently according to our definition in Section 4.3, the delegation of the protection to the client opens room for inconsistencies in the security guarantees offered to the users of different Web browsers. In particular, inconsistencies might arise due to the lack of support for the different levels of CSP as well as the different implementations of the underspecified XFO header. This section presents our analysis of the problem of inconsistencies in framing control policies across different browsers. We implement an automated policy analyzer to assess the state of click-jacking protection on the Web. Based on those results we then propose recommendations for Web developers and browser vendors to mitigate this issue. And we also designed and implemented a server-side proxy to retrofit security in Web applications such that every client gets the same level of protection against framing-based attacks.

4.5.2 Analyzing Framing Control Policies

Our colleagues Stefano Calzavara and Alvise Rabitti from Università Ca'Foscari Venezia created a formal semantics of every framing control policy on top of their CoreCSP framework, which provides a simple denotational semantics for the content restriction fragment of CSP [21]. The formal definitions, that check if a framing policy (either via XFO, via CSP frame-ancestors, or using both), is a *Consistent*, *Security-Oriented*, or *Compatibility-Oriented* Policy.

In essence, a *Consistent* Policy is a policy where all browsers have the same behavior. A *Security-Oriented* policy is one, where all legacy browsers enforce at least the level of security that modern browsers, while in a *Compatibility-Oriented* policy where all modern browsers enforce at least the level of security of legacy browsers. Thus, a policy where a site only deploys `frame-ancestors https://*.example.com`, is inconsistent, as it restricts framing in modern, but does not in legacy browsers. At the same time, it is compatibility-oriented as legacy browsers and does not restrict framing. If, however, at the same time XFO `DENY` is deployed, it would no longer be compatibility, but rather a security-oriented policy.

Inconsistent policies which are neither security-oriented nor compatibility-oriented are generally hard to justify as correct because they fall in one of the following cases: (1) two legacy browsers interpret the policy differently; (2) two modern browsers interpret the policy differently; (3) none of the above is true, yet legacy browsers and modern browsers give two incomparable interpretations of the same policy. We refer to such policies as *unduly inconsistent*.

Based on the formal definitions we designed and implemented FrameCheck, an automated analyzer of framing control policies of real-world Web sites. Given a URL to analyze, FrameCheck produces a security report on its state of click-jacking protection. Our implementation supports the 10, at that time, most popular browsers according to data from *Can I Use*.[3] For each browser, we downloaded the latest avail-

[3] `https://caniuse.com`

Browser Name	Type	Version	Market
Chrome	Desktop	76	$\sim$ 23%
Chrome for Android	Mobile	76	$\sim$ 35%
Edge	Desktop	18	$\sim$ 2%
Firefox	Desktop	69	$\sim$ 4%
Internet Explorer	Desktop	11	$\sim$ 2%
Opera Mini	Mobile	44.1	$\sim$ 1%
Safari	Desktop	12.3	$\sim$ 2%
Safari for iOS	Mobile	12.3	$\sim$ 10%
Samsung Internet	Mobile	10.1	$\sim$ 3%
UC Browser	Mobile	12.12	$\sim$ 3%

Table 4.5: Browsers considered in the present study

able version with at least 1% of market share[4] and we reverse-engineered its semantics through an exhaustive set of test cases (see Section 4.5.2.1). The set of browsers under study is shown in Table 4.5: only two browsers do not support framing control via CSP, i.e., Internet Explorer and Opera Mini, which we deem as a legacy. Note that, according to *Can I Use*, Opera Mini does not support any mechanism for framing control. However, we installed the latest available version from the Google Play Store, and, according to our tests, Opera Mini, in fact, supports XFO.

When given a URL, FRAMECHECKaccesses this Web site, once for each browser in our dataset by sending the corresponding User-Agent, and then produces a security report on policy consistency based on our formal definitions.

4.5.2.1 Test Cases for FRAMECHECK

In total, we developed more than 40 test cases to reconstruct the semantics of the underspecified XFO header in our set of browsers. We designed the test cases through a careful analysis of the XFO specification [108] and a preliminary inspection of a large set of framing control policies collected in the wild by a simple crawler. Hence, the test cases are not esoteric examples of problems that might possibly arise in theory, but rather represent classes of potentially ambiguous policies that we observed in practice. We report below on the most interesting findings.

Support for ALLOW-FROM: Only 3 out of 10 browsers actually support the XFO ALLOW-FROM directive: Edge, Firefox[5], and Internet Explorer. This means that every Web page that deploys ALLOW-FROM, but not a corresponding CSP, implements inconsistent protection against click-jacking and leaves (at least) 7 browsers unprotected. We also tested what happens when the ALLOW-FROM directive is not followed by a valid serialized origin, as mandated by the XFO specification. In all the cases we tested, the browser implementations were conservative and denied any framing. There is one exception to this rule, though: Edge also supports the use of ALLOW-FROM with a hostname instead of an origin as a value.

[4]Note that Chrome derivates like Brave also show their UA as Chrome, leading to a slight over-approximation of Chrome usage.

[5]During our project, Firefox dropped support in version 70

Browser	CSP	Allow-From	Multi-Header	Parsing	Double Framing
Chrome	✓	✗	✓	✓	✓
Chrome for Android	✓	✗	✓	✓	✓
Edge	✓	✓	✗	✗	✗
Firefox	✓	✓	✓	✓	✓
Internet Explorer	✗	✓	✗	✗	✗
Opera Mini	✗	✗	✗	✗	✓
Safari	✓	✗	✓	✓	✓
Safari for iOS	✓	✗	✓	✓	✓
Samsung Internet	✓	✗	✓	✓	✓
UC Browser	✓	✗	✓	✓	✗

Table 4.6: Framing control semantics of popular browsers

Support for Multiple Headers: When the same Web page sends multiple XFO headers, 7 out of 10 browsers simultaneously enforce all of them. Unfortunately, we observed that Edge, Internet Explorer and Opera Mini only enforce the first header and discard the other ones, which might lead to inconsistencies if the first one is laxer than the others.

Parsing of Header Values: According to the HTTP specification [44], it must be possible to replace multiple headers with the same name with a single header that includes a comma-separated list of the header values. Therefore, the standard implies that browsers must be able to handle XFO values like `SAMEORIGIN, DENY` correctly, and prevent framing. However, we discovered unexpected behaviors in 3 browsers: Edge, Internet Explorer and Opera Mini, as they do not split the header value on commas and rather parse the list as a single value, which is interpreted as a non-existing directive, i.e., not enforcing any framing restriction. This also happens when the *same* directive is repeated multiple times, such as in the case of `DENY, DENY`. This behavior has a particularly subtle implication on the interpretation of the XFO header that tries to allow multiple third parties (`ALLOW-FROM <orig1>, <orig2>`). Firefox parses this policy as two separate headers, one allowing framing from the first origin and the other one is an incorrect value that does not enforce any framing restriction: as a result, framing is only allowed from the first origin. Internet Explorer, instead, blocks every form of framing, since ALLOW-FROM is not set to a serialized origin. Notably, none of these two interpretations matches what the Web developer likely had in mind, i.e., allowing two different origins.

Double Framing Protection: Finally, we observed that most browsers implement XFO in a way that is robust against double-framing attacks. This shows that the current implementation has improved since the original XFO specification when all browsers used to perform origin checks against the top-level browsing context alone. However, still 3 browsers are susceptible to double framing attacks: Edge, Internet Explorer, and UC Browser. Notably, this part of the work does not consider inconsistencies arising from double framing, because otherwise even trivial XFO policies like `SAMEORIGIN` would be considered inconsistent and bias our study.

4.5.2.2 Summary of FrameCheck

The summary of our analysis is shown in Table 4.6. Based on our extensive set of test cases, we identified 6 different semantics across the 10 browsers we considered, without counting the unexpected support for hostnames in ALLOW-FROM implemented in Edge: this means that the room for inconsistent click-jacking protection is significant. Out of the 10 tested browsers, Firefox 69 is the only one that faithfully implements the specifications we checked, while Opera Mini offers little to no protection against click-jacking because it does not implement CSP, it does not support ALLOW-FROM, and even basic XFO directives like SAMEORIGIN and DENY can be incorrectly enforced due to other quirks in the treatment of HTTP headers.

4.5.3 FrameCheck Results

In this section, we report on a large-scale analysis performed in the wild with our policy analyzer. Our analysis shows that many popular Web sites implement inconsistent protection against click-jacking and sheds light on the root causes of this potential security problem.

4.5.3.1 Data Collection

To assess inconsistencies at scale, we decided to analyze the top 10,000 sites from the Tranco list of October 29, 2019. As we did not only want to check the start pages in a static manner, we instead used a Chrome-based crawler to visit the start pages, collect all links on them, and follow those links up to at most 500 items per site. (Here, “site” refers to the registrable domain name or eTLD+1.) In doing so, we did not only collect the headers delivered with the pages we visited, but also those of all iframes on the visited pages. This way, we were able to (partially) account for sites where only specific pages are protected against framing-based attacks. We then relied on Python’s Requests library to collect the XFO and CSP headers of those URLs once for each of the different user-agent strings considered in our study. Notably, Requests folds multiple response headers with the same name into a comma-separated list, as specified in RFC 7230 [44]. As discussed in Section 4.5.2.1, browsers do not necessarily follow this specification, but might rather consume each header separately, meaning that Requests’ approach to parsing headers would not properly account for that. Therefore, in case we detect a comma in either the XFO or CSP header, we fall back to curl, which outputs the headers line-by-line. To further improve resiliency against possible crawling errors, we filtered out from the dataset all the pages where we observed that at least one user agent was not receiving the XFO or CSP headers, while other user agents were. Though this might lose some inconsistencies, e.g., when CSP headers are not actually sent to legacy browsers, we preferred to be conservative and work on more reliable data rather than risking to unduly exacerbate the number of inconsistencies in the wild. In particular, we found that several pages did not consistently deliver the same XFO and/or CSP headers, even when visited multiple times with the same User-Agent string. Finally, we performed a de-duplication of the collected framing control policies by removing all the duplicate combinations of XFO and CSP policies collected

Defense	Number of Policies	Inconsistencies
Just XFO	15,415 (88%)	290 (16%)
Just CSP	714 (4%)	705 (39%)
XFO + CSP	1,484 (8%)	805 (45%)

Table 4.7: Defenses used in the collected policies

within the same origin, to avoid biasing the dataset construction towards origins with hundreds of pages all using the same policy.

At the end of the data collection process, we visited 989,875 URLs overall. Of those, 369,606 URLs (37%) across 5,835 sites carried either an XFO or CSP header aimed at framing control. After the dataset cleaning and the de-duplication process explained above, we were left with 17,613 framing control policies. Table 4.7 shows the adoption of the different security mechanisms in the different policies. We observe that XFO is still the most widespread defense mechanism against click-jacking in the wild by far, yet around 12% of the collected policies make use of CSP.

4.5.3.2 Inconsistent Policies

Overall, we identified 1,800 policies from 1,779 origins implementing inconsistent protection against click-jacking, i.e., where the enforced level of protection is dependent on the browser. This is 10% of the analyzed policies, which is already a significant percentage. But this result becomes even more concerning when we take a look at which click-jacking protection mechanisms are used by such policies.

Table 4.7 provides the breakdown: the relative majority of the inconsistencies (45%) occur when XFO and CSP are used together, which suggests that having two different mechanisms for the same purpose is potentially dangerous. Moreover, note that 805 out of the 1,484 pages (54%) which make use of both XFO and CSP together implement inconsistent protection against click-jacking, i.e., it is more likely to get the combination of the two defenses wrong than right.

Another interesting insight from our analysis is that 84% of the inconsistent policies make use of CSP. Intuitively, this seems related to the fact that the set of browsers we consider includes some legacy browsers without CSP support: in particular, Opera Mini provides very limited tools to protect against click-jacking. Hence, one might think that inconsistencies are motivated by its presence alone, yet this is not the case: if we removed Opera Mini from the set of browsers, the number of inconsistent policies would drop from 1,800 to 1,749, which is roughly a 3% reduction. One might then try to also remove Internet Explorer from the picture, since it also lacks support for CSP. However, this is a different story than Opera Mini, since Internet Explorer supports the ALLOW-FROM directive. Hence, inconsistencies could be fixed by simulating the behavior of CSP through different values of ALLOW-FROM based on the Referer header.

To understand the prevalence of such practice in the wild, we set up the following experiment: for each page in our dataset, we identify the hosts which are allowed framing according to CSP, and we send an HTTP request to the page with

Inconsistency Reason	Number of Policies	Fraction
Use of the ALLOW-FROM directive	323	78%
Comma-separated directives in XFO header	94	23%
Incomparable policies in XFO and CSP	53	13%
Use of multiple XFO headers	16	4%
Different policies sent to different browsers	5	1%

Table 4.8: Practices in unduly inconsistent policies (classes might overlap)

the Referer header set to one of such hosts. In the presence of wildcards in CSP, e.g., `*.example.com`, we generate a synthetic candidate Referer matching them, e.g., `https://test.example.com`. If we observe that the value of the Referer is reflected back in the XFO header of the response, it means that we might have false positives in our set of inconsistencies, because the originally collected XFO headers only provided a partial picture of the deployed policy. We managed to perform this test on the 2,198 pages with CSP and observed extremely low adoption of Referer sniffing: in particular, only 11 pages relied on such practice. This gives us confidence in the correctness of the conclusions we draw.

In the next section, we provide an in-depth analysis of the inconsistent policies we collected. We do this while considering the full set of browsers in Table 4.5, because those browsers are actively used, and we want to assess the state of click-jacking protection on the Web as of now. We elaborate on the impact of the chosen browsers on our study in Section 4.5.3.8.

4.5.3.3 Analysis of Inconsistent Policies

To have a more in-depth look into the set of inconsistent policies, we performed a further classification step: in particular, we identified 590 security-oriented policies (33%) and 795 compatibility-oriented policies (44%), while the other 415 inconsistent policies (23%) do not belong to any of these two classes, hence are unduly inconsistent. In the rest of this section, we perform an in-depth analysis of the collected inconsistent policies and identify dangerous practices therein.

4.5.3.4 Security-Oriented Policies

The existence of security-oriented policies is justified by the fact that XFO is less expressive than CSP, hence Web developers might be led into shipping XFO headers that are more restrictive than the corresponding CSP headers. For example, the Web site `https://www.icloud.com` deploys an XFO header set to SAMEORIGIN and a CSP whitelisting every subdomain of `icloud.com` and `apple.com`. A similar situation happens on `https://academia.stackexchange.com`, which sets XFO to SAMEORIGIN and uses CSP to whitelist both itself and `https://stackexchange.com`. These policies offer a good level of protection to legacy browsers, but might introduce compatibility issues therein.

We further categorized the 590 security-oriented policies in two classes. The first class includes ineffective policies, where CSP is overly liberal compared to XFO: these

policies allow framing from any host on CSP-enabled browsers, possibly just restricting its scheme, hence modern browsers are left unprotected. We noticed this problem just in 13 cases (2%), and we conjecture it might come from the wrong assumption that, when both XFO and CSP are enabled, they are both enforced, while CSP actually overrides XFO and voids protection. However, it is positive to see that this class of policies is highly under-represented. The other policies all take advantage of the additional expressive power of CSP over XFO for fine-grained whitelisting: specifically, we observed 99 cases (17%) where CSP was used to whitelist all the subdomains of the host whitelisted via XFO, while in all other cases CSP whitelisted at least two source expressions.

To the best of our knowledge, these look like legitimate use cases, where policy inconsistency is not necessarily dangerous for security. However, this discrepancy raises concerns, because it implies that either legacy browsers suffer from compatibility issues due to overly harsh security enforcement, or modern browsers are excessively liberal in their treatment of framing, i.e., the policies violate principle of least privilege.

4.5.3.5 Compatibility-Oriented Policies

Compatibility-oriented policies might be justified by the need to make Web applications accessible by legacy browsers, at the cost of (partially) sacrificing security in that case. For example, the Web site `https://www.spotify.com` deploys a CSP whitelisting every subdomain of `spotify.com` and `spotify.net`, but does not ship any XFO header, likely because XFO does not support such expressive whitelists. Another similar example is `https://www.sony.com`, which does not deploy XFO, but uses CSP to allow framing from itself and all the subdomains of three other trusted sites.

Recall that our dataset contains 795 compatibility-oriented policies. The first analysis we perform aims at understanding how much security legacy browsers sacrifice for such policies. For the very large majority of compatibility-oriented policies, we observed that XFO does not provide any protection at all, i.e., framing is allowed from any origin: this happened in 758 cases (95%). In particular, we found 705 pages where an XFO header is entirely absent (89%) and 99 pages where the XFO headers contain an incorrect directive or are misinterpreted by some legacy browser (11%). This shows that most Web developers are not actually concerned about offering security to users of legacy browsers, or are just entirely unaware of the existence of this problem.

To get a better understanding of the reasons underlying the existence of compatibility-oriented policies, we analyze the combination of XFO and CSP for the following scenario: if CSP is used to whitelist at most one origin, it is straightforward to write an XFO header which enforces exactly the same restrictions, hence the adoption of a compatibility-oriented policy is unjustified. We observe that this was the case for 105 policies (13%), where protection could be improved with minimal effort and expertise by the Web developers, i.e., without resorting to Referer sniffing. This shows that the bleak picture given above could be easily improved to some extent, yet this is not happening in practice.

Inconsistency Class	Vuln. (Any Browser)	Vuln. (Modern Browser)
Security-Oriented	13 (2%)	13 (2%)
Compatibility-Oriented	758 (95%)	3 (<1%)
Unduly Inconsistent	380 (92%)	278 (67%)
Aggregate	1,151 (64%)	294 (16%)

Table 4.9: Presence and distribution of vulnerable policies

4.5.3.6 Unduly Inconsistent Policies

Finally, we focus on the 415 inconsistent policies that are neither security-oriented nor compatibility-oriented. These policies are hard to justify as secure, or even as intended, as explained in Section 4.5.2. In particular, we observe the following distribution of (possibly overlapping) classes:

- 315 policies are interpreted differently by at least two legacy browsers (76%);
- 289 policies are interpreted differently by at least two modern browsers (69%);
- 29 policies are given the same interpretation by all legacy browsers and all modern browsers, yet these two interpretations are incomparable (7%).

What is worse is that 380 of these policies (92%) do not enforce any form of framing restriction on at least one of the browsers considered in our study, which confirms that this class of inconsistencies is particularly dangerous for security. For example, the Web site `es.sprint.com` sets an XFO header to ALLOW-FROM `https://www.sprint.com`, but does not ship a companion CSP, leaving legacy browsers unprotected. As another example, `whois.web.com` sends two XFO headers, one set to SAMEORIGIN and one set to DENY, which allows same-origin framing in some browsers but not others.

It is instructive to have a look at why these undue inconsistencies arise. Table 4.8 provides the breakdown of the main practices leading to policy inconsistency (classes partially overlap). We observe that the ALLOW-FROM directive is present in most of the unduly inconsistent policies, which shows that XFO is not properly coupled with CSP in those cases. Indeed, 322 out of 465 policies that use ALLOW-FROM do not come with any CSP (69%) and do not offer any protection on most modern browsers. It is also interesting that we found 53 policies where both XFO and CSP are syntactically correct, yet express incomparable policies. For example, we noticed that `https://gfp.sd.gov` deploys an XFO header set to SAMEORIGIN, while its CSP allows framing from every subdomain of `arcgis.com`, `soundcloud.com` and `flipsnack.com`. We do not have definite explanations for this kind of policies, but a plausible reason could be that XFO was deployed for a legacy version of the Web site and never updated later.

4.5.3.7 Perspective

We summarize here the security impact of our findings by computing the number of policies that do not offer any level of protection to at least one browser. We also present

the same perspective for modern browsers alone. The presence and distribution of vulnerable policies for these two cases are shown in Table 4.9. These numbers confirm our claim that not all inconsistencies are necessarily dangerous, yet their majority actually is (64%). In particular, almost every inconsistent policy that is not security-oriented is completely ineffective on at least one browser. Luckily, our experiments also show that users of modern browsers enjoy a significantly higher level of protection than users of legacy browsers since only 16% of the inconsistencies actually void any form of security enforcement in a modern browser, where undue inconsistencies are essentially the only threat.

4.5.3.8 The Role of Browsers

Since we assess inconsistencies over a set of popular browsers, one might wonder to which extent the chosen browsers bias the results of our study. To understand this point, we decided to run a second analysis by removing Internet Explorer and Opera Mini from the set of browsers under test. The rationale of this choice is that these browsers do not support CSP, and thus, we might get a picture of how much the current policy deployment would be inconsistent in a world without legacy browsers. It turns out that the total number of inconsistent policies would drop from 1,800 to 289, which is a major improvement. However, observe that all such policies fall in the class of unduly inconsistent policies (since we removed legacy browsers), and we computed that for 278 of them (96%) there is at least one modern browser which does not enforce any form of restriction. This confirms that the adoption of modern browsers strongly mitigates the problem of inconsistencies, yet not entirely solved. The main reasons for inconsistency would still be the use of ALLOW-FROM and the adoption of a comma-separated list of directives in XFO.

It is also particularly interesting that two of the browsers that we tested have been undergoing major changes at the time of writing. The first significant change was implemented in Firefox, which dropped support for the ALLOW-FROM directive in version 70.[6] Moreover, Microsoft announced that Edge will move to the Chromium architecture in 2020, which likely means that it will drop support for ALLOW-FROM and fix the problems with XFO headers. These changes go in the direction of reducing the risk of inconsistencies in modern browsers, which will eventually be uniformed to Chromium derivates. Unfortunately, we also showed that 322 out of 465 policies that use ALLOW-FROM do not come with any CSP (69%), which implies that these changes are weakening the state of click-jacking protection on the Web.

At the end of the day, we believe that the problem of inconsistencies in click-jacking protection is far from solved. Though legacy browsers not supporting CSP are likely going to disappear in a few years, it is hard to predict a precise temporal horizon for this: for example, Internet Explorer 11 was launched in 2013, and it still has $\sim 2\%$ of the market share based on publicly available data, while Opera Mini is still under active development and extremely popular with around 15% market share in Africa, where mobile traffic is still expensive [93]. Also, it should be noted that the versions of Edge and Firefox considered in the present study might still be around for a while, i.e., the

[6] https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/70#HTTP

Web platform will still be accessed by browsers supporting ALLOW-FROM at least in the near future. Though a full transition from XFO to CSP for click-jacking protection is the way to go to solve the issue of inconsistencies, the setting is complex and requires actions at different levels. We discuss recommendations and countermeasures in the next section.

4.5.3.9 Limitations

Though we strived to quantify the security impact of the detected policy inconsistencies, we cannot show that even policies that do not provide any form of framing control in some browsers lead to exploitable vulnerabilities in practice. To overcome this limitation, we would need to identify pages that are susceptible to framing-based attacks. However, identifying these in an automated fashion at a large scale requires accounts of all tested sites as well as an in-depth understanding of the application's semantics. However, we argue that it is fair to assume that site operators are deploying framing control for a reason. In our opinion, the widespread adoption of framing control policies (33% of all crawled URLs, spread across 58% of the sites we looked at) motivates that click-jacking is perceived as an important security threat. Our analysis acts as a cautionary tale aimed at raising awareness of the potential issues that arise from policy inconsistencies.

In addition to this, we also remark that our study specifically focuses on the 10,000 most popular sites at the time of writing the paper. Given the diversity of the Web in general, this does not necessarily enable us to generalize about framing control inconsistencies on the entire Web. As prior work has shown [133], though, the popularity of domains often represents a proxy for security measures, meaning that our results most likely are a lower bound of the actual problems discoverable in the wild.

4.5.4 Recommendations & Countermeasures

Based on the data gathered in our analysis of both browser implementations and real-world deployment of framing control, we discuss lessons learned to improve the situation. In particular, we first present recommendations for both Web developers and browser vendors, highlighting some room for improvement which we found. We then discuss our implementation of a server-side proxy capable of retrofitting framing control policies in existing Web applications for the diverse set of browsers we considered in our analysis.

4.5.4.1 Recommendations for Web Developers

The first important recommendation we make is that both XFO and CSP must be used for effective framing control on the current Web. XFO alone is insufficient for security because sites might be prone to double framing attacks (also in modern browsers like UC Browser) or even not protected at all (most notably, in the presence of the largely unsupported ALLOW-FROM directive). On the other hand, just using CSP results in leaving users of legacy browsers completely unprotected. Unfortunately, we found that only 8% of the collected policies use both XFO and CSP. Worse, the combination of the

Directive 1	Directive 2	Conjunction of Directives
SAMEORIGIN	SAMEORIGIN	SAMEORIGIN
SAMEORIGIN	ALLOW-FROM o'	DENY if $o \neq o'$, SAMEORIGIN otherwise
SAMEORIGIN	DENY	DENY
ALLOW-FROM o'	ALLOW-FROM o''	DENY if $o' \neq o''$, ALLOW-FROM o' otherwise
ALLOW-FROM o'	DENY	DENY
DENY	DENY	DENY

Table 4.10: Simplification of multiple XFO directives into a single one (adoption at origin o)

two mechanisms proved hard to get right for Web developers, as 54% of such policies are inconsistent.

The other crucial recommendations are about the use of XFO. Web developers should ensure that at most one XFO header is sent with every Web page because existing browsers have inconsistent interpretations in the presence of multiple XFO headers. What is worth noting here is that there is no good practical reason to deploy more than one XFO header. In the presence of multiple XFO headers, existing browsers either enforce the first one (thus voiding the others) or simultaneously enforce all of them. However, even this is useless, because any pair of XFO directives always contains either redundant or contradictory information, which can be expressed with a single XFO directive (see Table 4.10). For the same reasons we just discussed, Web developers should avoid the use of comma-separated values in XFO headers. These headers are parsed as multiple XFO headers in most browsers, while in other browsers, they are interpreted as non-existing directives that do not enforce any form of framing control. This latter observation shows that even the apparently innocuous practice of repeating the same directive multiple times is actually insecure because it voids protection on some browsers.

4.5.4.2 Recommendations for Browser Vendors

Though the `frame-ancestors` directive obsoleted XFO back in 2014, 88% of the policies we collected are still based on XFO alone. This means that this is not the right time to drop support for XFO, and one might wonder if this will ever be possible without leaving a significant fraction of the Web unprotected. An important point we would like to stress is the need for more informational messages for Web developers, e.g., in the JavaScript console. A prime example of this issue comes from the recent removal of support for ALLOW-FROM in Firefox. When visiting a page that sends an XFO header containing such a directive, Firefox merely notes an invalid header and points the developer to the generic Mozilla Developer Network page on XFO. This page does note that ALLOW-FROM is now obsolete and should not be used, but does not provide an immediately visible and explicit warning that sites using ALLOW-FROM have suddenly become unprotected. As to Chrome, the JavaScript console only shows a warning about an unrecognized directive and nothing more.

We argue that browsers should explicitly warn Web developers about the possibility of using CSP to achieve the same effect of XFO, which is straightforward considered that CSP is more expressive than XFO. In particular, XFO policies which do not contain glaring mistakes can be readily transformed into corresponding CSPs. We designed one such solution as part of our server-side proxy (see Section 4.5.4.3), which might be inspiring also for browser vendors since the same approach could be applied at the client. We understand that major browser vendors might consider such transformations dangerous for backward compatibility, yet even simple transformations might significantly increase security in the wild and are worth testing in our opinion. At the very least, a candidate value for `frame-ancestors` combined with a clear warning about the unprotected state of the site should be reported in the JavaScript console.

On more general terms, we think that our paper shows the importance of implementing only client-side security mechanisms that come with a clear and precise specification. The XFO specification was put together only after major browsers already implemented support for the XFO header, which led to many different implementations. Though the auto-update feature of modern browsers certainly helps in mitigating the problem of inconsistencies, real-world market share data show that legacy browsers are hard to eradicate. Once a client-side security mechanism has been inconsistently implemented across browsers, it might be challenging to understand its long-lasting impact in the wild. For example, without moving away from CSP, the `strict-dynamic` source expression has first been implemented in Chrome due to an independent effort from Google's engineers and then pushed into the CSP standard. This kind of practice is dangerous because other browser vendors might be unwilling to pick up: for example, Safari at the time of teh study still lacked support for `strict-dynamic`.

4.5.4.3 Retrofitting Security

As Web developers might not be aware of the intricacies of the two mechanisms available to control the framing of their sites, we developed a server-side proxy designed to enforce consistency in framing control policies, i.e., to ensure all browsers enforce the same level of protection. The proxy is a Python script ($\sim$ 800 LoC), which can be run at the server. It inspects the HTTP traffic to automatically fix the framing control headers so as to ensure policy consistency. To enable researchers to build on our work and administrators to benefit from the tool, we made the proxy publicly available [A3].

In particular, for any request r, let $\overline{r}$ stand for the corresponding HTTP response. If $\overline{r}$ contains XFO headers, but no CSP header with a `frame-ancestors` directive, the proxy behaves as follows:

1. if multiple XFO headers are present in $\overline{r}$, they are first folded into one XFO header set to a comma-separated list of the specified directives;

2. after step 1, $\overline{r}$ is guaranteed to contain exactly one XFO header. If the header contains a comma-separated list of directives, it is replaced by a single directive enforcing the same security restrictions of the conjunction of the directives. This is always possible, thanks to the simplification rules in Table 4.10;

3. the proxy finally attaches to $\overline{r}$ a new CSP header enforcing the same framing control restrictions of the sanitized XFO header. This is straightforward, since CSP is more expressive than XFO, and does not conflict with other CSP headers possibly present in $\overline{r}$, as all CSPs are enforced.

If $\overline{r}$ contains CSP headers with a `frame-ancestors` directive, the proxy instead behaves as follows:

1. all the XFO headers of $\overline{r}$ are stripped away;
2. the proxy computes the union of the source expressions whitelisted in all the `frame-ancestors` directives contained in the CSP headers of $\overline{r}$;
3. if CSP denies framing, $\overline{r}$ is extended with an XFO header containing the DENY directive. If instead CSP only allows same-origin framing, $\overline{r}$ is extended with an XFO header containing the SAMEORIGIN directive. Otherwise, the proxy checks if the Referer header of r contains a URL allowed by any of the source expressions identified at step 2: if this is the case, $\overline{r}$ is extended with an XFO header containing an ALLOW-FROM directive set to the origin of the Referer header; otherwise, the XFO header is set to DENY. If r lacks the Referer header, the proxy conservatively sets the XFO header to DENY.

Eventually, the proxy ensures the consistency of framing control policies with respect to the set of tested browsers, by equating the security guarantees of XFO and CSP (up to double framing). Observe that, although Opera Mini supports neither CSP nor ALLOW-FROM, the proxy still manages to rectify its limitations. In particular, if the Referer of the request is set to a whitelisted URL, the proxy sets XFO to the corresponding ALLOW-FROM directive, which is just ignored by Opera Mini and framing is allowed. Otherwise, the proxy sets XFO to DENY, and the page cannot be framed.

In our design, we prioritize CSP headers over XFO headers when both are present since CSP is the preferred method to enforce framing control in modern browsers. This means that it is occasionally possible for the proxy to relax security restrictions beyond least privilege: for example, if a page sets XFO to DENY and CSP allows same-origin framing, then XFO will be relaxed to SAMEORIGIN. However, this is sensible from a security perspective, because modern browsers already allow same-origin framing, so we assume this was intended by the site administrators, as modern browsers are the primary target in the market and are also easier to test. This is also backed up by our dataset, where we observed only 13 policies where XFO was tighter than CSP and CSP was configured in an obviously insecure manner (see Table 4.9).

As a final point, we note that the Referer header may be stripped when controlled through the Referrer-Policy [85], which would disable the possibility of performing Referer sniffing in the proxy. However, Referrer-Policy is only supported in browsers that also support the `frame-ancestors` directive of CSP. Since the proxy only relies on Referer sniffing in the presence of `frame-ancestors`, the DENY directive placed in the absence of the Referer header would be overridden by CSP in all cases. After implementing our proxy, we tested it out against the full set of test cases of Section 4.5.2.1. By doing so, we confirmed that the proxy behaves as expected and enforces the same security restrictions in the entire pool of browsers.

4.5.5 Summary of Framing Control Inconsistencies

Our analysis of 10,000 Web sites from the Tranco list showed that the problem of inconsistencies is widespread on the Web since around 10% of the (distinct) framing control policies in the wild are inconsistent and most often do not provide any form of protection to at least one browser. Given the insights into the dangers caused by inconsistencies, we proposed different countermeasures in terms of recommendations for Web developers and browser vendors, as well as the implementation of a server-side proxy designed to retrofit security to existing Web applications.

4.6 Discussion

Here, we discuss limitations of the work and summarize the security impact of our findings.

4.6.1 Limitations

Our analysis already shows that client characteristics play a relevant role for Web application security, however it could be improved along different directions. One limitation of our study is the assumption that all the tested browsers implement all the security mechanisms according to their official specifications, which simplified the technical development. This assumption is motivated by our focus on modern clients, yet we are well aware that it is not entirely accurate, e.g., at the time of writing Safari does not support the 'strict-dynamic' source-expression of CSP Level 3 and browsers might suffer from bugs (like all software), especially in corner cases. That said, we manually vetted most of the detected security inconsistencies and we confirm that they are not subtle enough to invalidate the general findings of our study due to our assumption on browser behavior.

Another limitation of our work is the best-effort attribution of the identified security inconsistencies. Discussing correlation rather than causation is a common and accepted limitation of Web measurements. We crawled each page multiple times and formalized different definitions of consistency to mitigate the effects of non-determinism, however we cannot entirely rule out non-determinism, e.g., due to the presence of server-side load balancers. It is possible that we collected five times the same response from a Web page due to non-determinism, rather than due to our testing conditions, however all the cases explicitly named in Section 4.4 have been manually vetted and confirmed as vulnerable.

4.6.2 Overall Security Impact

In general, an attacker can abuse the inter-test inconsistent behavior of some sites to attack a certain part of the user base by specifically targeting the less secured clients like specific User-Agents or users from certain geolocations. For the intra-test inconsistent sites, an opportunistic attacker can exploit the non-determinism of the deployed security mechanism by executing the attack multiple times. The individual advantage of the attacker in both cases depend on the mechanism that is deployed inconsistently.

For **Cookies**, a missing security attribute enables an attacker to access the cookie via XSS (*HttpOnly*) or to steal the cookie by downgrading the connection security and eavesdropping on the traffic (*Secure*). Also, missing or inconsistently deployed *SameSite* Attributes allows attackers to successfully execute cross-site attacks such as CSRF. In either case, the difference between inter-test and intra-test inconsistencies in the case of cookies does not change the attack itself but only the way it can be successfully executed, because the attacker either needs to target a certain group of users (*inter*), or perform the attack multiple times (*intra*). Therefore the user-base (or party of it) of more than 172 Web sites can be attacked due to inconsistencies.

In case of an inconsistent **Content Security Policy** header an attacker can perform XSS attacks (inconsistent *XSS mitigation*), framing-based attacks such as Clickjacking (inconsistent *frame-ancestors*), or perform network-based attacks (inconsistent *TLS enforcement*). While the latter is only relevant for functionality rather than security, because Chromium-based browsers nowadays auto-upgrade mixed content [27] and Firefox and Safari block it, the other two cases can indeed be exploited by an attacker. Thus in case of inconsistent XSS mitigation and/or inconsistent framing control, the attacker can exploit a certain group of users (*inter*), or try the attack multiple times (*intra*) on 41 different Web sites.

Similar to the exploitability of inconsistent CSP *frame-ancestors*, inconsistent deployment of the **X-Frame-Options** header can lead to framing-based attacks such as Clickjacking. Notably, however, XFO will be ignored by CSP Level 2 supporting browsers as soon as CSP *frame-ancestors* is present. Still, only ten sites that showed inconsistencies in XFO have deployed a CSP that restricts framing. Thus, the users of 43 sites would still be exploitable by performing the attack multiple times (*intra*), and a specific group of users would be attackable on 17 sites.

For **Strict Transport Security** we have cases that lead to different attacks depending on the type of inconsistency. If, for example, the *preload* directive is deployed intra-test inconsistently, an attacker can remove this site from the HSTS preload list by asking for removal of the site multiple times until the HSTS checker encounters the header without preload. For inconsistencies in the *includeSubDomains* or inconsistent *max-age* duration, an attacker can run network attacks against a certain group of users (*inter*), or perform the attack multiple times (*intra*) on 60 different Web sites. Notably, those sites are only cases where the *page similarity* was considered, so the number of potentially exploitable sites could be higher, as HSTS protects the connection security between client and server, and does not care about the actual content of that is delivered via the server.

4.7 Summary

In this chapter we investigated the inconsistent configuration of client-side security mechanisms on top sites across different client characteristics (*inter-test*) or even across multiple communications of the same HTTP request (*intra-test*).

Our measurement has highlighted that client-side security mechanisms are not equally delivered to all clients. Specifically, we found several sites in our dataset that returned different security policies with different semantics in at least some of our tests.

Our findings have implications in three dimensions: first, Web users may receive different protection based on subtle differences in their browser or vantage point (*inter*-test inconsistencies). Second, *intra*-test inconsistencies may enable an adversary to launch attacks in an opportunistic fashion, given that the responses for the same request may non-deterministically enforce different security. Acquiring this knowledge is an easy task for the attacker, as they can probe for non-deterministic behavior of the server as we did. Third, our analysis has shown that prior measurements may have inadvertently under- or over-reported findings with respect to the deployment of security mechanisms. Specifically, we identified intra-test security inconsistencies in 127 sites and inter-test security inconsistencies in 194 sites. Our semantics-based analysis gives clear evidence of the potential security implications of the detected inconsistencies, by identifying characteristics that might enable exploitation while being expressive enough to generalize over previous studies which only focus on missing security headers [110].

To the best of our knowledge, we are the first to systematically study the problem of intra-test inconsistencies. Luckily, dealing with such inconsistencies in Web security measurements appears relatively easy: since most of them (80.4%) are due to unexpectedly missing headers, it suffices to crawl the same page multiple times to detect and fix these omissions. Nevertheless, prior Web measurements on the impact of client characteristics on Web security and privacy might have performed an incorrect attribution of security downgrades, since a single page access does not suffice to assess the impact of non-determinism. Luckily, the number of sites suffering from intra-test inconsistencies is not high enough to invalidate the big picture drawn by prior studies.

Inter-test inconsistencies are likely less surprising to researchers working on Web measurements, due to the publication of papers studying variations of the topic [63, 62]. However, inter-test inconsistencies are particularly concerning to site operators, because they identify weak spots in their security policies reported by our analysis. We observe that inter-test inconsistencies across network access methods might arise due to misconfigured origin server for specific geolocations. Also, User-Agent sniffing leads to security inconsistencies on 177 sites, which can all be attributed to site operators. Notably, due to backwards compatibility of the investigated security mechanisms, none of the individual responses for specific browsers were actually necessary.

In this chapter, we have seen that not all users of a Web site get the same level of security (*RQ2*). However, even if a developer manages to consistently deploy a security mechanism, for example, a consistent CSP, chapter 3 showed that the vast majority of CSPs are trivially bypassable. Thus, consistently deploying a security header is only one of the problems. We still need to investigate what is the root cause of the omnipresent misconfiguration of CSP (*RQ3*), such that we can consistently and effectively mitigate attacks for more of the user of the Web.

5
The Developers’ Struggle with CSP Deployment

Contributions of the Authors

This Chapter is based on *"12 Angry Developers – A Qualitative Study on Developers' Struggles with CSP"* [P4]. Sebastian Roth conceived the study. Sebastian Roth and Lea Gröber designed the study. Sebastian Roth and Lea Gröber implemented the Web application for the screening survey. Sebastian Roth created the Web application for the main study and conducted all interviews. Sebastian Roth and Lea Gröber coded all interviews and evaluated the results. Ben Stock, Michael Backes, and Katherina Krombholz supervised the study. All authors commented on the text and contributed to individual sections.

5.1 Motivation

In Chapter 3 we have seen that the vast majority of Content Security Policies (CSPs) that are deployed to defend against Cross-Site Scripting (XSS) are, and have always been, trivially bypassable. Numerous other works [137, 136, 19, 122] confirm this omipresent misconfiguration of CSP, but the reasons remained largely hidden.

While Chapter 3, but especially Steffens et al. [122], attributed this to the high reliance on inline event handlers, which are not trivial to allow by a CSP, those investigations were only educated guesses based on deployment histories or inclusion trees of Web applications.

The survey conducted during the work presented in Chapter 3 revealed that developers are often not knowledgeable about all CSP capabilities and that its complex content control mechanism is blocking the easy-to-use features of CSP for framing control and TLS enforcement. Importantly, none of the prior works actively focus on the developers and their mindset, experience, and problems when *deploying a CSP*.

To close this gap in research regarding CSP, we conducted a study with 12 developers who are familiar with the development of a CSP. The study consists of (1) a semi-structured interview, including a drawing exercise, and (2) a coding task involving creating a CSP for a small Web application. Our findings suggest that not only the complexity of the mechanism but also inconsistencies in how the different browsers and frameworks handle and support CSP and how they report and assist the developer during CSP deployment cause issues. In addition to that information sources regarding CSP not only push developers in the wrong direction but nearly emphasize the usage of insecure practices in the policy.

To sum up, our work provides the following contributions:

1. We present the first qualitative study with 12 real-world Web developers to evaluate the usability of the CSP.
2. We investigate the Web developers' mindset regarding the different attacker models that CSP covers.
3. We uncover the root cause of insecure CSP deployment in order to improve the usability of the security mechanisms' initial deployment.
4. We provide a methodological discussion of conducting an online interview study along with coding and drawing tasks with developers and share the lessons learned from that.

5.2 Methodology

To uncover the underlying problems in the CSP deployment process, we examine developers' mental models of CSPs, as well as their real-world experiences with the mechanism. We complement the data with insights from a controlled coding task. Hence, we provide in-depth qualitative results that fill the gaps of previous quantitative studies [P1]. Our findings can be used to improve the mechanism and ultimately remove roadblocks to its successful adoption. Accordingly, we answer the following research questions:

RQ3.1: What are the root causes of insecure practices when deploying a CSP?

RQ3.2: What strategies do developers adopt when creating a CSP?

RQ3.3: How well do developers understand the associated threat models of CSP?

RQ3.4: What are the perceptions and motivations of developers in terms of deploying a CSP?

We carefully designed our qualitative study to account for the research questions. Thus, we combined semi-structured interviews with a controlled coding task to cover both real-world experiences and in-situation programming, which allows us to take a holistic view of the topic. Our study consists of three parts: (1) a 3min screening survey that covers basic information about the participants' demographics and their familiarity with the technologies involved in CSP. We use this information to ensure that our set of participants is as diverse as possible; (2) a 45min semi-structured interview covering perceptions and prior experiences with CSP, as well as a drawing task about associated threat models of CSP; (3) a 45min coding task in which participants created a CSP for a small Web app in a programming language of their choice (Python, JS, or PHP). The total financial cost of this study is 800€, each participant received 50€ compensation for participation, and the LinkedIn campaign cost 200€. In the following sections we provide details on each part of our methodology, the methods and choices regarding study population and recruitment, as well as evaluation and ethical considerations.

5.2.1 Recruitment and Participants

Our target population is real-world developers that actively deployed, try to deploy, or are testing a CSP. To get in touch with this group, we first tried to find participants using our national chapter of the Open Web Application Security Project (OWASP) Foundation. By using this channel, we are not only reaching developers of big Web applications behind known companies but also small development teams, which increases the diversity in our study population. In addition to the tweet of the OWASP, researchers of our institution held a talk at an OWASP event and promoted our study to all attendees. For the invitation via the OWASP, we created a poster with all necessary information about our study (see Figure 5.1). Those include the goal and the procedure of the study, as well as the amount of time required for participation and the compensation. In addition to that, we also used targeted advertisements on the business social network LinkedIn. Similar to the poster for the OWASP recruitment, we designed a

sponsored content post (see Figure 5.2) that includes the goal, the procedure, the time required, and compensation. The advertisement was targeted towards Web developers and associated with the official account of our institution to underline the soundness of the invitation.

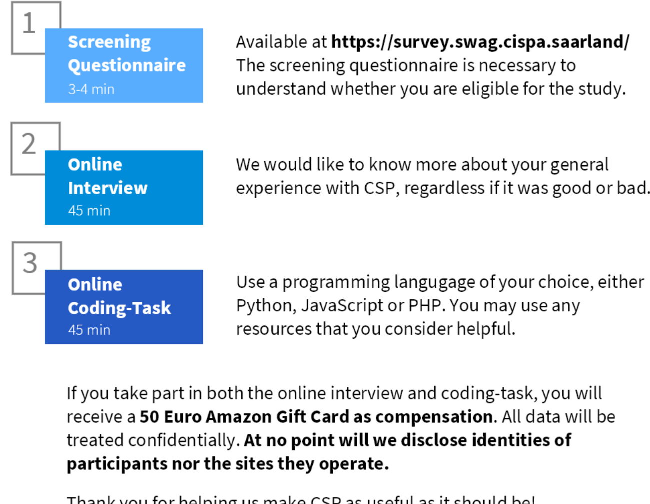

Figure 5.1: Poster that was used by the OWASP to advertise our study.

We also tried other recruitment techniques, which did not lead to any new possible participant completing our screening survey. We created a recruiting email (see Sec-

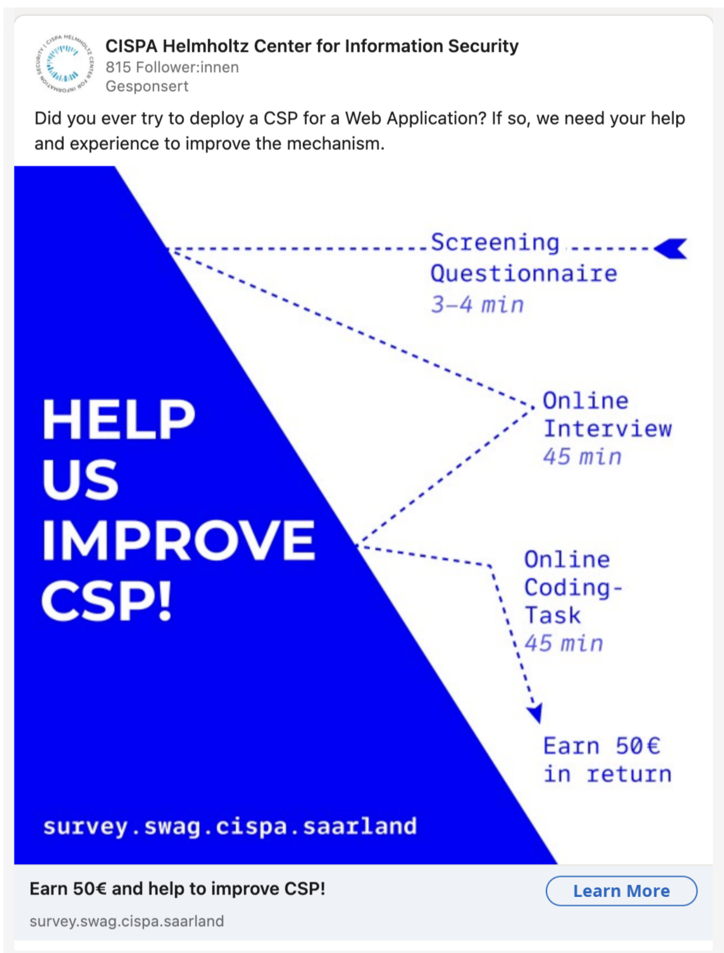

Figure 5.2: Advertisement that was used in our LinkedIn campaign.

tion C.3) that we sent to the set of contact email addresses extracted from the WHOIS entries of Alexa Top 10,000 sites that are using CSP. These invitation emails were sent from an institutional email account to ensure that the possible participants see this as a reliable offer. We also tried direct recruiting people by calling Web development companies via phone. Here we used an official phone number that is associated with our institution in order to increase trust. In general, we can confirm that the recruitment of a specialized population, such as Web developers, who are knowledgeable in CSP, is very challenging [74].

5.2.2 Screening Survey

Our screening survey is a self-built and self-hosted Web application, including a sane CSP. The self-hosting under a subdomain of our research institution provides us exclusive and full control over the data entered into the survey. The survey itself consists of 16 questions, with the time required to answer them no more than three minutes. The landing page of the survey also reminds the participant about the number of questions

and the time required to answers all questions (see Figure 5.3). In addition to that, they are informed that all answers are optional (except for the contact email), as well as the fact that the collected data will be used for scientific purposes only. First, we ask questions about security decisions within their working environment. Notably, we also ask for which Web application the participant takes part in the deployment or maintenance of CSP, asking them to supply a URL if possible (see Section C.1.1). In case of inviting a participant to our interview, we use this link to the Web application to customize the semi-structured interview. We either focus on the root cause of certain insecure expressions used in their CSP, or we ask the participant how they arrived at their secure policy and what roadblocks occurred during that process. Further, we ask questions regarding technology perception and security awareness when using Web applications (see Section C.1.2). Last, we ask for demographics covering age, gender, profession, education, home country, and the company size [39] (see Section C.1.3).

Hi there!

Welcome to our short **screening questionnaire**, covering basic information about your **demographics and your familiarity with CSP**. The survey takes approximately 3-4 minutes and consists of 16 questions.

Irrespective of whether you are selected for the follow-up interview, **all data will be treated confidentially**. At no point will we disclose identities of participants (neither in public nor in our research papers) nor the sites they operate.

Who are we?

We are a group of **security researchers from the CISPA** Helmholtz Center for Information Security located in Saarbrücken, Germany. In previous work we found that the creation of a sound Content Security Policy (CSP) is a challenge many Web sites are struggling with. This motivates us to uncover the roadblocks of CSP with a user study, and ultimately facilitate the development process.

Terms and Conditions

Your participation in this research is voluntary. You may decline further participation and abort this survey, at any time. However, you will forfeit the possibility to be invited to the interview by doing so. The collected data will be used for scientific purposes only.

☐ **I agree with the Terms and Conditions.**

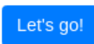

Figure 5.3: The landing page of our screening questionnaire.

As soon as a candidate completes the questionnaire, assess if they are suitable and send an invitation email. In doing so, we were able to manually filtered out bots and pay particular attention to prior experience with CSP. The invitation email (see Section C.4) contained the following information: (1) a reminder about the study compensation of 50 Euro (as an Amazon gift card) and the fact that the study will be recorded; (2) a detailed description of the study procedure including a schedule; (3) detailed choices about the coding task, as well as video conferencing systems. In our study, we pay special attention to making the participants feel as comfortable as possible and to replicating their usual programming environment as closely as possible. Therefore, we offer our participants the greatest possible freedom concerning the configuration of the

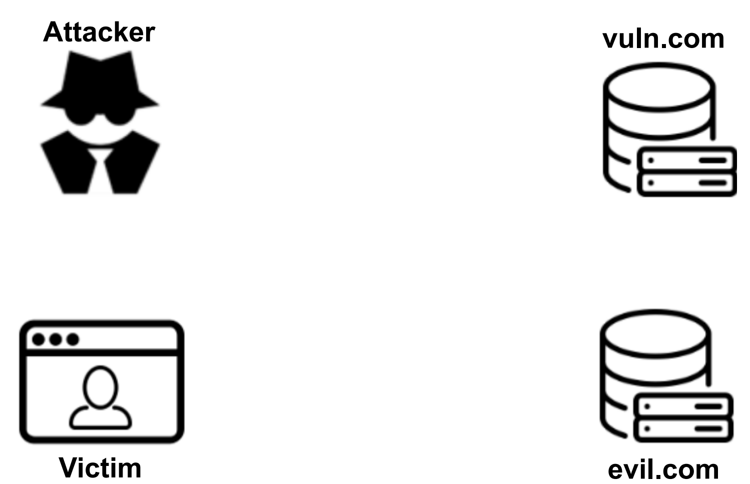

Figure 5.4: Template for XSS/CSP Drawing Task

coding task, choice of video conferencing software, and choice of the study date [1].

5.2.3 Interview

The interview consists of three blocks: **(1) General questions** on the participant's work in the company and educational background. These easy-to-answer questions provide an introduction to the interview and also act as a warm-up. However, the questions are designed in a way such that the participants might also give us valuable information regarding their mindset about CSP, and how they first got in contact with the mechanism. **(2) Threat Model covered by CSP:** This block covers attacker capabilities, different use cases for CSP, as well as the process of decision-making that leads to a (secure) CSP. Those questions are supplemented by a drawing task in which participants draw and explain a XSS attack of their choice in detail. For this purpose, we provide the participants with a graphic containing four stakeholders: The attacker, the victim, the vulnerable server, and an attacker-controlled server (see Figure 5.4). The drawing is done either via Zoom's Annotate feature or alternatively with enabled screen sharing and diagrams.net [2]. After finishing the drawing, we ask the participants at which point of this attack a CSP could successfully stop exploitation.

While Roth et al. [P1] already hinted that XSS mitigation is the prominent use-case for CSP deployment, this block of our interview enables us to get a deeper understanding of the underlying threat model that developers have in mind when dealing with CSP. Furthermore, it allows us to see which of the different use-cases of CSP (XSS mitigation, framing control, TLS enforcement) is the most prominent and how knowledgeable each participant is about CSP's full capabilities. The third part of the questions regarding **(3) Roadblocks for CSP** is customized towards the participant. In the screening questionnaire, we ask the participants for (a list of) domains for which they took part in the deployment or maintenance of a CSP. If provided with a URL, we then rely on

[1] `https://calendly.com/`
[2] `https://www.diagrams.net/`

the live and Internet Archive version of the said domain to determine if this CSP is secure, insecure (i.e., trivially bypassable), or the company gave up on CSP by eventually dropping the header. While the Internet Archive's crawl, such as crawls in general, might not fully reliable due to e.g. bot detection, prior work [P1] has shown that the validity of the Internet Archive regarding CSP headers. Also, during the interview the CSP that was or is deployed on those pages was lateron confirmed by the participants. Depending on which of the three groups the study participant falls into, we ask different questions. In essence, we ask participants that have worked on policies without any insecure practices, such as the unsafe keywords, how they managed to achieve this and which roadblocks they faced during this achievement. In contrast, we ask those with a trivially bypassable policy what caused the usage of certain insecure practices in their CSP. Finally, for those who gave up on CSP, we ask why they aborted their experiment of deploying a CSP and what changes would be required before they would consider attempting CSP deployment again. During this block of questions, we not only want to hear about their stories of CSP deployment, but we also ask about tools and consulting that helped them during this process. The complete interview guideline can be found in Appendix C.2.

5.2.4 Coding Task

In the coding task, we ask each participant to create a CSP that mitigates the effect of XSS attacks for a small Web application. We inform the participants that they can use any resources throughout the task to make sure that we are simulating their usual programming behavior as closely as possible. During the procedure, the participants are asked to share the screen such that we can observe their coding behavior during the experiment. Furthermore, we ask them to think aloud during the task, such that we not only see what they are doing but also understand the decisions made during development. The task is available in three different Web backend languages. Here we looked up the most popular programming languages in 2020 and checked if there is a popular Web framework in that language. As a result, we created the same Web application in JavaScript (Express), Python (Django), and PHP (Twig). In all cases, we build the functionality in a very similar way, e.g., by using the Jinja2 templating language for the HTML files in all three cases. By doing so, we make sure that the choice of the programming language is not interfering with the complexity of the task. After choosing their preferred language, the participants download the corresponding source code (<2MB) and ideally use `docker-compose` to start the application. However, if they do not want to use docker, we also provide alternatives like an install script for Linux-based systems, which also includes the Windows Subsystem for Linux (WSL), or downloading and importing a preconfigured virtual machine for virtual box. In addition to that, we offer the opportunity to remote control this VM using Zoom, or alternatively use TeamViewer [3]. The source code changes of the Web application can then be done in the participant's favorite IDE, such that they feel as comfortable as possible.

As laid out by several prior works [19, 136, 137] and more explicitly by Roth et

[3] `https://www.teamviewer.com/`

al. [P1] and Steffens et al. [122], deploying CSP is made harder if certain constructs are used in a Web application, in particular ever-changing third-party inclusions, inline scripts, and inline event handlers. Since we aim to understand how developers understand these roadblocks and find solutions, we build our application *with* such roadblocks in place. The goal is to investigate if, how, and why the participant resolves these issues. In most cases, multiple solutions are possible, broadening the exploration space. If the participant, for example, uses nonces to fix the inline JS, it is interesting to see how the nonces are generated, as well as how they are placed on the script tags. Overall, the goal of this coding task is to enrich the data we got from the interview with real hands-on coding experiences with CSP. By conducting this coding task, participants can also recall roadblocks and challenges that they forgot to mention during the interview such that the data that we gather is as complete as possible. In essence the Web application is a small app to log in and save notes, specifically designed to carry the CSP-inhibiting patterns mentioned above. All three versions of the Web application are available at the CISPA GitHub Repository [4].

5.2.5 Pre-Study

We conducted a pre-study to ensure that our interview guideline is appropriate to provide answers to our research questions, to identify errors and inconsistencies in the coding task and its setup, and, to give the interviewer the opportunity to practice. The pre-study consisted of two steps. First, we had the interview and coding task tested separately, each by a person knowledgeable in the respective area. A researcher created a persona representative of our study population and took the role of this persona for the first interview test. Another researcher familiar with CSP tested the coding task. In the second step, we conducted three complete runs of interviews followed by the coding task. We recruited two students and one web developer who had experience with CSP. In doing so, we made sure to cover each programming language once. The results of the pre-study are not included in the results of the main study. Based on the pre-study, we slightly changed the order of the questions, among other things, to ensure a more natural interview flow. We also included code snippets in the coding task that could be used as templates to programmatically add events. This was to ensure that the participants had enough time to focus on creating the CSP, as searching for the right syntax in the pre-study took a lot of time.

5.2.6 Data Analysis

We manually transcribed all collected data, including the coding and drawing task. Afterward, we unitized [23] the transcript and conducted open coding according to Strauss and Corbin [128] to analyze the data. For the analysis of the drawing and coding task, we additionally used the screen recordings to ensure no information is missed. In total two coders (the core authors) were involved in the coding process and construction of the codebook. The first coder constructed an initial version of the codebook, taking into account two interview transcripts. Based on the initial

[4]https://github.com/cispa/12-angry-developers-web-applications

codebook, both coders coded all interviews, resolving issues and adjusting the codebook accordingly after each iteration. We continued with the coding procedure until both coders agreed that saturation was reached. In our case, this meant no new concepts emerged from the newest two interviews. We calculated the intercoder reliability of the different codebook versions before resolving issues. With the saturated version of the codebook, we re-coded all interviews to ensure that no information was missed during the initial coding. The final codebook is attached in Section C.7.

Codes are partitioned into high-level *primary* codes and more detailed *secondary* descriptions. For example: if a participant complained during the interview about the false positives in CSP's report feature, the corresponding *primary* code is "Roadblock" and the *secondary* code is "False Positive Reports". Using this way of assigning codes, we made sure that we can better evaluate which roadblocks occurred, which strategies were used, and which motivations and perceptions the participant had during working with CSP. We applied *thematic analysis* [14] to the coded data to identify emerging themes and patterns. Then we conducted *axial coding* [128] to investigate the relationship between themes. To this end, we investigated co-occurrences of codes. For example, we analyzed links between strategies and roadblocks and explored the origin of misconceptions. We then used the combined results to identify strong factors which lead to success or failure of deploying a sane CSP.

5.2.7 Ethical Considerations

We carefully considered risks and benefits for participants when developing the study design, especially as some data collection methods may be perceived as invasive. Especially the installation script in case of the direct deployment of our coding task on the participant's machine may be perceived as invasive. However, we wanted to be as close as possible to the participant's normal coding behavior, so we decided to offer the docker file and the direct execution option. Notably, every participant had the free choice of using the provided VM or remote access. We informed all the study participants about the screen and audio recording before and during the online interview. All participants gave their electronic (pre-questionnaire) and verbal (beginning of the interview) consent to data collection and processing. This data is processed and stored in compliance with the General Data Protection Regulation (GDPR). In addition to that, the study methodology and data collection processes have been approved by our institution's ERB.

5.3 Results

In this section, we present participant demographics and results of our *thematic analysis*. The *inter-coder reliability* Krippendorff's α [70] was between 0.71 and 0.92 for each version of the codebook. Our results shed light on why people decide to deploy a CSP and how they perceive the mechanism. Additionally, we examine roadblocks for a secure deployment of a CSP, as well as the types of strategies used during this procedure. Here, we combine findings from the interview, which give us real-world insights into the work environment of our participants, with results from the coding task, which

reveal concrete technical and conceptual problems in creating a CSP. We support our findings with participant quotes (translated verbatim into English where necessary).

5.3.1 Participant Demographics

Our study population includes both male and female participants. Their age ranges from 20 to 50 years (survey captures ten-year ranges). The participants' employers range from small and medium-sized enterprises (<9 people) to big companies with over 250 employees. While seven participants stated that the Web presence is the main business of their company, this was not the case for three of them, and one participant did not provide an answer to this question. The detailed demographic of the 11 participants that completed the survey is depicted in Section C.5. The majority of participants (eight) are located in Germany, but we also had participants from Czechia, Ireland, and the United Kingdom. Their education level was divided into three master's degrees, three people with bachelor degrees, two with a software developer apprenticeship, one who only specified the education level as secondary degree, one that entered "degree" into the text field, and one that did not answer this question. For their current occupation, all of our participants entered different Job titles: Software Security Engineer, Freelancer / Developer, Software Development Manager, Analytics & Business Intelligence, Founder, Ph.D. Student, Software Developer, Student, AppSec Specialist, Web Developer, and a Student Jobber. Note that all participants had prior experience with CSP and (have) worked for or founded a Web company. The last involvement in the maintenance or deployment of CSP was still ongoing for three participants, in the last week for one participant, within the last month for four, and more than a year ago for three participants. Throughout the interview, we also asked the participants if they have an IT security background. While six participants stated that they had courses that targeted security, or especially Web security, in their free time or during their studies, the other half of our participants noted that they had no security background but self-taught knowledge about Web security.

5.3.2 Motivation for CSP

When asked about their motivation to deploy a CSP, participants referred either to threat models associated with CSP or external factors such as financial implications. Figure 5.5 provides an overview of the concepts. Most prominently, participants explicitly mentioned XSS mitigation (five), as well as pen-tests and consulting (four) as their primary motivation.

Among others, external factors include penetration tests, consulting, or build-pipeline warnings complaining about a missing CSP. One participant also mentioned that he attended a security training that suggested deploying a CSP to protect the application. Also, the effect of a missing CSP on the company's reputation is one reason why companies decide to use CSP. Concurrent to this, big or known companies see themselves as role-model for others and therefore should include a CSP. Seven participants perceived CSP as an additional security layer that kicks in if all other measures, such as secure coding practices, fail. Notably, seven participants also mentioned positive side-effects that they discovered during their CSP journey. For example, during the

External Motivations

- Pentest / Consulting
- Additional Security Layer
- Reputation
- Role Model
- Security Training
- Build Pipeline Warning
- Financial Implications

Attack Mitigation

- XSS Mitigation
- Resource Control
- Framing Control
- TLS Enforcement
- Data Connection Control

Figure 5.5: Categories of Motivations

deployment process, they re-evaluate the resources used by the application and whether they are still required for the application's functionality. Also, the re-evaluation of the application structure was something that the participants mentioned as a benefit during CSP deployment.

5.3.2.1 Threat Models governed by CSP

In addition to the threat model-based motivations to deploy a CSP, the participants also mentioned their perceptions of CSP. We also talked with the participants about the different capabilities of CSP and discovered knowledge gaps regarding CSP.

XSS Mitigation: Five participants explicitly stated that the primary motivation for using a CSP was to mitigate XSS, making this the most prominent motivation. Each of them perceived the CSP as an additional security mechanism stating.

> *"I started learning about XSS, and then I became interested in the solutions to XSS. And of course, you know, you start on the road of input sanitization, output in coding and then eventually, CSP also, you know, becomes a factor."*
>
> – Participant 5

The perception that CSP can mitigate the effect of XSS attacks was present in ten interviews. Eight participants knew how CSP could mitigate the effect of XSS. However, three participants had misconceptions on how and where CSP kicks in. For example, they thought that CSP only forbids the connections to the attacker's server or that CSP prevents the attacker from injecting a malicious payload to the server-side in case of a stored XSS. Closely related to the XSS mitigation use-case of CSP is the participants' perception that CSP can be used to control the resources included.

Also, five participants mentioned that CSP enables the developer to have fine-grained control about data connections of the Web application. Here, participants specifically mentioned the *connect-src* and *form-action* directives. One participant had the attack scenario of sensitive data exfiltration in mind. Notably, the fact that one can simply redirect the browser to exfiltrate data because of missing support for the *navigate-to* directive was not mentioned by any participant.
During the drawing task, we identified that one of the participants had the misconception that CSP is capable of defending against Cross-Site Request Forgery (CSRF) attacks. Such a misconception might lead to a scenario where certain defense techniques, like CSRF tokens, are not used because the developer thinks of CSP as the "holy grail" of Web security defends against everything.

Framing Control Another known feature of CSP is the defense against Click-Jacking attacks [89]. For one participant, this was the main use case of CSP. One participant admits that he only knew about this capability of CSP because information sources for the old and deprecated *X-Frame-Options* header suggest using CSP's *frame-ancestors*. However, two of our participants argued that they do not use *frame-ancestors*, because the *X-Frame-Options* header is doing the same while having better support:

> *"For this, we have something else. The old XFO is, to my understanding, still well supported, so we thought of using CSP for that. But the old blunt method is working for us anyway. So, it did not add extra protection."*
>
> – Participant 3

Notably, XFO and CSPs *frame-ancestors* are not only different functionality-wise, but also XFO is no fully supported by every browser, which can lead to inconsistencies [P2]. If, for example, a Web sites operator deploys XFO in the *ALLOW-FROM* mode, some browsers are ignoring the entry because this mode is not well supported. In addition to that, even if this mode works, it is not possible to allow multiple hosts to frame a page if XFO is used.

TLS Enforcement The capability of CSP to enforce secure network connections was the least known one. While talking about this use-case of CSP, eight participants mentioned that they are using the HTTP Strict Transport Security (HSTS) mechanism to defend against the underlying threat model of a man-in-the-middle. However, five out of those did not know about CSP's capabilities to block mixed content or upgrade insecure requests. Notably, those features of CSP might become less relevant nowadays because Chrome is disallowing any type of mixed content [28] and Firefox is automatically upgrading HTTP connections [88]. Nevertheless, not all browsers behave the same way, and thus, the lack of knowledge about this easy-to-use feature of CSP makes HTTPS adoption harder for the development team because they need to take care of HTTP URLs that are present in their application.

Key Takeaways: (1) The motivation to deploy CSP is, in the best case, the incentive to mitigate XSS; in the worst case, it is only a checkbox that arose from a penetration test. (2) External factors, like the company's reputation or serving as a role model, such that more Web sites use CSP, can be a motivation to deploy CSP.

5.3.3 Roadblocks of CSP

Throughout the analysis of our dataset, we identified different problems that hindered the deployment of a sane CSP without using any insecure practices. For the thematic analysis of those, we combined different codes and clustered them into four different categories of roadblocks shown in Figure 5.6. Notably, these roadblocks are results from both the semi-structured interview and the coding task.

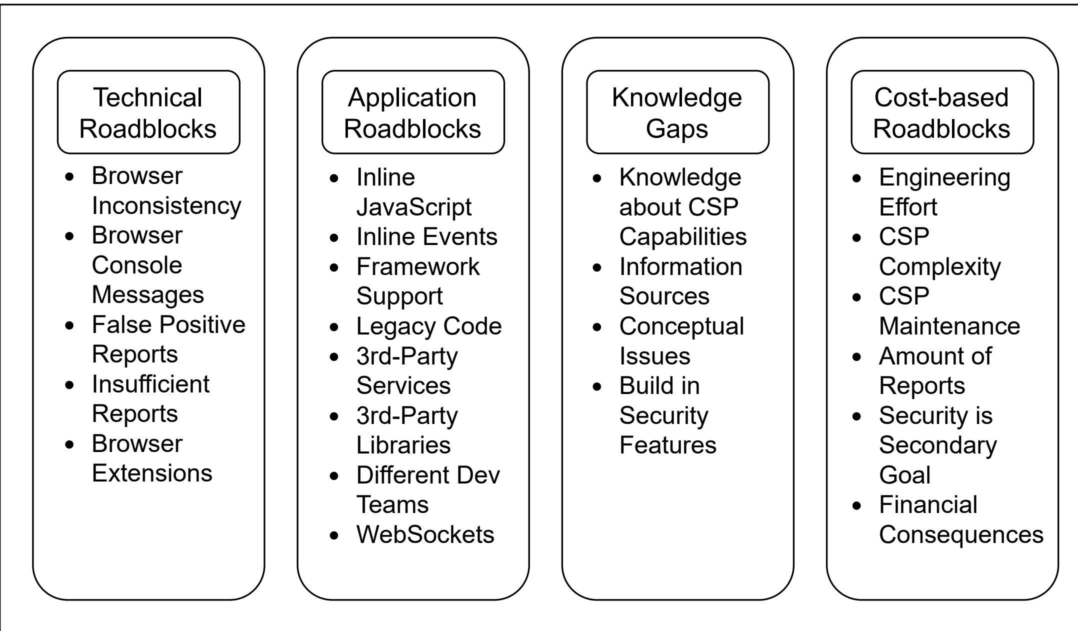

Figure 5.6: Categories of Roadblocks

The *technical* roadblocks are technological problems that the Web applications operator has no control of. In contrast, the *application*-based roadblocks describe problems that occur due to the choices that the developer made when creating the application. The *Cost-based* roadblocks are monetary limitations set by the administrative level of the company or time constraints of the Web applications operator. Finally, the category of *Knowledge Gaps* includes insufficient or bad information sources, lack of documentation, and knowledge about the concepts and capabilities of CSP.

5.3.3.1 Technical Roadblocks

One of the *technical* roadblocks mentioned by eight of our participants is the inconsistent browser support for certain CSP features. While, for example, most browsers support the *'strict-dynamic'* source expression, Safari has not implemented this feature yet, which means that the CSP is too strict in this case and might block important features of the Web application. Also, seven participants complained about the inconsistent way how console messages are designed depending on the browser. The preinstalled browser of our VM was Firefox, and participants that use a chromium-based browser in their usual workflow missed helpful information in the console error messages of Firefox.

"Wait, there is no hash in there."
– Participant 4

Also, browser extensions on the client-side might cause problems with the CSP. Some of them tend to inject their own code into the Web Application, which will result in false-positive CSP errors which might be reported to the development team. But not only extensions cause those errors, but also browser features, plugins, or other client-side interactions with the Web application can generate those reports. From the five participants that denounce those false-positive reports, four also complained about the level of detail in the reports send by CSP violation events. It was mentioned that a detailed code location of the violation, as well as a corresponding hash to allow the code snipped, would ease the deployment and maintenance of a CSP.

5.3.3.2 Application Roadblocks

Some of the application-based roadblocks are those that were hypothesized by previous work on CSP [P1]. Through the interview and especially in the coding task, participants noted the usage of inline JavaScript (8) or inline event handlers (8). Notably, the participants already mentioned the inline scripts in the interview part, while the problem with event handlers was mainly identified during the coding task. One critical point was that the strategy to resolve the issue with inline scripts could not be applied to the inline event problem.

"I'd like to insist on attaching such a nonce here. But I just tested. It does not accept a nonce."
– Participant 12

If, for example, participants use *hashes* or *nonces* for inline scripts, they were not able to apply the same technologies to allow inline events. Adding nonces to non-script tags is a no-op, and allowing event-handlers through their hashes requires `'unsafe-hashes'` in Chrome and its derivatives. Another application-based problem is the usage of third-party code. During the interview, four participants complained that even if their code is fully CSP-compliant, third-party services like advertisements and

third-party libraries such as Angular require them to use a more lax CSP, which might be bypassable [122]. A similar limitation is introduced by choice of the framework which is used to create the Web application. Six participants explained that the choice of the framework could also block a successful CSP deployment. Either the framework supports CSP or not, and even if the framework itself is compliant with CSP, its plugins might not adhere to that. Another problem that one participant mentioned is the way how WebSockets are handled in CSP. While some browsers allow WebSocket connections to the domain itself if *'self'* is present, as specified since CSP level 3 [139], others require to allow the own domain with the WebSocket protocol explicitly. This issue was already discussed in the CSP GitHub repository [144], leading to a change in the living standard [139]. According to five of our participants, the presence of legacy code in a Web application also causes problems during CSP deployment. Usually, the development teams first build the application and later want to add a CSP to it. Thus, the application is full of inline codes, inline events, non-CSP compliant libraries, which makes creating a sane CSP hard and very costly.

5.3.3.3 Knowledge Gaps

During the interview, we also discovered that information sources and online tools might be counteracting the deployment of a sane CSP. Some online security scanners only check if a CSP header is deployed, but such tools often do not check if the deployed policy is trivially bypassable or missing important directives. Moreover, we identified five cases where the used information source about CSP is misleading for the participant or gives wrong information about CSP. Those sources are not designed to give people the *most secure* solution for allowing sources (e.g., the usage of *nonces*) but rather suggest allowing the third-party domain, which is even worse than suggesting to use full URLs. Also, those sources mislead the reader in case of inline event handler. Instead of suggesting to add the events programmatically, they first suggest the usage of hashes which, however, leads to inconsistent behavior among major browsers. One result of such misguided information available for CSP is that developers have conceptual issues with CSP. Through the interview part, we probed participants about the different capabilities of CSP. While the initial use case of CSP is known to most of the developers, other use-cases like framing control and TLS enforcement are less present in their mind. Notably, a participant mentioned that usually, security headers are built-in and deployed per default in some frameworks and are wondering why this is not the case for CSP.

5.3.3.4 Cost-based Roadblocks

A reason for the lack of a CSP or for the usage of insecure practices is, according to ten of our participants, not only the lack of knowledge about CSP or its capabilities but, in many cases, time or monetary reasons. CSP is seen as a rather complex security mechanism that requires massive engineering efforts, which makes it costly for a company to deploy. Also, half of our participants admit that security is often seen as a secondary goal during the development of a Web application, which is why non-CSP compliant technologies are used in the development process. Thus, if during the initial deploy-

ment of a CSP, for example, legacy code is present, the decision to use *unsafe-inline* is taken instead of costly actions such as refactoring the application. But not only the initial deployment of a sane CSP might be costly, but also the maintenance of CSP can be quite hard, as four participants pointed out. For every new content or feature that is added to the application, new entries might need to be added to the policy, requiring constant changes to the allowlist. Another problem that makes the curation of a CSP harder is related to the false-positive reports mentioned in Section 5.3.3.1. Four participants explained that depending on the number of clients that are visiting the Web application, the reporting endpoint is flooded with both meaningful and false-positive reports, which requires massive effort to distinguish them from another and might, in worst-case, result in a denial-of-service for the machine running the reporting endpoint.

> *"They accidentally DDoSed their endpoint because they had a policy misconfiguration which generated, like, hundreds of errors per page."*
> – Participant 5

Key Takeaways: (1) Application-based roadblocks such as third-party services or libraries, inline scripts, and inline events hamper the deployment process. (2) Also, technological limitations like browser inconsistencies and missing framework support, knowledge gaps introduced by misleading information sources, or cost-effectiveness considerations are blocking factors for CSP adaption. (3) The false-positive reports and the number of reported violation events further complicate the maintenance of a CSP.

5.3.4 Deployment Strategies

During the interviews, we observed different strategies for the initial deployment of a CSP, principles of how to deploy and maintain a CSP, as well as strategies to solve certain problems during deployment. Those problems include inline JavaScript code, inline events, and the handling of third-party scripts. While the interview mainly shed light on strategies for the initial deployment and the deployment principles, the coding task showed detailed strategies into how to solve problems regarding inline code, events, and third-parties. Figure 5.7 gives an overview about the strategies that the participants mentioned during the interview or strategies they have taken into account for the coding task.

5.3.4.1 Initial Deployment

Nine participants claim that they tend to start with a rather restrictive policy to end up with a sane one. The resulting error messages, e.g., in the browser's developer console, can be used to identify the fragments in the code that are blocked by the CSP. Six of the participants that choose this way used the restrictive policy in the enforcement mode, while three tend to use the restrictive policy in the report-only mode. The remaining

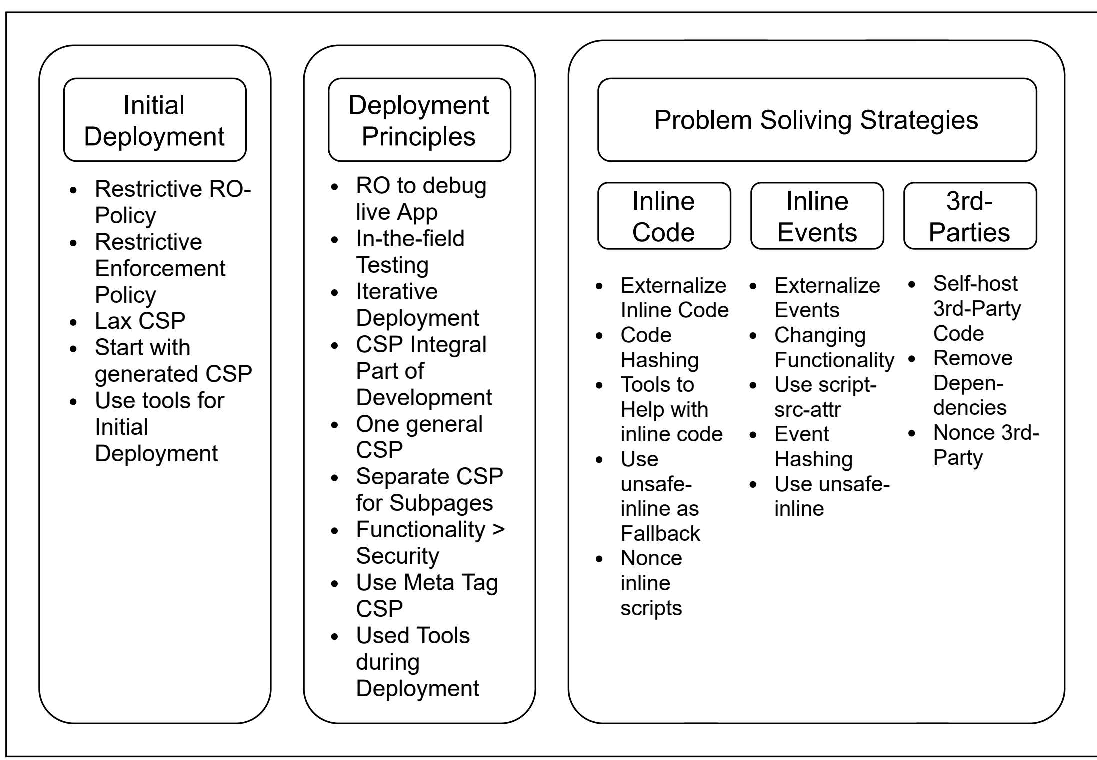

Figure 5.7: Categories of Strategies

three participants are, however, starting with a rather lax CSP and improve this one until they arrive at a secure policy without insecure practices such as *unsafe-inline*. By choosing this path, the participants were not flooded with error messages but were able to solve one problem after another. Four of the participants used an automatically generated CSP as starting point for their CSP deployment. To do so, they used tools like CSPer.io, the Mozilla CSP Laboratory, or the report.URI wizard.

> *"Yes because I would then really install this Firefox add-on if that's okay. That always gives me the fastest baseline where I can then build on it."*
> – Participant 10

Those tools, however, pushed the participants into allowing third parties via their domain or the full URL rather than using nonces. In addition to that, none of the tools solved the problems regarding inline codes and events. They either allowed the execution via the *unsafe-inline* source-expression, which resulted in a trivially bypassable policy, or they blocked it, which results in a loss of functionality.

5.3.4.2 Deployment Principles

Throughout the interview, the participant shed light on different deployment principles for CSP. As it was initially thought of by the team that is curating the CSP standard, four developers use the *report-only* mode to debug the CSP they created in the live environment, such that they do not destroy any functionality during that experiment. Not only for the initial deployment but also for general CSP maintenance, the process in how people managed to get to a sane CSP involved an iterative deployment cycle where they started with a rather lax CSP and, over the course of time, strengthen their policy. While the before mentioned principles are finding their use-cases in scenarios where the developer thought about CSP after creating the Web application, two participants mentioned CSP as an integral part of their development, which drastically reduced the amount of application roadblock occurring during this process. Six of the developers mentioned using one general CSP for the whole application, while two others thought about using separate CSPs for every subpage, and three evaluated both of those principles. To actually deploy the CSP, our participants not only used HTTP headers but two of them used HTML meta tags instead to deploy their CSP. Features like *frame-ancestors* which defends against clickjacking attacks or the definition of a *report-uri* to easier maintain the policy can, for security reasons, not be used in a CSP defined inside the HTML structure. Thus, a developer should prefer to deploy CSP via HTTP header rather than a deployment inside the HTML. One essential part of the deployment procedure is the usage of tools. To ease the maintenance of CSP, two participants used a CSP Preprocessor. To know if the created CSP is actually secure, tools for CSP evaluation such as Google's CSP Evaluator are used by nine of the participants. Also, tools for the evaluation of the CSP violation reports were used by six participants. Three participants claimed that the functionality of the Web service is usually more important than the security for many companies.

> *"But it's very easy to break your site with CSP and the wrong CSP. You're forgetting that you load [...] some payment method on this page and not on every other page. And so many people have problems when the site goes down."*
> – Participant 3

5.3.4.3 Problem Solving Strategies

In order to solve some of the application issues mentioned in Section 5.3.3.2, the participants had several different strategies.

Inline JavaScript: to allow JavaScript code to present in inline script tags, seven of the participants decided to move the inline code to a self-hosted external script. Therefore, the script then can be added to the allow-list using the *'self'* source-expression. One participant used hashes of the code in order to allow it, and because he used a chromium-based browser, the developer console was used to generate the code hashes. Three participants allowed the execution of inline scripts using nonces,

while one of them also mentioned the possibility to use unsafe-inline as a fallback for non-CSP Level 2 compliant browsers.

Inline Event Handlers: another problem, that especially occured during the programming task, was the presence of inline event handlers. Similar to the inline code problem, the participants decided to programmatically add those events to the HTML elements and allow these code snippets using the aforementioned technique. While four participants thought about using hashes to allow the events, the required presence of the *unsafe-hashes* expression discouraged them from doing so. Notably, one participant tried to use this source expression together with hashes to allow the inline events. This participant used this inside the *script-src-attr* expression, which is one of CSP's newest features that only apply to JavaScript defined within HTML attributes. However, because this directive was just added to the standard, it has not been implemented by all major browsers yet, which results in a loss of functionality or security depending on the CSP. One of the participants also came to the conclusion that using the *unsafe-inline* keyword is the only way to allow events in CSP.

Third Parties: There are several different solutions to allow a third-party resource in a CSP. Ten participants allowed the whole domain of a third party, mainly due to bad information sources or due to using an auto-generated CSP as starting point. Only one participant decided to allow the full URL, so only the specific resource. Notably, four participants thought about the possibility to self-host the third-Party code such that it falls under the source-expression *self*. While this would have been easy for the third-party resources we have used (Bootstrap & jQuery), it might get more complicated in case of other third parties that are again loading other assets. One participant also thought about changing the application such that certain third-party resources are not necessary anymore, e.g., by migrating to Bootstrap 5, which works without jQuery. The easiest and most secure solution, to use nonces, was used by none of our participants. However, this lack of knowledge about the capabilities of nonces might originate from the used information sources because highly ranked sources such as `content-security-policy.com`, advertise nonces as a way to allow inline scripts and get rid of *unsafe-inline*, rather than informing people about the fact that they can be used to allow any source [33]. Notably, the information sources also do not mention that hashes will only work for third-party scripts if those scripts allow access to their source code via the Cross-Origin Resource Sharing HTTP header [32]. We have notified the corresponding information sources about their potential for improvement.

Key Takeaways: (1) Both allowing scripts via their domain and using hashes or nonces are used to allow scripts, while usage of the latter is focused on inline code. (2) Developers tend to use one general CSP and tend to start with a strict policy to get more error messages. (3) The usage of tools for generating an initial CSP, evaluating the policy, or analyzing the violation reports seems common for CSP deployment.

5.4 Discussion

In this section, we investigate the relation between certain roadblocks and strategies with axial coding. To this end, we investigated co-occurrences of the *primary* codes *Roadblock* and *Strategy*. In addition to that, we give suggestions to improve CSP as a mechanism and the deployment process of CSP. We also interpret the outcome of the drawing task and discuss the limitations of our work.

5.4.1 Relations of Roadblocks and Strategies

The roadblock regarding missing framework support was often mentioned alongside with the strategy of using nonces for inline scripts (three times), the strategy of using hashes to allow inline events (two times), as well as using *unsafe-inline* to (two times). The participants complained about missing support for hashes or nonces in the frameworks that they are using in their company. This missing support was one reason why participants admit to having included *unsafe-inline* in their policy instead of using nonces or hashes for their inline scripts.

The strategy to use one general CSP for a Web application, as well as the strategy to use separate CSPs for each page, were both mentioned two times together with the roadblock of the CSP maintenance requiring too much effort. The participants argue that using one general CSP for all pages results in a policy with a lot of entries, which makes it complicated to maintain all those values. On the other hand, participants argue that using separate CSP for subpages might result in a vast amount of different CSPs, resulting in a huge effort to maintain all those small policies. As Some et al. [117] showed, having multiple CSPs on a single origin can still expose a site to a successful XSS exploit if one of these policies is bypassable or not even set on certain pages.

In two of our interviews, the participants reasoned about using hashes to allow event handlers. However, due to the incomplete information on how to actually allow them in both Chrome and Firefox, those participants backed away from using hashes. In general, online information sources often lack important information. This not only applies to hash support but lacked important links, e.g., from the explanation of `'unsafe-inline'` to nonces, such that developers know how to handle inline scripts.

Key Takeaways: The strategy of using nonces to allow inline code gets hard to complete if the used framework or its plugins are not CSP compliant.

5.4.1.1 Improvement Suggestions

The information sources used by our participants partly pushed them into implementation paths that caused more work than necessary. Also, the presented information was incomplete and did, in many cases, not recommend the best practices in terms of CSP deployment. In particular, the sources proposed to use *unsafe-inline* to resolve problems with inline scripts. While at least some sources mentioned that this makes the policy trivially bypassable, none of them directly linked to the more secure alternative of using nonces. Similarly, although they all noted events could be added programmat-

ically, some guides first presented hashes as a way to solve the issue and then provided the developer with information about the inconsistent support for that method and the requirement to use *unsafe-hashes*.

In addition to that overwhelming amount of information to solve specific issues, the examples presented as the primary example of a CSP always included entire domains allowed in the policy. This might cause the misconception that nonces cannot be used for third-party resources and ignores the fact that allowing complete URLs is the more secure way of creating a policy. Instead of presenting the CSP Level 1 way of allowing JavaScript in the application, the information sources should emphasize the usage of nonces in CSP. The best practices for CSP deployment, as they are recommended by Google with *strict CSP* [46], are a good and proactive approach to arrive at a secure CSP. However, the missing support in browsers and frameworks for the recommended features might prevent developers from choosing *strict CSP*. Based on the ideas from *strict CSP* and the problems and strategies that we identified throughout the interview and the coding task, we created a decision tree (see Figure 5.8) that can be used by developers to start with CSP deployment. Notably, this tree does not take into account edge cases like self hosted script-gadgets [75, S2] or vulnerable JSONP endpoints [136]. Also, the common practice of third parties to programmatically add scripts [122] is not considered here. However, as a developer, it is possible to check the own resources and/or self propagate the nonces to all programmatically added scripts by hooking JavaScript's `createElement` API.

As mentioned in Section 5.3.4.1 tools can be used to generate a CSP. However, in the case of inline code, they are doing an unsatisfying job. Either they include the `'unsafe-inline'` keyword, which makes the policy trivially bypassable, or they are just blocking inline code, which requires the developer to externalize it. None of the tools emphasize the usage of nonces in the CSP, but they instead allow URLs or even entire domains. Also, those tools take an existing page and build a CSP around it, which is possible, but arguably the wrong way of approaching CSP deployment. A tool that would help the developers from scratch, like a built-in CSP feature in common IDEs, would likely be more useful. Such a tool would not only be capable of warning users as soon as they use inline scripts or events, but it can also help with the computation of hashes or the propagation of nonces for the JavaScript assets. In addition to that, known and widely used frameworks should assist the developers in creating and propagating nonces to all scripts present in the application and should also enforce this behavior for plugins that are available for their platform. Also, many tools that are used to check the security of a deployed CSP or header configurations in general, as mentioned in Section 5.3.4.2, can be built into IDE extensions. However, it would also be handy for developers if warnings about a misconfigured CSP, or security headers in general, would be printed in the developer console because every participant of our study used this browser tool. If the developer would get the help and the information they need by default, e.g., via the browser, it would ease the deployment and maintenance of CSP. By lowering the engineering effort required to deploy a CSP, problems like the monetary factor might have a lower impact on the security of real-world policies. Some browsers already try to give rudimentary help for CSP deployment, like printing the script hashes in the console error messages. By standardizing those warnings and

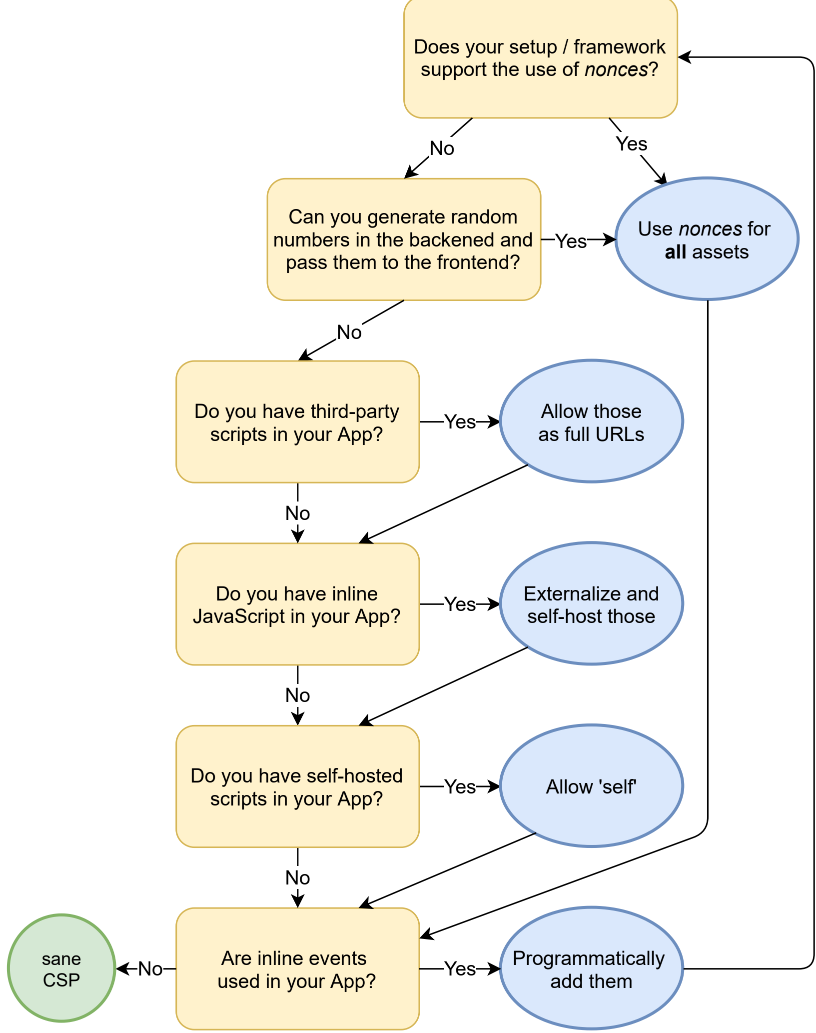

Figure 5.8: Developer decision tree to create a sane CSP.

the error reporting in general, all developers, independent of the browser, can benefit from those messages, as long as everyone is adhering to the standard. This standard compliance, however, seems to be a problem in general. Browser inconsistencies like the different support for a feature such as `'unsafe-hashes'` and `'strict-dynamic'`, or the inconsistent way of handling nonces or WebSocket connections, cause additional confusion for developers in deciding if and how to use these features.

5.4.2 Drawing Task

As mentioned in Section 5.2.3, we ask the participants to draw and explain their favorite XSS vulnerability, so as to better understand the participants' mindset about XSS and CSP. All participants freely chose to draw XSS as a server-side issue. Although one participant at least mentioned "DOM-based XSS", this indicates that the client side of the problem is less prominent in developers' minds. Eight participants decided to draw

a stored server-side XSS vulnerability, one drew a reflected server-side XSS, and two participants explained both variants, which indicates that the stored variant is more prevalent in the developer's mind. The prominence of the server-side might be one reason why two of our participants reveal certain misunderstandings throughout the drawing task. XSS seems to be a server-side issue in their mindset, so the server is *loading* the malicious payload. Similarly, they thought that the server also enforces the CSP. This focus on the server-side was also reported by one of our participants, who is working as a consultant:

> *"The browser decides that in case of CSP and then forbids it [...] but when I then come to cross-origin resource sharing, that this is then the server, [...] and teaching that to my customers is always difficult, too."*
>
> – Participant 9

Another misunderstanding of CSPs capabilities is that three of our participants only explained that if CSP would be present on the attacked page, it would prohibit the data exfiltration to an attacker controller server, but not mentioned that the actual code execution is also forbidden. The full results of our drawing task analysis are available in Section C.6. Notably, one participant did not draw an XSS attack, but rather a CSRF attack.

Key Takeaways: The server-side variants of XSS are far more prominent in the developer's mindset. Two even had the misconceptions that XSS is a server-side problem and concluded that CSP is, therefore, *enforced* on the server.

5.4.3 Reflections on Methodology

Due to the extensive pre-study, we eliminated any application bugs that could have resulted in problems during the programming task. In general, the coding task achieved the intended goal. Participants revealed strategies, information sources, and roadblocks only during the coding task, either because it resulted in additional concepts or shed light on problems that the participant forgot about during the interview but then remembered during the coding task. The way how the coding task was performed revealed different pros and cons. While the remote-control process drastically reduced the time required for the setup, it seemed to be more exhausting for the participants. This is because they needed to control a foreign system, not set up to their liking. Moreover, depending on the network connection, remote control incurred lags. On the other hand, natively running our application requires significant effort to set up, but it also resulted in the participants being able to work as they wish. Conducting the coding task using the provided docker files has shown to be a good tradeoff between the aforementioned options. The setup was not complicated because the docker image only required a few minutes (usually less than three) to build. Because the files were on the participant's machines, it was possible to use their coding setup for code changes. However, one drawback of this approach was that changes to the code required a docker

rebuild, causing a slight delay of few seconds (less than 20). While in theory, we could have also mounted the directory directly into the docker to allow live updates. However, this would require additional packages to allow for auto-reload and incurs the risk of inconsistencies between the developer's view and the docker-run system. Moreover, some participants changed the Web server config through nginx, which would have required a reload in either case.

Similar to the results of the coding task, the drawing task helped us to enrich our data and get a deeper understanding of the participant's mindset about XSS and CSP. Some of the misconceptions regarding CSP and the underlying threat model would not have been uncovered without the drawing task, which is why we recommend this addition to a study like ours. Ten participants used Zoom's annotate feature to draw, which in some cases were not present in the recordings. Fortunately, we made screenshots of the drawings during the interview, and therefore we did not lose any data due to that problem. Using diagrams.net for the drawing task required one or two minutes of setup, but seems to be easier for the participants and was present in both recordings.

Only a small fraction of Web sites actually deploys a CSP. Thus, recruiting Web developers of real-world applications that have dealt with CSP before is even more challenging than recruiting developers in general. Not only is our targeted group limited and hard to reach, in addition to that, they are also educated in both Web development and IT-Security. This high level of education was the reason why we decided to compensate the participation with a 50€ voucher for their time. In total 30 possible participants completed the screening survey. The first recruitment attempt, to send bulk emails to the top sites that deployed CSP, resulted in zero responses. Thus, we continued with the attempt to use the OWASP as a trusted third party to advertise our study. After this campaign 13 potential participants completed our survey. We invited all of those participants to the online interview, but we only received answers from six of them even after two additional reminders. The next idea to increase the number of potential participants was to cold-call Web development companies from our country. This attempt again resulted in zero responses. Thus, we decided to try out recruitment via LinkedIn advertisement, which was seemingly successful with eleven new entries in our database. However, we had indications that ten of those were only bots that entered data in the survey as a result of the LinkedIn advertisement. Those bots not only entered bogus values in the fields of the survey but also supplied emails of randomly concatenated words and numbers. Nevertheless, we send an invitation email to three of those because they entered occupations similar to Web developer but got no response to that email. We also invited one non-bogus entry, but this potential participant did not reach back to us even after the reminders. Right before the advertisement campaign, two colleagues from our research institution gave a talk at an OWASP event. At the end of their talk, they also advertised our study, which resulted in five new database entries, from which four actually took part in the interview. Notably, the OWASP event resulted in one additional participant, who did not take part in the survey but participated in the interview. In addition to that, one participant that completed both survey and interview was recruited via word-of-mouth propaganda. Notably, we did not ask the participants from where they found out about our study. Thus, the num-

bers above are the results of temporal coherence between the recruitment procedure and the participant completing the survey. In total 20 (30-10 bots) people finished the survey, and 12 people participated in our interview, including the drawing and coding task. While we tried several ways to recruit those, we must admit that using a trusted third party, in our case the OWASP, for recruitment surpassed all other recruitment procedures.

5.4.4 Limitations

Our work has three main limitations that are either tied to the recruitment process or the methods we used to gather our data. First, our sample might be biased towards security-aware developers, given our main recruiting path through the OWASP. We invested a lot of time in various recruitment efforts, including contacting web development companies, sending emails to websites with CSP, and advertising on LinkedIn. However, it proved to be very difficult to recruit specialized developers. Yet, as depicted in Section 5.3.1 only half of our participants reported having a security background. Second, we acknowledge the limitations that interviews entail. Although we made every effort to build rapport and ensure that participants freely and willingly recounted their experiences, we cannot guarantee that no concepts were missing or misstated. This portion of the data is based on the recollection of the participants, and it is possible that portions may have been forgotten, intentionally omitted, or misremembered. For this reason, we decided to supplement the interview with the coding task. Nevertheless, we do not claim the completeness of the interview results. Third, the coding task was likely biased due to its artificial setting. To mitigate this, we tried to make the coding task as pleasant as possible for the participants by offering three popular programming languages and various setup options. To avoid that participants feel pressured, we clarified that we do not rate any of their solutions regarding their correctness but are only interested in the process of *how* they approached the problem and developed the CSP.

5.5 Summary

In this chapter, we present the first qualitative study involving 12 real-world Web developers to evaluate the usability of the CSP. Throughout our interview which involved a drawing and coding task, we investigate the participant's mindset regarding XSS and uncover the reason behind the usage of some insecure CSP practices as well as strategies and motivations that interact with the deployment procedure of a CSP. The motivation to deploy CSP is, in the best case, the incentive to mitigate XSS; in the worst case, it is only a checkbox that arose from a penetration test. We shed light on different kinds of roadblocks for CSP deployment. For the roadblocks based on knowledge gaps and conceptual issues, we argue that better information sources can mitigate this problem. However, the technical roadblocks require that the browser vendors finally need to agree upon how CSP should be implemented, how error messages should look like, and introduce warnings and information for poorly configured CSP in the developer console. Those steps would reduce the uncertainty of many developers and reduce the impact of lousy information sources on the deployment procedure to ease

the deployment procedure. Our participants tend to use the old way of allowing scripts via their hostname, but some also use hashes or nonces to allow scripts, while usage of the latter is focused on inline code. Also, tools for generating an initial CSP, evaluating the policy, or analyzing the violation reports seem common for CSP deployment. In addition to that, we also discuss our methodological choices and share the lessons we learned from those, such as the success or failure of certain recruitment techniques. We believe the insights highlight that secure-but-complex technology on the Web leads to confusion and lacking deployment.

CSP is arguably one the most complex mechanisms in Web security, but we hope our insights influence the design process of new mechanisms to avoid such problems. However, introducing possibly breaking changes to an already in-the-wild used mechanism is hard, if not impossible. Also, interviewing developers that are familiar with the mechanism is not working for the design process of new ones. Therefore, chapter 6 takes a look at a new, not widely used mitigation mechanism for client-side XSS called Trusted Types, to see if it is repeating the design mistakes made for CSP (*RQ4*).

6
Is Trusted Types repeating the mistakes made for CSP?

Contributions of the Authors

This Chapter is based on *"Trust Me If You Can – How Usable Is Trusted Types In Practice?* [P5], which is currently under submission. Sebastian Roth conceived the study. Sebastian Roth and Lea Gröber designed the study. Sebastian Roth created the Web applications for the main study. Sebastian Roth and Lea Gröber the Web application for the screening survey. Sebastian Roth and Philipp Baus conducted all interviews (5 P.B.; 7 S.R.). Sebastian Roth coded all interviews and evaluated the results. Philipp Baus and Lea Gröber coded two interviews again to compute the Inter-Coder Agreement. Ben Stock and Katharina Krombholz supervised the study. All authors commented on the text and contributed to individual sections.

6.1 Motivation

Chapter 5 hinted that many developers do not know the differences between client- and server-side XSS, which could lead to problems in the deployment process of client-side mitigation techniques as for example CSP work differently that for example auto-encoding or type safety on the server-side. To transfer the from the server-side known concept of type safety to the client side, Google proposed the Trusted Types API [69]. With this mechanism, it is not possible to invoke dangerous JavaScript APIs without passing the input through sanitization functions that the developers need to specify themselves. However, at the point where the World Wide Web Consortium (W3C) has discussed shipping Trusted Types as First Public Working Draft (FPWD) [81], browser vendors raised concerns that the complexity of self-creating sanitization functions is too difficult for the majority of developers, which is in line with the results from chapter 5 about the (un-)usability of the related CSP mechanism. Wang et al. [134] have exemplified how Trusted Types could be incorporated into a Web framework. However, the claim of the browser vendors lacks a scientific foundation, because the actual usability of Trusted types deployment has not been evaluated yet.

To close this research gap and provide a scientific foundation for decisions of the W3C, the objective of our work is to get insights into each step of the deployment process of Trusted Types and investigate the developer's knowledge about the underlying issue, namely client-side XSS. Therefore we designed a qualitative interview study in a "bottom-up" approach [61] that includes a coding task where the participants deploy Trusted Types for a small Web application. Throughout our study, we evaluate the participant's knowledge of the different dimensions of XSS and their knowledge of existing mitigation techniques. In addition, our iterative study process, allowed us to get fine-grained insights into all deployment steps allowing us to evaluate the entire deployment process of Trusted Types.

Our work first presents a comprehensive qualitative interview study methodology to evaluate the usability of Trusted Types (Section 6.2). With ten participants, we were able to uncover important roadblocks of Trusted Types deployment, such as misconceptions introduced by information sources or problems caused by same-origin iframes (Section 6.3). Based on our findings, we discuss and propose strategies on how to successfully deploy Trusted Types, such as the usage of third-party sanitization libraries.

Also, we present general improvement suggestions for the Trusted Types standard, like the inheritance of Trusted Types sanitizers for same-origin iframes (Section 6.4).

With the improvement suggestions, we want to ease the deployment process of Trusted Types, and influence the W3C decision process. Also, we suggest that the W3C sees this work as a role model to test their working drafts with real-world Web developers before releasing them and before browser vendors start implementing them in their products.

6.2 Methodology

A GitHub discussion [81] of the World Wide Web Consortium (W3C) showed that some browser vendors block the deployment of Trusted Types because they claim it is too complicated for the long tail of Web sites. Therefore, Trusted Types, as a new mechanism, is currently only supported by Chromium-based browsers (since version 83, May 2020) [26]. The goal of this work is to uncover problems with the current version of the mechanism and evaluate how hard or easy it is to deploy, such that the W3C can use our results to improve the mechanism to enable developers to mitigate client-side XSS attacks.

Due to the youth of the mechanism and its limited support in modern browsers, the adoption rate of the mechanism is negligible. Therefore the target for our study is not Web developers with experience with the mechanism, as it is nearly not existent, but rather the average Web developer, that potentially will be using Trusted Types in the future. Together with them, we want to get detailed insights into the deployment process of Trusted Types from a controlled coding task. Such that we can uncover roadblocks, and develop deployment strategies for the mechanism.

Thus, we answer the following research questions:

RQ4.1: Do developers understand client-side XSS and how Trusted Types can mitigate damage?

RQ4.2: What roadblocks do developer face when deploying Trusted Types?

RQ4.3: What strategies do developers follow in case of problems?

6.2.1 Recruitment and Participants

To maximize the ecological validity of our results the target population for our study is real-world Web developers that have experience with the development and deployment of Web sites. Given that we want to study the deployability of Trusted Types for all sorts of Web applications and given that Trusted Types is not widely used by Web sites anyways, we not only considered professional Web developers but also non-professional Web developers, for example, students, as long as they mention prior Web development experience during the screening-survey. Since Web developers are a hard-to-recruit population [P4], we relied on a combination of different recruitment strategies to optimize outreach. First, we posted a recruitment flyer on the social media channel (Twitter) of our institution. This flyer contained details and timelines of the study

You are a Web developer, want to learn something new, and get 50€? We are conducting a study to understand the challenges of deploying a mechanism to defend against client-side XSS.
So, if you are interested, please visit survey.swag.cispa.saarland and/or share this invitation.

Figure 6.1: Recruitment Flyer posted on Twitter

procedure as well as information about us and the compensation for the study (see Figure 6.1). In addition to that multiple researchers from our team presented talks about Web security at multiple industry-focused developer conferences. Those talks included a call for participants for this study, such that we can have direct contact, and immediately answer questions from the Web developers that showed interest. Also, we used contacts of researchers of our institution to industry and ask those people to forward our recruitment Web site to their internal team chat or in their Web development communities. In general, irrespective of the recruitment procedure, we always mentioned that in addition to the monetary compensation of 50€, the participants can learn something about client-side XSS vulnerabilities and how they can mitigate this issue for their Web applications.

6.2.2 Screening Survey

To ensure a homogeneous set of participants that are all security aware and professional real-world Web developers, we conducted a screening questionnaire before inviting selected participants to the actual interview study. This survey included 12 input fields, where ten were actual questions about the demographics of the participant plus a field for the contact email address as well as a feedback/issue field for the survey. Notably, the Web application is self-written and self-hosted under a subdomain of our institution such that we have full and exclusive control over the data entered into the survey.

The landing page of the survey informs the participants about the approximate time required for the survey (1-2 minutes) and the number of questions. Furthermore, they are informed that all data will be treated confidentially and that we at no point will we disclose their identities of them or the sites they operate. The page also includes information on who we are and the terms and conditions of participating in the survey. In the survey itself, the participants are asked to (optionally) provide age, gender, current occupation, highest education, country of living, and the size of the company that they are working at. In addition to that, we also ask questions about their experience in deploying or maintaining Web security mechanisms, their IT security background, the Web presence of their company, and questions regarding the development team. With that information, we are then able to better assess the participant's eligibility for the study. Notably, all of the input fields, except for the contact email, are optional.

After the survey, we manually check the eligibility of the participants, and in case of a positive evaluation invite them to the actual interview via email. The eligibility of the participants is based on their occupation being related to Web development or other answers showing that they have experience as a Web site's operator. The email consists of a short reminder regarding the schedule for the interview process, a link to the interview scheduling system of the Nextcloud instance hosted by our institution, and information about the interview setup. To give the participants a comfortable environment we offer them to use any video conference software that they want to use, as long as it allows for screen sharing for the coding task. We also offer them different ways of setting up the Web application for the coding task. They can either use a docker image provided by us, can directly execute the python Django code, or can remote control the interviewer's machine (e.g. via Zoom or TeamViewer) to conduct the coding task.

6.2.3 Interview

In the first few minutes of our interview, the researcher gives the participant an introduction to the timeline of our interview session. The interview is partitioned into questions regarding the participant's working environment, some questions about Web Security, the coding task, and a debriefing. The question set about the working environment is to check the Security and Web development background of the participant, but also to get the chat between researcher and participant rolling.

After the general questions, we bring the conversation to XSS to answer our research question regarding the understanding of this vulnerability and the differences between client- and server-side XSS. Therefore we explicitly ask our participants if they could

explain to us how the vulnerability happens, and about the differences between client- and server-side XSS. Here, we also try to find out if the participant knows client-side execution sinks for XSS attacks. Next, we chat about Web security mechanisms that come into the mind of the participants when they want to defend against XSS, which then transitions into questions regarding Trusted Types and its method of operation. The question block is topped off with questions about the impression of Trusted Types and if they would use it in one of their Web applications.

Then, the coding task is performed which will in detail be explained in Section 6.2.4. After the coding task, a set of debriefing questions will follow. Based on the recent interactions with the deployment of Trusted Types the impression of the mechanism might have changed, which is why we ask about the feasibility of Trusted Types as a defense mechanism against XSS. The last question that we ask is about improvement suggestions for Trusted Types, to assess which is the biggest problem that currently hinders the successful deployment of Trusted Types. Finally, the participant needs to fill out a System Usability Scale (SUS) [16] to assess the usability of deploying Trusted Types.

6.2.4 Coding Task

In our coding task, we ask the participants to deploy Trusted Types for a small Web application, while the main work in the case of the deployment process is the implementation of the sanitizer functions. The Web application itself uses features, libraries, and frameworks that are frequently used by real-world Web applications, to increase the ecological validity of the data we gathered throughout the coding task. Details of how we chose those, are explained in Section 6.2.4.1. Notably, throughout the coding interviews, the functionality of the Web application has been changed, to ease the initial deployment process such that we get more fine-grained insights into the later deployment steps, but also to lower the level of frustration of the participants because they could get nowhere near enough a proper solution. Due to our usage of real-world third parties for a better ecological validity of our results, we also had changes in the functionality of the application that were not driven by us, but by third parties that change the way their libraries operate during our study. We explain the exact changes of this iterative interview process in detail in Section 6.3.2.

6.2.4.1 Web Application

The Web application's backend is written in Python 3.10 using the Django Framework. For Trusted Types the actual backend language is less important, because of the main work, so the Trusted Types sanitizer is being written in JavaScript anyways. Also as Trusted Types is enforced on the client-side, client-side technologies used by Web sites are more important for our Demo application that the server-side technologies.

To get fresh data about the usage statistics of third-party JavaScript we crawled the Tranco Top 5k sites[1] with a chromium browser instrumented by puppeteer[2]. Our crawl also collected the URLs present on the loaded page and visited those up to two

[1] `https://tranco-list.eu/list/JX83Y`

[2] `https://github.com/puppeteer/puppeteer`

Third-Party Site (eTLD+1)	Inclusions (# Sites)
google-analytics.com	2,626
googletagmanager.com	2,329
gstatic.com	2,311
doubleclick.net	2,162
google.com	2,132
facebook.net	1,632
youtube.com	1,571
googleadservices.com	1,359
googleapis.com	1,233
twitter.com	988

Table 6.1: Top 10 third parties by inclusions

levels of links from the start page. To not put too much load onto the servers, and to speed up the crawling process, we also limited the number of total URLs crawled per site to a maximum of 500 URLs. During a page load, we waited a maximum of 20 seconds for the load event to be fired and then additional 5 seconds before we continue. During this time we capture all script URLs that are loaded.

In Table 6.1 we show the Top 10 sites, by the number of including sites, that were loaded during our crawl. The most requested site by far was `google-analytics.com`, with 2,626 sites that included scripts from it. As the name suggests, the vast majority of those sites (2,541) are inclusions of the scripts to run the Google analytics services on a Web site. Due to this prevalence of the service in real-world Web applications, our demo application also uses Google Analytics.

Advertisements, specifically those from `doubleclick.net`, were also used frequently (on 2,162 sites). However, showing real advertisements to our participants would put costs without benefits on those who ordered the advertisement. These costs would not only be problematic from an ethical point of view but could also make us accountable for ad fraud. Therefore, instead of just ignoring the prevalence of advertisements in real-world applications, we decided to create a custom advertisement service for the Web application.

The biggest player in terms of social media integration was `facebook.net`. The majority of those page loads (742/1,632) were inclusions of the script that includes the Facebook widget. This is why we also included this feature in our application. Notably 336 of the sites that used the Facebook integration also used the Twitter widget at the same time. Due to this intersection, we also incorporated the Twitter widget in our Web application.

Another common third-party service used in Web sites is the integration of Maps. Because the most widely used service here (Google Maps), needed a domain name to properly use all features, but we only used a local setup for our study, we had to use a different service that supported our use case. Therfore, we used `openstreetmap.org` as Maps integration. In addition to that, we also had a page that showed a News blog. To enable users to comment on that news we included the third-party comment system `disqus.com` in our application.

Framework	Usage (# Sites)
jQuery	1,005
Bootstrap	358
Dojo Toolkit	79
Vue.js	75
Angular	68
Backbone.js	19
Ember.js	1

Table 6.2: Usage of JS frameworks

Notably, both `openstreetmap.org` and `disqus.com` have been removed from the application throughout our study such that it takes less time for the participants to deploy Trusted Types. With this decision, we reduced the level of frustration of our participants such that our study is not causing them to refrain from using it in the future. We have chosen those two services to be removed, as they were the least widely used third parties that were included in our application.

To assess which of the client-side frameworks is the most used one we not only checked the script's origin, but also the actual script that is loaded. Table 6.2 shows that jQuery is by far the most used framework with over 1k sites that use it. Notably, the intersection of those sites with those from the second most used framework (Bootstrap) shows that 62.8% (225/358) of the sites that are using Bootstrap are also using jQuery. Thus, we used a combination of Bootstrap and jQuery to build our Web application.

6.2.5 Pre-Study

To ensure that our coding task and its setup do not contain errors and can be solved, such that no participant is frustrated after the interview, we conducted a pre-study involving two participants (P1 & P2). Both worked for an IT company, one as a developer, and the other one as the security incident response team. Notably, both took Web-related courses at university. By selecting participants that are presumably less trained in deploying features for real-world Web sites we want to find a lower bound for the scope of our coding task. Given that none of the participants of the pre-study was close to sufficiently deploying Trusted Types for the Web application we decided to remove some features from the Web site to ease the deployment. We based the decision of which of the third parties to drop on the results presented in Table 6.1. Thus, we removed the subpages that are incorporating third parties for *Maps* integration as well as the one that added a comment section below our *News* page. In addition to that, we also added code snippets in the coding task to speed up the deployment. The pre-study showed that participants copy-paste the CSP header that enables Trusted Types and the skeleton for the sanitizer functions (see Section 2.3.3) from online resources. Thus, by adding them to the source code files, we do not ease the creation of the sanitizer functions, we only reduce the setup time for the actual task. The pre-study has also shown that our interview guideline is sufficient to answer our research questions. From the transcribed interviews we were able to learn more about the developer's mental

model of XSS, especially client-side XSS, and uncover roadblocks as well as strategies for Trusted Types deployment. In addition to that, the interviews also shed light on the participant's perceptions of Trusted Types and motivations and disincentives to deploy Trusted Types.

6.2.6 Data Analysis

For the analysis of the collected data, we used the GDPR-compliant online transcription service `amberscript.com`. Here we used the *Human-made* mode which ensures that the transcriptions are created by professional transcribers and captioners. One of the authors created a codebook using an open coding approach [128]. To better analyze the results of the coding task, and not miss important information, the recordings of the shared screen during the coding task were also considered during the analysis. We continued conducting interviews and analyzing them until no new concepts were added to the codebook from the newest two interviews. After reaching our criteria for saturation of our dataset, the main author handed over three interviews with the saturated version of the codebook (see Section D.1) to two other authors to calculate the inter-coder reliability of our results. Those three interviews were the three interviews with the most different variety of codes, such that we cover as many possible codes as possible with the inter-coder reliability calculation.

To better evaluate the mindset of our participants and to get a better understanding of the occurring roadblocks, and the strategies that the developer chooses, our Codes are segmented into high-level categories and more detailed low-level information. If for example, a participant struggled with Trusted Types deployment due to a programmatically added sourceless iframe, we assigned the code "Roadblock: Same-Origin Iframe", where *Roadblock* is our high-level category and *Same-Origin Iframe* the low-level description. Following the methodology of previous research [99, P4, 116] to identify emerging concepts, patterns, and themes, we analyzed our data using thematic analysis [14]. We then used axial coding [128] to find relations between those concepts and patterns. For example, we analyzed the connection between roadblocks and the applied strategy to explore which factors in the deployment process can lead to success or failure of the deployment of Trusted Types.

6.2.7 Ethical Considerations

To the best of our knowledge, we designed our interview, the coding task, and the data collection process always with the risks and benefits for the participant in mind. The participants had the freedom of choice to use their own machine for the coding task or to remote control the interviewer's machine. While using the own device is closer to the participant's normal and familiar coding behavior, the direct installation of python and Django might be perceived as invasive. However, they could have used the Dockerfiles to create an easy-to-remove docker image that runs our code.

In case the participants used a browser on their own device to access the Web application, the current IP address of the user is leaked to third parties, as the App performs requests to Google Analytics, Twitter, and Facebook. We decided to still use those real-world third parties to maximize the ecological validity of our results.

Also, our participants, especially those from the pre-study, might have felt heavy frustration when they after 90 minutes were not even close to a proper solution. This was one of the main reasons why we cut down some of the features of our application, such that we do not leave behind a long-lasting bad feeling of working with Trusted Types or deploying security features.

We used the online service `amberscript.com` to transcribe the audio of our interview sessions. Notably, we uploaded the audio (not the video) track to only reveal the necessary minimum to the service. Also, Amberscript [3] is fully compliant with GDPR and advertises their service by guaranteeing full confidentiality, transparency, and Non-Disclosure Agreement (NDA) of all the transcribers. In addition to that, all participants gave their electronic consent in the pre-survey and again verbal consent at the beginning of the interview to our data collection and processing methods. Notably, all data (on our side and one Amberscript) is processed and stored in compliance with the General Data Protection Regulation (GDPR) and the entire methodology of our study and data collection processes have been approved by our Ethical Review Board (ERB).

6.3 Results

In this section, we present the results of our analysis, starting with the demographics of our participants. We then elaborate on the participant's mindset regarding the difference between client- and server-side XSS, highlight the different roadblocks that we investigated for Trusted Types deployment, and we shed light on different deployment strategies and their success.

Notably, our additional coding of the three interviews with the most diverse set of codes resulted in an inter-coder reliability Krippendorff α [70] of 0.987 for the three interviews coded with the saturated codebook. Given that our codes are mainly based on technical concepts, which does not leave much room for interpretation, the Krippendorff alpha is close the perfect, even without the discussion of the codings.

6.3.1 Participant Demographics & Background

In total, we had ten participants in our study population. All of them are male and their age ranged from 20 to 50 years. Notably, our survey asked the age in slices of ten, e.g, 30-40, to not be too privacy-invasive.

Four participants are working in an IT Security field, e.g., an Incident Response Team, while two of them also considered themselves as casual Web Developers because they created small Web applications for example for personal use. Six are working as Web Developers, where one of them also mentioned IT Security as one of his work areas. Also, one of the Web Developers is at the same time working in DevOps, which was also mentioned as a working area by another participant.

Regarding security education, five participants mentioned that they took security-related courses during their studies. The other five participants have taken courses or certificates for IT security but reported to have self-taught knowledge about the

[3] `https://www.amberscript.com/en/data-security-and-privacy/`

topic via videos and blog posts. Thus, we reached our target to conduct this study with a homogeneous set of participants that are security aware and familiar with Web development.

During the chat about Trusted Types, five participants stated that they have never heard about the mechanism before, while the others did not explicitly mention knowing (or not knowing) Trusted Types before participating in our study. We also asked the participants about their knowledge of XSS mitigation techniques. Here, seven participants mentioned CSP, eight mentioned proper sanitization and/or filtering of user-controlled input, and one considered Web Application Firewalls (WAF) a mitigation for XSS. Notably, many of the participants mentioned multiple mitigation techniques here, so our participants are knowledgeable when it comes to the mitigation of XSS attacks.

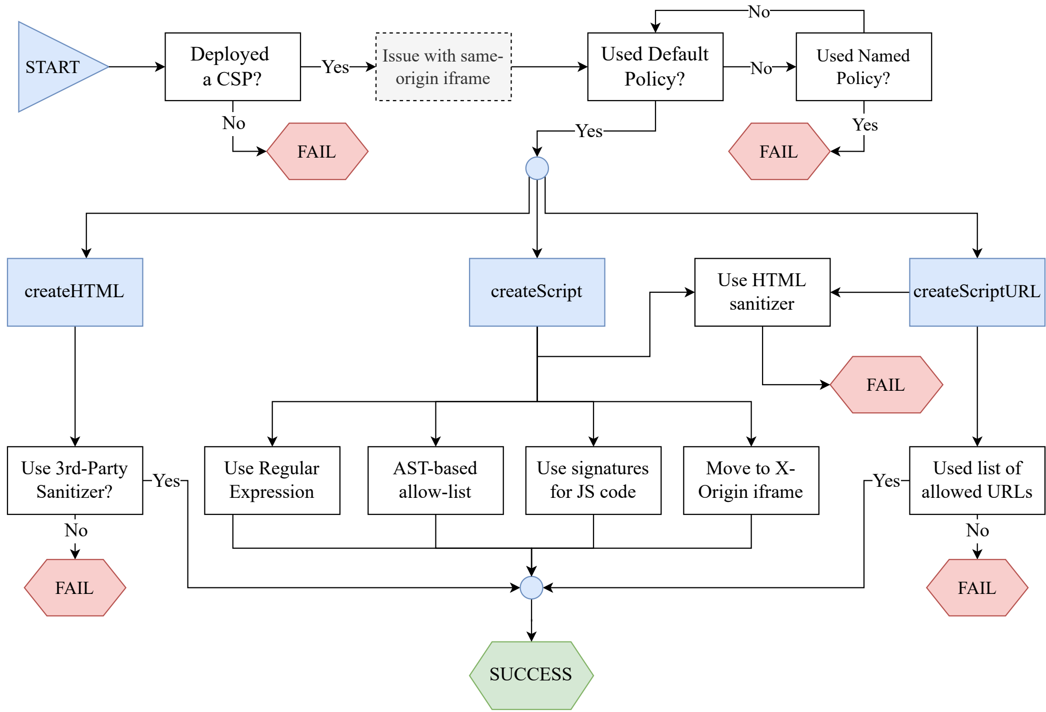

Figure 6.2: Diagram of the deployment workflow based on our participants.

6.3.2 Roadblocks for Trusted Types and Where to Find Them

Based on the approaches that we have seen throughout the coding task, created a workflow diagram Figure 6.2 that we use in this section to shed light on the different roadblocks that we investigated in the different deployment stages. We also explain in detail the changes in the Web application and the reasons for those changes. In addition to that, we will explain the implications of the decisions and strategies on the success of the deployment process.

Stage 1: Enforcement and Initial Deployment

Participants P3 & P4 deployed CSP very late, after writing at least one sanitizer such that they were overwhelmed by the number of errors that are caused. Also, participants P5 & P6 did not manage to deploy a CSP. To get deeper insights into the deployment process we tried to ease this part of the deployment process. Thus, we added commented-out boilerplate code to set a CSP header that enabled Trusted Types for scripts and enforced a *default* policy.

Also, the participants were confused to use named policies, because one of the information sources recommended their usage.

> *"Is this the default policy? Because they had something written here, you should not always take the default, but tinker with the specific one."*
> – Participant 12

However, this is not possible as third-party code cannot be adjusted to use the named policy. Therefore, we added boilerplate code for the creation of a *default* Trusted Types policy that is using the *id* function for all sanitizers after participant P5.

With the new changes, Participant P7 managed to deploy the CSP that enforces the default policy with the ineffective id function sanitizers. The enforcement of this default policy resulted in errors caused by Twitter's usage of source-less same-origin iframes for their timeline widget. Those kinds of iframes inherit the CSP policy that mandates the usage of Trusted Types, but they do not inherit the sanitizer functions. Requiring the usage of those functions without them being present then results in multiple errors occurring in the App. Notably, Twitter's behavior regarding how the widget is built has changed throughout the process of this study. Upon request at the Twitter Community Hub [132], they acknowledged that this change is more of a usability/accessibility-driven change because the server-side rendering of the timeline is reducing the loading time of the widget, rather than a change to ease the deployment of Trusted Types.

Stage 2: HTML Sanitizer

All participants started with implementing the *createHTML* sanitizer. Participants P4, P7, P8, P11, and P12 used a third-party HTML sanitizer (DOMPurify[4]) to implement the *createHTML* function. Because none of the third parties that we used required the usage of adding JavaScript in any form into the DOM, this path lead to a successful deployment of the *createHTML* sanitizer.

[4] `https://github.com/cure53/DOMPurify`

> *"There was also a library recommended here. If OWASP recommends DOMPurify, then I'll take it!"*
> – Participant 11

The other participants (P3, P5, P6, P9, P10) implemented the sanitizers themselves, which in all cases led to an easily bypassable filter. Most participants started by copy-pasting the example sanitizer from one of the information sources. This sanitizer however replaces all < with their HTML equivalent (<), thus all inserted tags are destroyed, which causes Twitter and Facebook to not work anymore. Some of them (P6, P10) then changed this code to only removed <script. In either case, the injection of inline event JavaScript event handlers, such as the onload handler of an HTML image tag, would still result in code execution. Thus, in total, all custom sanitizers were trivially bypassable, so only five out of 10 participants managed to implement a secure HTML sanitizer function for Trusted Types.

Stage 3: JS Sanitizer

Five of our participants (P4, P7 & P10-12) first reused their createHTML sanitizer to also sanitize JavaScript. Both, the participant's custom HTML sanitizers as well as DOMPurify replaces an opening HTML tag (<) with its encoded equivalent (<) or the empty string. The evaluated code from the third-party Mensa widget contains the following JS code snippet:

```
for (let i = 1; i <= 5; i++) { ... }
```

With the HTML sanitizer function, this code was therefore changed to, for example:

```
for (let i = 1; i &lt;= 5; i++) { ... }
```

Thus, the change causes a syntax error during the eval call. The participants that took this path needed quite a bit to notice this, as Chromium only pointed to the call of the eval for the syntax error. To find the root cause of the error the developers were required to investigate how this functionality works in detail and investigate the source of the evaluated source code and compare it with the result of the sanitizer.

Those participants that did not re-use their createHTML and those who afterward changed their sanitizer due to the errors mentioned above, mentioned diverse ways how to implement the createScript sanitizer. Participants 6 & 8 mentioned that Regular Expressions could be used to differentiate between benign and attacker-controlled code. The possibility to base the createScript sanitizer on cryptographic signatures to only evaluate benign code was mentioned by participant 11. Others (P7 & P11), however, tried to circumvent the problem by moving the code that required the usage of eval into Cross-Origin iframes such that their execution content differs and Trusted Types are not inherited into the iframe. The feasibility of those sanitization abroaches will be discussed later in Section 6.4.3.

Stage 4: URL Sanitizer

Also here participants re-used the HTML sanitizer, but because the URLs in our application did not contain any HTML characters, like <, they did not recognize that this might be an issue. Those that noticed or directly created a sanitizer themself used an allow-list-based approach. Throughout that process, some participants noticed that the dynamically added URLs contained a random string that might change.

> *"It is a hash here. I could well imagine that it changes with every refresh or so."*
> – Participant 8

In those cases, the participants changed their implementation to an allow-list based on hostnames rather than URLs or used the `startsWith` function to check for the URL without considering the seemingly random string. As using hostnames might lead to JSONP-based script executions [136] being possible, the `startsWith` allow-list is the solution that we would consider as secure.

Edgecase:

We mentioned during the introduction of the coding task that the participants can freely choose their information sources and that they can use anything during the coding task that they consider helpful. One participant used the AI-based GitHub Copilot[5] to write the Trusted Types sanitizer functions.

> *"I could actually be quite bold here once and say: '// Generate Trusted Types using DOMPurify', then I'll see what it does. Ah, look there."*
> – Participant 12

The generated code for the `createHTML` sanitizer was correct because it simply used DOMPurify (without actually importing it though). However, the generated code for the other two sanitizers also used DOMPurify, which not only resulted in trivially bypassable sanitizers as the library is not designed to sanitize JavaScript code or URLs, but it also caused the Mensa feature of the application to not work anymore due to the reasons explained in Section 6.3.2. See Listing 7 for the entire output of the GitHub Copilot.

6.3.3 Client- vs. Server-side XSS

While nine participants properly explained the concept of server-side XSS, only seven of those managed to describe client-side XSS and the difference between them. Notably, none of the participants were able to create a proper sanitizer as they tried to write a customized sanitization function (e.g. based on Regular Expressions).

[5] `https://github.com/features/copilot`

Listing 7 GitHub Copilot output

```
// generate trusted types using dompurify
var trustedHTML = DOMPurify.sanitze(dirtyHTML, {RETURN_TRUSTED_TYPE:
↪ true});
var trustedScript = DOMPurify.sanitze(dirtyScript,
↪ {RETURN_TRUSTED_TYPE: true});
var trustedScriptURL = DOMPurify.sanitze(dirtyScriptURL,
↪ {RETURN_TRUSTED_TYPE: true});

// assign trusted types to DOM elements
document.getElementById('trustedHTML') .innerHTML = trustedHTML;
document.getElementById('trustedScript') .innerHTML = trustedScript;
document.getElementById('trustedScriptURL') .innerHTML =
↪ trustedScriptURL;

// assign trusted types to attributes
document.getElementById('trustedScriptURL') .src = trustedScriptURL;
document.getElementById('trustedScriptURL') .href =
↪ trustedScriptURL;
document.getElementById('trustedScriptURL') .onclick =
↪ trustedScriptURL;
```

"In the real world, if I wanted to protect against XSS attacks, I wouldn't just use JavaScript, I would use PHP, which has much more functions!"
– Participant 6

Those custom HTML sanitizers also ignored the fact that JS cannot only be executed by script tags but also via event handlers or URLs, as they only removed script tags but left nothing else though. Therefore JavaScript URLs (URLs using the `javascript:` schemata), or `onload/onerror` JavaScript handlers of for example HTML image tags, can be used to execute malicious code. Also, those participants spend quite some time creating those trivially bypassable filters and therefore did not manage to work on sanitizing JavaScript code or script URLs.

6.3.4 Perceptions on XSS and Trusted Types

From the five participants that used third-party sanitization libraries for the implementation of the `createHTML` sanitizer, four explained that they are doing this because self-created sanitization functions or custom security solutions, in general, are prone to be bypassable.

"Looks like a lot of manual work and gives the user many possibilities to do things wrong with this API"
– Participant 10

While this perception of sanitization seemingly pushed the participants into using proper sanitization libraries instead of creating a sanitizer themself, others might cause them to incautiously write dangerous code, as three participants mentioned that if you use common frameworks, XSS attacks cannot happen anymore.

"I don't know how all these frameworks, like React or something do that, that they don't have XSS."
– Participant 4

However, although React automatically escapes string variables in views if added to the DOM, there are still XSS vectors that are not secured per default. Despite the trivial exploitability if functions like `dangerouslySetInnerHTML` are present, it is still, for example, possible to execute JavaScript by setting `href` or `src` attributes of HTML tags to `javascript:` or `data:` URIs [101]. Even if frameworks would secure all first-party code, the common practice in the Web to include third-party code, which might contain client-side XSS vulnerabilities, can still cause harm to the users of the Web application.

6.4 Discussion

This section explains in detail some of the problems that our participants have faced and discusses ways how those issues can be fixed or mitigated. Additionally, we discuss improvement suggestions that our participants mentioned after the coding task. Also, we discuss the possible limitations of this work at the end of this section.

6.4.1 Sourceless iframe Problem

Up until participant 7 Twitter's timeline widget was loaded via a script that created a sourceless iframe (see Listing 8) that contains the requested timeline. Because iframes without a source attribute are considered same-origin iframes, the CSP deployed by the including page is inherited, because otherwise, it would be trivial to bypass the protection offered by the CSP. If an attacker abuses a markup injection to inject an

Listing 8 Sourceless iframe Example

```
<iframe id='twitter-widget'>
  <!-- Content to show the timeline -->
</iframe>
```

Listing 9 Example Hook for appendChild.

```
let TTCode = `(${addTrustedTypesFunc.toString()})();`;

let ogFunc = Node.prototype['appendChild'];
Node.prototype['appendChild'] = function () {
    let el = ogFunc.apply(this, arguments);
    if (el.tagName && el.tagName.toLowerCase() === "iframe" &&
     ↪  el.contentWindow) {
        let trusted = window.defaultPolicy .createScript(TTCode);
        el.contentWindow.eval(trusted);
    }
    return el;
};
```

iframe tag without a source, but content or a `srcdoc` attribute, and the parent's CSP would not be inherited, JavaScript code within the iframe would be executed within the execution context of the page's origin and thus be able to successfully execute arbitrary JavaScript code.

One problem that at least one of our participants identified before Twitter changed this behavior is that in the case of Trusted Types deployment the CSP that enforces the usage of Trusted Types is inherited by the sourceless iframe twitter creates, but the actual sanitizer policies are not as they are handled as every other included JavaScript function is. As a result of this, the iframe enforces the usage of a Trusted Types policy for which no named sanitizer exists, which therefore will result in errors and malfunction of the Twitter timeline widget.

One way for developers to fix this issue themself is to hook calls to all of the functions that can be used to append elements to the DOM (e.g., `insertBefore` or `appendChild`) and in case of an iframe being added directly inject the Trusted Types sanitizer into this frame (see Listing 9). We cannot hook the element creation of the iframe here as we can only script into iframes that are already added to the DOM. If now, however, code in these iframes again creates sourceless iframes the problem will reoccur such that the JavaScript snippets that inject the sanitizers into added iframes need to be aware of their source code and inject themselves into the iframe such that all iframes will be covered.

However, we argue that it would be easier for the developer if not only the CSP but also the registered sanitizers are inherited into same-origin iframes. Given that the allowed names of the sanitizers are mentioned in the corresponding CSP directive, and given that the sanitizers are created with this name using the `trustedTypes.createPolicy` API, we hope that the W3C will change the standard accordingly. Notably, it might happen that a developer explicitly wants to have a different sanitizer in the iframe. In this case, the standard should allow the developer to explicitly mention which sanitizer should be the default sanitizer for iframes.

6.4.2 Standardized & Customizable Sanitazation

Third-Party sanitization libraries such as DOMPurify were used by five of our participants. Notably, all five then used this sanitizer of HTML content to sanitize also JavaScript code and URLs, but three of them later noticed that this will not work. Thus, the initial misconception that DOMPurify is a sanitizer for everything was problematic in those cases.

One problem of using a client-side third party is that depending on how it is deployed, the version needs to be maintained as new variants of XSS keep popping up.

> *"XSS attacks, they are always evolving. I'm not sure if Trusted Types can also keep up."*
> – Participant 6

Notably, the planned Web Sanitizer API [13] would fix those issues, or at least shift the responsibility from the operators to the browser vendors' side. However, the Web Sanitizer API can only be used for `createHTML`, but not for `createScript`, as it can also only sanitize HTML content. Also at the time of writing, the Web Sanitizer API cannot exempt certain code snippets from being sanitized, such that specific functionalities of third-party code can be preserved. Thus, the capability of deploying a secure HTML sanitizer using the Web Sanitizer API currently depends on the coding behavior of third parties, which is beyond the control of the Web developer.

6.4.3 Sanitizing JavaScript

As mentioned earlier, but also pointed out by our participants, the `createScript` sanitizer is the hardest part as it cannot properly be solved by an allow-list nor does there exist a sanitization library or API.

> *"createScript is another beast, because it checks the created JavaScript strings. Of course, it is difficult to ensure that this is valid at all, whether it is secure or not."*
> – Participant 8

Nevertheless, our participants mentioned or even implemented some ways how we might be able to sanitize or distinguish attacker code from benign code (see Section 6.3.2).

During the implementation of the `createScript` sanitizer, participant 10 thought about an allow-list based on the Abstract Syntax Tree (AST). However, as shown by Fass et al. [41] attacker could then still create malicious JavaScript with the same AST as the benign and allowed JavaScript snippet.

Participant 11 mentioned that one way to allow specific scripts to be executed is signatures to ensure that the origin of that script is trusted. However, while this works for external resources under the control of the operator, it will only work for third-party scripts if the developer convinces this party to change their functionality to also sign

the code that they deliver to the application.

Participants 6 & 8 had the idea to use Regular Expressions to only allow code with a certain pattern. Besides this only being maintainable at a small scale, it might be bypassable as attackers could be able to write their attacker code in the benign and allowed pattern.

> *"We can not insert a general-purpose sanitizer here now. Maybe Regex, but how efficient is that then? And if-else stories about plain text comparisons of any script content is also kind of difficult."*
>
> – Participant 8

Notably, our sample solution (see Listing 10) for the coding task uses an implementation for `createScript` that used the call stack to allow JavaScript execution from specific functions or parties. However, this only shifts the trust to the third party that delivers the script which invokes the string-to-code function, because they could also be prone to an XSS attack.

Participants 7 & 11 mentioned that instead of somehow implementing the sanitizer they would rather move the content that requires the usage of string-to-code conversion into an iframe that runs under a different origin. While this works for the third parties that we used, others will not work properly anymore as some advertisements require access to the Web page (including its DOM) to work properly [36]. Also, this solution is more of a compartmentalization than an actual sanitization, which is why we would only consider this solution if none of the others are applicable.

6.4.4 Impact of Information Sources

The Information Sources at the time of conducting the study focused on named policies rather than default policies, and partially even advised to not use the default policy sparingly. For four of our participants (P5, P5, P8 & P9) this suggestion caused problems as they first tried to use named policies and then later on noticed that they need to implement a default policy anyway because they cannot change the JS code provided by third parties.

Listing 10 Sample solution for createScript sanitizer.

```
let trustedCallers = [/* allow-list */]

let createScript = function (rawJS) {
    let trustedCaller = arguments.callee.caller.toString();
    if (trustedCallers.indexOf(trustedCaller) === -1) {
        return null;
    }
    return rawJS;
}
```

"I found it difficult in the explanations how to handle own policies or how to set default policies."
– Participant 8

While the usage of named policies enables the developer to create sanitizer functions that are customized towards the data that flows into the respective sink, creating the policies and refactoring the code is work that should be done after having the default policy as a fallback, such that the developer is not overwhelmed by the number of errors. Thus, we argue that Information Sources should advertise the usage of a default policy as a first step and then the usage of named policies for certain parts of the application to harden the protection offered by Trusted Types.

"I could imagine, for example, would be to somehow outsource that into an IFrame or something. So to sandbox that, if I already have such a case. But that would be a completely different topic."
– Participant 11

The participants P3, P6, P8 & P9 also complained about insufficient examples, especially for the usage of a default policy and the `createScript` and `createScriptURL` sanitizer functions. The examples on the information sources only contain explicit examples where strings are sanitized but only superficial examples of the sanitizers.

"In the examples here there are just the strings that can be escaped and then appended, but I don't have any strings here to work with"
– Participant 3

The missing examples for the non-HTML sanitizer might be one of the reasons why five participants initially used the examples for the `createHTML` sanitizer for all three sanitizers. Using the HTML sanitizer for JavaScript and URLs is not only insecure as it does not really sanitize anything for those strings but also leads to a loss of functionality depending on the third-party JavaScript code, as we have pointed out in Section 6.3.2. To ease the debugging process for code that is evaluated through string-to-code conversation, browser vendors might consider changing their error output for those functions to point to the evaluated string.

In general, our participants (P3, P5) mentioned that the information sources for Trusted Types only provide superficial information on the use case for Trusted Types and the deployment process.

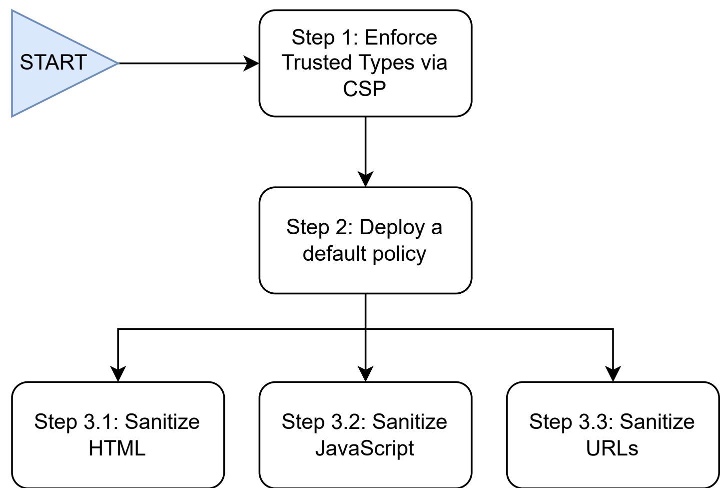

Figure 6.3: Starting point of the interactive roadmap for successful Trusted Types deployment.

"Just with the half-hour now I understood it much better than just in the article and it makes complete sense to me now why I would use those."
– Participant 5

To improve the situation, we are currently in the process of using the results of this work to create an information source including an interactive roadmap (see Figure 6.3 as examples) for successful Trusted Types deployment. By clicking on the parts of the roadmap, a modal opens with explanations and code examples for ach of the individual deployment steps. We also plan to incorporate examples for edgecases like the hook for appending iframes to the DOM or a HTML parser that preserves specific event handlers.

6.4.5 Limitations

Our work has two main limitations. First, we only conducted this study with ten participants. Another problem is the time constraint of the coding task, as the interview was restricted to 90 minutes. However, we argue that if a security mechanism needs massive engineering effort even for a small Web application, the deployment for a big Web application in a company will hardly happen in practice due to monetary and time interests.

Nevertheless, although we reached saturation of our results with ten participants, we do not claim the completeness of the interview results. Half of the participants took courses that involved IT Security during their studies, so they might be more educated than the average Web developer. Thus, we rather see our results as a lower bound for problems that occur during the deployment of Trusted Types.

6.5 Summary

Our work presents the first qualitative study involving ten security-aware Web developers to evaluate the usability of Trusted Types deployment. With our qualitative interview study, and especially the coding task, we were able to uncover important roadblocks of Trusted Types deployment and the strategies that lead to successful sanitizers. Based on those we elaborate on improvement suggestions for the mechanism such that we can ease the deployment of Trusted Types for developers. As the quality of the available information sources was one of the major problems that developers faced or mentioned during the study, we are currently in the process of creating an interactive information source for Trusted Types deployment based on the insights of this work. Also, we hope that our results and the improvements influence the W3C decision process if Trusted Types (maybe together with the Web Sanitizer API) has a future. In general, we encourage the W3C to scientifically evaluate the usability of its drafts with real-world Web developers, such that security mechanisms are protecting all users of the Web platform.

7
Conclusion

In the last chapter, we summarize our contributions, answer our initial research question, and discuss future work before concluding this thesis.

7.1 Summary of Contributions

Our overall research question was *How can developers consistently deploy security mechanisms for their Web applications?*. In order to answer this general question, we divided the problem into four main research questions that we tackled in the respective chapters of this dissertation.

In chapter 3 we analyzed *How the quality of the Content Security Policy configurations and its use-cases has changed over the course of time* (RQ1). By analyzing the deployed CSPs of the 10,000 high-ranking sites from January 2012 through December 2018, we were able to confirm that the vast majority of CSPs in the wild are, and have always been, trivially bypassable. Moreover, we discovered that the use-case of CSP has changed from a mitigation tool against XSS to a multitool for Web security purposes. At the end of our experiment (December 2018), we spotted nearly equal usage of CSP for XSS mitigation, framing control, and TLS enforcement. Also, our longitudinal lens enabled us to quantitatively assess the struggles of deploying a CSP.

Chapter 4 answers the question: *Do all users of a Web site get the same level of security?* (RQ2). Here we tested if factors like the client's geolocations, user agent, or language setting influence the security headers that a client receives. By requesting the top 10,000 most visited sites, with different tests for each of our factors, we were not only able to spot deterministic inconsistencies depending on the factor on 429 sites, but we also discovered 765 sites that non-deterministically delivered different levels of security. This seemingly random behavior of Web sites might not only introduce their customers to security risks, but it has potentially impacted the results of Web measurements on security headers. Furthermore, we discovered that around 10% of the framing control policies in the wild are inconsistently deployed across different browsers.

The question of *What are the root causes for insecure practices in CSP deployment?* (RQ3) is elaborated in chapter 5. After encountering the omnipresent misconfiguration of CSPs in chapter 3, we qualitatively assess the reasons why developers struggle with CSP deployment. By conducting a semi-structured interview that included a drawing and a coding task, we were able to shed light on several roadblocks that are caused by different stakeholders of the Web ecosystem. Moreover, we also investigated motivations to deploy a CSP, strategies for the successful deployment of a CSP, and misconceptions regarding the underlying thread model of CSP.

And finally chapter 6 checks if *Trusted Types is repeating CSP's mistakes.* (RQ4). Trusted Types as a mechanism to mitigate client-side XSS attacks, is a new and not yet standardized security mechanism, which is not widely used. Thus, breaking changes to eliminate roadblocks in the mechanism would not cause issues in real-world Web sites. Notably, the reason why it is not standardized is that the W3C, especially Mozilla, rejected the proposal claiming that it is to complex, without having scientic evidence. Therefore, we conducted a semi-structured interview including a coding task to build a scientific foundation for discussions on the standardization of Trusted Types. By carefully designing the coding task as close as possible to the real world, we were able

to not only investigate similar issues to the one of CSP but also shed light on technical problems of the current approach, e.g., the handling of same-origin iframes.

Overall, we showed that deploying properly configured security headers for Web applications in a consistent manner is a non-trivial task. However, instead of blaming developers for this situation, we showed that other stakeholders are to blame. We discovered technical and design issues of the mechanism, and uncovered the impact of poor information sources. Also, we found rather political issues like the impact of third-party services that are hindering the deployment of security mechanisms without being affected themselves. Thus, to *Deploy Security Mechanisms Online (Consistently)* is currently only possible with high enthusiasm, engineering effort, and willingness to refactor applications or remove dependencies of hindering third-party services.

7.2 Future Work

In this thesis, we have seen that deploying security mechanisms consistently seems to be a non-trivial task, especially when the application is hosted using multiple origin servers. Thus, a developer-centric approach to evaluate the deployment in a multiple-origin server infrastructure can uncover the problems that are causing security inconsistencies. Also, one should create and evaluate tools for detecting inconsistencies and reporting them to the operator of a Web application e.g. from the browser or CDN side. Instead of just open-sourcing research prototypes, those tools should be developed together with Web developers in a participatory design process, in order to increase their acceptance in the Web development community.

Web frameworks provide developers with a set of tools and libraries for building and deploying Web applications and are used by many real-world Web developers. Therefore, research on the interaction between security headers and Web frameworks should be conducted in order to understand how we can ease the creation of secure Web applications. This could involve exploring how different web frameworks handle the implementation of security headers, as well as examining the trade-offs and limitations of using security headers within different web frameworks. Additionally, research in this area could also involve studying the impact of security headers on the performance and scalability of Web applications.

The importance of security in software development extends beyond just web development. It is crucial to consider the security of other systems, which everyone uses on a daily basis as well, e.g., embedded systems. Embedded systems are found in a wide range of devices, including medical equipment, industrial control systems, and automobiles. Vulnerabilities in those areas can have severe and sometimes even deadly consequences. [1, 5]. Thus, it is important to also conduct developer-centric research, that reveals roadblocks and success strategies, in those areas of software development in order to better support the development of secure software.

7.3 Concluding Thoughts

Every developer needs to deploy security mechanisms for their Web site because they help to protect against a wide range of security threats. However, as we pointed out in this thesis, deploying security mechanisms properly configured and consistently is not a trivial task.

A simple remedy for us would be to blame this on the Web developers, but many of the problems that we have seen are caused by the available information sources, the design of the security mechanism, or impeding third-parties behavior. Thus, we as a security community must work together with real-world developers on better information sources. The boards responsible for the design of security mechanisms need to rethink their design process, to be more developer-centric. And we need to encourage third parties to not impede the deployment of security mechanisms, e.g. via changes in browsers in order to ease the deployment of security mechanisms for a Web site's operator. By facilitating the deployment process of security mechanisms for developers, we want to ensure that more Web sites protect their users and data in the best way possible. Thus, Web developers not only protect their reputation and build trust with users, but they also help to make the internet a more secure place for everyone.

Appendix

A Appendix: Complex Security Policy

This section is the appendix of our work *"Complex Security Policy? – A Longitudinal Analysis of Deployed Content Security Policies."* [P1].

A.1 Email notification template

Dear $domain team,
We are a team of academic researchers from $institutions investigating the usage of security headers on the Web.
As part of our analysis, we are investigating the usage of the `X-Frame-Options` header (XFO) to control framing on the Web. Based on our analysis, your site is attempting to control framing with the following XFO directive: `SAMEORIGIN`.
We noticed that this directive potentially allows for double-framing attacks with certain browsers, such as Internet Explorer and Edge (see `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options#Browser_compatibility`)
The currently proposed way to ensure that all modern browsers properly protect against framing attacks is to use the Content Security Policy directive `frame-ancestors`.
In particular, for your value of XFO, the corresponding CSP is: `frame-ancestors 'self'`.
Note that in order to protect older browsers, keeping XFO in place is recommended. As CSP takes precedence over XFO, securing legacy clients without interfering with modern browsers is possible through the usage of the `DENY` directive in XFO.
For more information on CSP's `frame-ancestors`, please see `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors`
As this email is part of a research project in which we are trying to understand the lack of adoption of CSP in the wild, it would be immensely helpful if you could provide us with feedback regarding the lack of CSP's `frame-ancestors` to protect against framing attacks on your Web site (i.e., were you not aware of the CSP mechanism, that specific CSP directive, or, were you not adopting it for some other reason?)
Please note that this email is only part of an academic research project and not meant to sell any products or services.
Best regards,
$researchers

A.2 Quotes from Responses

This appendix sections contains quotes from the responses to our framing control notification campaign.

A.2.1 Complexity of CSP

“Reading the experience of people that tried to do a full CSP implementation is just scary”. “In previous discussions about CSP, we’ve been worried that the risk of accidentally breaking some interaction we have with other [sitename] systems (or the few third party tools we integrate with) outweighs the benefit of implementing these sorts of changes.” “We are pretty certain that there are a lot of pitfalls with implementation of these headers, that might break sections or uses of our site.”

A.2.2 XFO Dangers

“We were vaguely aware of the frame-ancestors option, but our understanding was that XFO was sufficient for securing all clients”. “While we were aware of CSP itself, we were unaware of the fact, that X-Frame-Options allows for attacks under certain conditions, which can be mitigated by using the frame-ancestors directive of CSP.”

A.2.3 `frame-ancestors`

“In my opinion the only advantage of CSP is to protect against XSS [...]”. “As we were not that aware of CSP framing control, we were also not aware of its implementation (no side effects)”

A.3 Survey Questionnaire

1. Did you know about the inconsistent understanding of browsers of the XFO header before our notification (such as the lack of support for `ALLOW-FROM` in Chrome and Safari as well as the potential threat of double-framing in Edge/IE)? (Yes/No)

2. Why have you implemented the XFO header on your site? (Penetration test or consultant suggested it/Tools we used suggested it/Own decision to restrict framing/Other (free text))

3. Did you know about CSP before our notification? (Yes/No)

4. (only if Q3 was yes) Would your site work out of the box if you deployed a script-content restricting CSP today (disallow eval, inline scripts, and event handlers)? (Yes/No/Don’t know)

5. (only if Q3 was yes) Do you believe CSP is a viable option to improve your site’s resilience against XSS attacks? (Yes/No/Don’t know)

6. (only if Q5 was no) Why do you think CSP is not viable for your site? (free text)

7. (only if Q3 was yes) Did you know about the `frame-ancestors` directive of CSP and its improved protection capabilities compared to XFO before our notification? (Yes/No)

8. (only if Q7 was yes) Did you know that `frame-ancestors` can be deployed independently of any other part of CSP before our notification? (Yes/No)

9. (only if Q3 was yes) Did you know that CSP can be used (in isolation) to ensure no HTTP resources can accidentally be loaded (through `block-all-mixed-content`) and to enforce TLS for all resources (through `upgrade-insecure-requests`)? (Yes/No)

10. Do you ever use the developer tools to debug or analyze your site? (Yes/No)

11. What kind of tool support would be useful to you to secure your application? (free text)

B Appendix: The Security Lottery

This section is the appendix of our work *"The Security Lottery: Measuring Client-Side Web Security Inconsistencies"* [P3].

B.1 Disclosure Email

```
Hello,

We are a team of security researchers from the CISPA Helmholtz Center for Information
↪  Security located in Saarland, Germany and Università Ca' Foscari Venezia, Italy.
↪  In our current research project, we investigate inconsistent behavior in the
↪  deployment of security headers for Web applications.

For that, we have visited your site through different vantage points (VPN and Tor) as
↪  well as with different configurations (User-Agents and Accept-Language request
↪  headers).

In our automated tests, we detected both non-deterministic differences (e.g., we
↪  received different levels of security even with the same user agent) or those
↪  differences which seemed related to the vantage point or configuration.

We would like to raise your attention to one of those inconsistencies that occurred
↪  on <DOMAIN>:
<DETAILS_ABOUT_INCONSISTENCY>

We would appreciate if you can check the reason for the issue, address it to ensure
↪  consistent security, and also let us know about what such a reason might have
↪  been, since this will allow us to better help others in the future.

If you have any questions or need further information, please do not hesitate to
↪  contact us by answering this email.
```

B.2 Overview of Additional Crawls

Our confirmation crawls offer two additional insights: on the stability of inter-test inconsistent sites and on the *in*stability of the intra-test inconsistent cases. The data, which is shown in the following tables, highlights that even 12 days after our original crawl, we could still detect 194 inter-test inconsistent sites (see Table 7.2). Intersecting the sites with *intra*-test inconsistencies, however, shows that the numbers seemingly decline (through 100 sites down to 96). This is to be expected, as we are measuring non-deterministic behavior. However, if we take the *union* of all sites which had at least one intra-test inconsistency across any of our crawls, this sums up to 210 (see Table 7.1) sites instead of only 127. This likely means that the actual dangers of non-deterministic header deployment is more severe than what we are able to measure through our limited number of observations.

Mechanism	Usage	# Sites w/ intra-test inconsistencies				
		UA	Lang.	VPN	Tor	**Any**
Content Security Policy	2,029	20	16	42	32	50
- for XSS mitigation	364	1	-	3	1	4
- for framing control	1,313	11	10	23	17	28
- for TLS enforcement	673	12	10	23	16	25
X-Frame-Options	5,751	30	30	64	39	74
Strict-Transport-Security	4,607	27	22	49	50	80
*w/o page similarity**	*4,607*	*80*	*57*	*365*	*947*	*1,152*
Cookie Security	3,975	22	16	26	28	33
- Secure attribute	3,009	9	8	13	13	17
- SameSite attribute	812	13	8	13	17	18
- HttpOnly attribute	3,196	2	-	2	2	3
Any	8,237	90	73	163	135	210

Table 7.1: Union of all intra-test inconsistencies snapshots.

C Appendix: 12 Angry Developers

This section is the appendix of our work *"12 Angry Developers – A Qualitative Study on Developers' Struggles with CSP"* [P4].

C.1 Screening Questionnaire:

The screening questionnaire was conducted using a Django based survey instance hosted by our institution.

C.1.1 Specific Questions:

- For which Web application did you take part in the deployment or maintenance of CSP? Please provide a URL, if possible.
- To what extent were you involved in the maintenance or deployment of CSP for the Web applications you mentioned above?
- When confronted with security-critical decisions, do you make them mostly alone or mostly in a team? *Note: Likert Scale 1–7, 1=strongly agree, 7=strongly disagree*
- Security is my main focus when writing Web Apps. *Note: Likert Scale 1–7, 1=strongly agree, 7=strongly disagree*
- Security plays an important role in my everyday work. *Note: Likert Scale 1–7, 1=strongly agree, 7=strongly disagree*
- How many people are working in your team? And how many of those are specifically dealing with security?

Mechanism	Usage	# Sites w/ intra-test inconsistencies					# Sites w/ inter-test inconsistencies					# Sites w/ only inter-test inconsistencies				
		UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**	UA	Lang.	VPN	Tor	**Any**
Intersubsection of January 2 and January 6																
Content Security Policy	1,987	7	4	27	18	29	15	-	25	15	43	15	-	8	4	27
- for XSS mitigation	357	1	-	-	1	2	9	-	1	1	10	9	-	1	-	10
- for framing control	1,281	3	3	14	7	14	2	-	13	5	17	2	-	6	2	10
- for TLS enforcement	659	4	1	16	11	17	4	-	11	9	16	4	-	1	2	7
X-Frame-Options	5,662	15	13	35	17	44	7	-	22	7	30	7	-	7	2	15
Strict-Transport-Security	4,553	13	12	23	17	30	8	-	17	9	28	8	-	9	3	19
w/o page similarity	-	*37*	*23*	*75*	*322*	*394*	*18*	*2*	*515*	*145*	*583*	*17*	*2*	*489*	*27*	*520*
- preload	918	3	3	6	6	10	-	-	8	3	9	-	-	6	-	6
↳ w/o page similarity	-	*5*	*4*	*12*	*59*	*67*	*1*	*1*	*115*	*29*	*129*	*1*	*1*	*109*	*4*	*112*
Cookie Security	3,836	9	7	10	11	15	147	1	9	4	158	147	1	8	1	156
- Secure attribute	2,907	4	2	5	6	8	142	-	6	3	148	142	-	6	1	148
- SameSite attribute	777	5	5	5	5	7	5	1	3	1	10	5	1	2	-	8
- HttpOnly attribute	3,069	-	-	1	1	2	2	-	2	1	4	2	-	2	-	4
Any	8,145	39	31	86	59	100	174	1	64	30	244	172	1	26	8	191
Intersubsection of January 2 and January 10																
Content Security Policy	1,986	9	4	27	18	30	15	-	26	16	43	15	-	10	4	29
- for XSS mitigation	354	-	-	-	1	2	9	-	1	1	10	9	-	1	-	10
- for framing control	1,285	5	3	15	8	15	2	-	14	5	18	2	-	7	1	10
- for TLS enforcement	658	5	1	16	10	18	4	-	11	10	15	4	-	2	3	9
X-Frame-Options	5,654	14	12	35	19	43	7	-	20	12	30	7	-	6	5	17
Strict-Transport-Security	4,549	12	12	21	16	30	8	-	17	9	27	8	-	10	4	20
w/o page similarity	-	*32*	*24*	*77*	*370*	*443*	*18*	*2*	*512*	*139*	*573*	*17*	*2*	*480*	*18*	*503*
- preload	914	2	3	5	5	9	-	-	8	4	9	-	-	6	1	7
↳ w/o page similarity	-	*4*	*4*	*11*	*71*	*81*	*1*	*1*	*114*	*28*	*130*	*1*	*1*	*108*	*3*	*112*
Cookie Security	3,841	10	8	11	10	16	147	1	10	4	159	146	1	7	1	154
- Secure attribute	2,914	4	3	5	5	8	141	-	6	3	147	141	-	5	1	146
- SameSite attribute	781	5	5	5	5	7	6	1	4	1	12	6	1	2	-	9
- HttpOnly attribute	3,075	1	-	2	1	3	2	-	2	1	4	2	-	2	-	4
Any	8,142	39	30	86	58	100	174	1	66	35	244	173	1	29	12	194
Intersubsection of January 2 and January 14																
Content Security Policy	1,985	8	5	26	20	31	15	-	26	16	43	15	-	10	4	29
- for XSS mitigation	359	-	-	-	1	1	9	-	1	1	10	9	-	1	-	10
- for framing control	1,278	5	2	15	8	16	2	-	13	5	17	2	-	6	2	10
- for TLS enforcement	659	4	3	15	12	18	4	-	12	10	16	4	-	3	2	9
X-Frame-Options	5,654	14	8	32	18	38	6	-	18	11	26	6	-	7	5	15
Strict-Transport-Security	4,548	12	10	19	17	26	7	-	15	7	24	7	-	11	2	19
w/o page similarity	-	*33*	*22*	*65*	*369*	*424*	*17*	*2*	*535*	*136*	*595*	*16*	*2*	*512*	*20*	*535*
- preload	913	3	2	5	6	9	-	-	8	4	9	-	-	6	-	6
↳ w/o page similarity	-	*5*	*5*	*10*	*66*	*73*	*1*	*1*	*119*	*29*	*131*	*1*	*1*	*114*	*2*	*116*
Cookie Security	3,825	10	9	11	11	16	148	1	11	4	161	147	1	9	1	157
- Secure attribute	2,897	4	4	5	6	8	143	-	8	3	151	143	-	7	1	150
- SameSite attribute	778	5	5	5	5	7	5	1	3	1	10	5	1	2	-	8
- HttpOnly attribute	3,066	1	-	2	1	3	2	-	2	1	4	2	-	2	-	4
Any	8,135	38	27	79	61	96	174	1	61	33	239	173	1	31	10	194

Table 7.2: Overview of overlap with additional snapshots of our analysis

C.1.2 Attacker Model:

- Which of the following technologies and services below have you used in the past year? (Check all that apply.)

  - ☐ Social Networks (Facebook, Twitter, etc.)
  - ☐ Online Audio and Video Conferencing (Skype, FaceTime, Google Hangout, etc.)
  - ☐ Office Software (Google Docs, Office Online, etc.)
  - ☐ Mobile Messaging (Signal, Whatsapp, etc.)
  - ☐ Online Banking / Payment (PayPal, etc.)
  - ☐ Online Shopping (Amazon, Zalando, etc.)

- Are you aware of Web attacks when using those services? If yes, name some attacks you consider ...

C.1.3 Demographics:

- Age / Gender / Home Country / Highest (completed) Education Level / Current Occupation / Recent Professional Status
- How big is the company that you are working for? *Note: Buckets according to the EU Commission Recommendation (2003/361/EC) [39]*
- Is the Web presence your company's main business?
- Email address *Note: Mandatory for contact reasons.*

C.2 Interview Protocol:

The interview was an online video conference where we capture screen and audio of us and the participants. **General:**

- In your company, what is the specific area that you cover with your work? What is your specific task in this team?
- Are you considering yourself a Web developer? If yes, since: ...
- Do you have an IT-Security background? If yes, please specify: ...
- Was Web Security and CSP part of your education? If yes, where did you learn about it? If possible, briefly outline the basic content and topics where covered.

C.2.1 Threat Model covered by CSP:

- What was your (or your company's) motivation to deploy CSP?
  - What are the Use-Cases of CSP? (XSS-Mitigation, Framing Control, TLS Enforcement)
  - What are the Attacker capabilities?
    * Why is XSS/Framing/Network attack bad / What bad things might happen?
    * How does CSP defend your Web application against those attacks?
- Do you use other HTTP headers to prevent those attack scenarios? (XFO / HSTS / ...)
- Drawing Task for XSS attack (on a drawing sheet with prefilled icons of the stakeholders.)

C.2.2 Roadblocks for CSP:

Personalized to individual developer group:

1. How did you manage to create a sane CSP?
   - What challenges did you encounter and how did you resolve them?
2. In your CSP, you used $InsecurePractice, what caused the deployment of this source-expression?
   - Do you see any problems regarding this choice?
   - How could you resolve this issue / fix your CSP?
   - Would it be feasible to do this for your Web application?
3. Between $startCSP and $endCSP you experimented with CSP for your Web application. What challenges did you encounter?
   - Why did you abort your experiment of deploying a CSP?
     - Technical issues or bad cost-effectiveness consideration?
     - If technical, what exactly, and do you know how to resolve this issue?
   - What changes would be required that you consider retrying CSP deployment?

Ask all Groups: Have you used any tools / consulting that helped you to create the Policy? If yes, which ones: ...

Validation: Think aloud while deploying a CSP that defends against XSS for the following Web application.

C.3 Recruiting Mail:

```
Subject: Study Invitation to Improve CSP
——
Dear $DOMAIN.TLD team,
We are security researchers from $RESEARCH_ORANIZATION. In our
 research, we have been analyzing several web applications to
evaluate the effectiveness of the deployed security mechanisms
. Currently, we are investigating the Content Security Policy,
 and we noticed that ...

Option 1. your Web site is one of the very few exceptional
cases that actually deployed a sane CSP. In order to
understand how you managed to arrive there and to help other
Web sites to emulate your success, we would like to invite you
 to participate in our study.

Option 2. like a plethora of other Web sites your CSP includes
 insecure entries. With using $INSECURE_PRACTICE an attacker
can easily bypass the XSS mitigation offered by CSP. We would
like to invite you to participate in our study in order to
help us better understand the issues and challenges with
regard to CSP.

The study will consist of two parts: First, a short ( < 5 min
) screening questionnaire, and afterward, we might invite you
to an online video interview which will last for approximately
 one hour. Of course, we will give you an appropriate
compensation of 50 Euro (as Amazon gift card) after finishing
the interview.

If you or one of your colleagues who is familiar with CSP
wants to participate, please fill out our short screening
questionnaire at $LINK. Note that you should have dealt with
CSP at least once. If not, please forward this email to your
colleagues that are responsible for CSP in your company.
Should you need further information or have any other
questions, please do not hesitate to contact us by answering
this email.

Best Regards,
$NAME
——
Footer with full name, exact address, phone number, and email.
```

C.4 Invitation Mail:

```
Subject: Invitation to CSP Interview
---
Hi,

thanks again for participating in our study and helping us to
improve CSP. Your insights and first-hand experience with CSP
will help us to identify roadblocks to a secure CSP deployment
.

We would like to invite you to our 90-minute study. It
consists of (1) a ~ 30 min interview about your experiences
with CSP and (2) a ~ 30 min programming task (+ 30 min buffer
e.g., for setup) where you modify a small Web app in a
programming language of your choice (PHP, Python, or JS).
Please enter your availabilities here to find a suitable date
for your participation in the study: $CALENDLY_LINK

The interview takes place online and can be conducted with any
 software that supports audio and screen recording (e.g. zoom)
.
We want to make the programming task as comfortable as
possible for you. Therefore you have the choice between three
programming languages (PHP, Python, or JS) and can freely
choose between the following setup options:

1. Docker:
We'll provide you with the code as well as Dockerfiles to
build and run the WebApp in a docker container.
Requirements: docker docker-compose

2. Virtual Machine:
We'll provide you with a VirtualBox VM (.ova) that contains
the code and can build and run the WebApp.
Requirements: VirtualBox (and VM Image $OVA_LINK)

3. TeamViewer:
We'll provide you remote access to a VM that contains the code
 and can build and run the WebApp.
Requirements: latest TeamViewer Client

4. Direct Code Execution:
We'll provide you with the source code and a shell script that
 installs all dependencies on your device.
```

```
Requirements: Linux or Windows with installed Windows-
Subsystem for Linux

It would be awesome if you can set up the dependencies for
your preferred version, e.g., install VirtualBox and check if
it is working. At the beginning of the interview, we'll ask
you which of the options you prefer and provide you with the
source code of the WebApp.
Should you need further information, assistance, or have any
other questions, please do not hesitate to contact us by
answering this email.

Best Regards,
$NAME
---
Footer with full name, exact address, phone number, and email.
```

C.5 Demographics of Participants

Gender:	male female	10 1
Age:	20-30 30-40 40-50	6 3 2
Company Size:	< 9 10-49 50-249 > 250	2 2 1 6
Is the Web Presence Main Business:	Yes No No Answer	7 3 1

Table 7.3: Demographics of the 11 participants that completed the survey. (One participant only took part in the interview.)

C.6 Data from the Drawing Task

XSS Type?	Stored Server-Side Reflected Server-Side Both Server-Side	8 1 2
Inline or external payload?	Inline External	8 3
Who executes payload?	Browser Server	9 2
What can happen?	Leak from Browser to evil.com Leak from Server to evil.com Impersonate victim Cryptomining	8 1 1 1
Who enforces CSP?	Browser Server Not mentioned	8 2 1
What is mitigated?	Script execution Script loading Only exfiltration Not mentioned	4 3 3 1

Table 7.4: Drawing task results with concepts and number of participants. (One participant did not draw an XSS attack.)

C.7 Final Codebook:

```
Demography

Motivation: Pentest/ Consulting
Motivation: Role Model
Motivation: Reputation
Motivation: Additional Security Layer
Motivation: XSS Mitigation
Motivation: Framing Control
Motivation: Security Training
Motivation: Build Pipeline Warning
Motivation: Financial Implications

Disincentive: Security Secondary Goal
Disincentive: Build in Security Features —> Frameworks, APIs,
Libraries
Disincentive: Financial Consequences

Benefit: Re—evaluate Resources
Benefit: Re—evaluate Application Structure

Perception_CSP: Additional Security Layer
Perception_CSP: Secondary Security Factor
Perception_CSP: XSS Mitigation
Perception_CSP: Resource Control
Perception_CSP: Framing Control
```

```
Perception_CSP: TLS Enforcement
Perception_CSP: Data Connection Control -> connect-src, form-action
Perception_CSP: CSRF Defense

Knowledge_Gap: TLS Enforcement
Knowledge_Gap: XSS
Knowledge_Gap: Framing Control
Knowledge_Gap: CSP Enforcement
Knowledge_Gap: CSP Concept

Attack_Vector: Click-Jacking -> deployed XFO
Attack_Vector: Session Hijacking
Attack_Vector: XSS
Attack_Vector: MitM -> deployed HSTS
Attack_Vector: Data Exfiltration -> Magecart

Drawing_Task: Stored Server-side XSS
Drawing_Task: Stored Client-side XSS
Drawing_Task: Reflected Server-side XSS
Drawing_Task: Reflected Client-side XSS

Strategy: In-the-field Testing
Strategy: Restrictive RO-Policy
Strategy: Restrictive Enforcement Policy
Strategy: Iterative Deployment
Strategy: Start with generated CSP
Strategy: Externalize Inline Code
Strategy: Externalize Events -> Have Events added programmatically
Strategy: Code Hashing
Strategy: Event Hashing
Strategy: Nonces for inline Scripts
Strategy: Nonces for external Scripts
Strategy: Lax CSP
Strategy: CSP Integral Part of Development
Strategy: Self-host 3rd-Party Code
Strategy: One general CSP
Strategy: Separate CSP for Subpages
Strategy: Remove Dependencies
Strategy: Use script-src-attr
Strategy: Use unsafe-inline as Fallback
Strategy: Changing Functionality
Strategy: Functionality > Security
Strategy: Use Meta Tag CSP
Strategy: Use unsafe-inline

Tool: CSP Evaluation -> Google CSP Evaluator, security-headers.io
Tool: Initial Deployment -> CSPer.io, Mozilla CSP Laboratory, report
.uri wizard
Tool: Report Evaluation -> sentry.io, report-uri.com, DIY
Tool: Developer Tools of Browser
Tool: CSP Preprocessor
```

```
Tool: Code Hashing

Roadblock: Inline Scripts
Roadblock: Inline Events
Roadblock: 3rd-Party Libraries -> Angular
Roadblock: Websocket
Roadblock: 3rd-Party Services -> Google/ Youtube
Roadblock: Legacy Code
Roadblock: Different Development Teams -> Restricted Code Access
Roadblock: Browser Console Inconsistency
Roadblock: Browser Inconsistency
Roadblock: Browser Extension
Roadblock: False Positive Reports -> hacked browser, extensions,
browser features
Roadblock: Amount of Reports
Roadblock: CSP Maintenance -> new content, long header
Roadblock: Engineering Effort
Roadblock: Complexity of CSP
Roadblock: Framework Support -> DIY
Roadblock: Insufficient Error Reports
Roadblock: Information Source

Language: JavaScript
Language: PHP

Bias: Framework Familiarity
Bias: Interview Preparation
Bias: Nervousness

Information_Source: Mozilla Development Network
Information_Source: Blogs -> Scott Helme
Information_Source: Stack Overflow
Information_Source: W3C Specification
Information_Source: content-security-policy.com
Information_Source: Conferences -> OWASPday
```

D Appendix: Trust me If you Can

This section is the appendix of our work *"Trust Me If You Can – How Usable Is Trusted Types In Practice?"* [P5], which is currently under submission.

D.1 Final Codebook

```
{
   "Background":[
      "Hobby-Webdeveloper",
      "Working in IT Security",
      "Security Education",
      "Server-Side-XSS Knowledge",
      "Client-Side XSS Knowledge",
      "XSS-Defense Sanitization/Filtering",
      "XSS-Defense CSP",
      "Not knwon TT before",
      "Not understood Default Policy",
      "Usually no plain JS",
      "Working in DevOps",
      "Understands Default Policy",
      "Working as WebDev",
      "XSS-Defense WAF",
      "Not know client vs. server-side"
   ],
   "Information Source":[
      "Only Superficial Information",
      "Focused on named Policies",
      "Insufficient Exmaples"
   ],
   "Perceptions":[
      "Initial Policy Insecure",
      "XSS is fixed in Frameworks",
      "Manual Sanitizer are bypassable",
      "XSS Mitigations needs to be done on Server-Side",
      "New XSS type -> New Sanitizer",
      "TT complements CSP",
      "TT is as complicated as CSP",
      "Dyn. added script have changing URLs",
      "createScript has no proper solution"
   ],
   "Roadblocks":[
      "CSP Setup",
      "Not universally Supported",
      "Named Policies do not Work",
```

```
      "TT needs HTTPS",
      "Understanding of the sanitizers",
      "Engeeniering Effort",
      "Developer Education",
      "Same-Origin iframe",
      "3rd-Party behaviour"
   ],
   "Strategies":[
      "3rd-Party Sanitizer",
      "Debug with Browserconsole",
      "Setting a CSP",
      "Same Sanitizer 4 All",
      "Use Regular Expressions",
      "Move features to iframes",
      "URL allow-list",
      "Use default as fallback",
      "Only Eval signed-Code",
      "AST based Allow-list"
   ],
   "Improvement":[
      "Frameworksupport",
      "Information / Documentation",
      "Handle iframes differently",
      "Standardized Sanitizer"
   ]
}
```

Bibliography

Author's Papers for this Thesis

[P1] Roth, S., Barron, T., Calzavara, S., Nikiforakis, N., and Stock, B. Complex Security Policy? – A Longitudinal Analysis of Deployed Content Security Policies. In: *Network and Distributed System Security Symposium (NDSS)*. 2020.

[P2] Calzavara, S., Roth, S., Rabitti, A., Backes, M., and Stock, B. A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web. In: *USENIX Security Symposium (USENIX Security)*. 2020.

[P3] Roth, S., Calzavara, S., Wilhelm, M., Rabitti, A., and Stock, B. The Security Lottery: Measuring Client-Side Web Security Inconsistencies. In: *USENIX Security Symposium (USENIX Security)*. 2022.

[P4] Roth, S., Gröber, L., Backes, M., Krombholz, K., and Stock, B. 12 Angry Developers – A Qualitative Study on Developers' Struggles with CSP. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021.

[P5] Roth, S., Gröber, L., Baus, P., Krombholz, K., and Stock, B. Trust Me If You Can – How Usable Is Trusted Types In Practice? In: *Currently under Submission*. 2023.

Other Papers of the Author

[S1] Musch, M., Steffens, M., Roth, S., Stock, B., and Johns, M. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2019.

[S2] Roth, S., Backes, M., and Stock, B. Assessing the Impact of Script Gadgets on CSP at Scale. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2020.

[S3] Stolz, P., Roth, S., and Stock, B. To hash or not to hash: A security assessment of CSP's unsafe-hashes expression. In: *SecWeb Workshop co-located with IEEE Security and Privacy 2022 (SecWeb)*. 2022.

Open-Sourced Artifacts of the Author

[A1] Roth, S. *CISPA GitHub - 12 Angry Developers - Web Applications*. Online at https://github.com/cispa/12-angry-developers-web-applications.

[A2] Roth, S. *CISPA GitHub - Framing Control Analysis*. Online at https://github.com/cispa/framing-control-analytics.

[A3] Roth, S. *CISPA GitHub - Framing Control Proxy*. Online at https://github.com/cispa/framing-control-proxy.

[A4] Sebastian Roth. *CISPA GitHub - The Security Lottery*. Online at https://github.com/cispa/the-security-lottery.

[A5] Sebastian Roth. *GitHub - Archived Dataset for Complex Security Policy*. Online at https://archive-csp.github.io.

Other references

[1] Alex Hern. *Hacking risk leads to recall of 500,000 pacemakers due to patient death fears*. Online at https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update. 2015.

[2] Alotaibi, B. and Elleithy, K. Rogue access point detection: taxonomy, challenges, and future directions. *Wireless Personal Communications* (2016).

[3] Ambroise Maupate. *Nginx CSP example*. https://gist.github.com/ambroisemaupate/bce4b760405558f358ae. 2019.

[4] Anderson, R. *Work with SameSite cookies in ASP.NET*. https://docs.microsoft.com/en-us/aspnet/samesite/system-web-samesite.

[5] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. Online at https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/. 2015.

[6] Barth, A. RFC 6265: HTTP State Management Mechanism. *Oneline a https://tools.ietf.org/html/rfc6265* (2011).

[7] Barth, A. RFC 6454: The Web Origin Concept. *Online at https://www.ietf.org/rfc/rfc6454.txt* (2011).

[8] Barth, A., Jackson, C., and Mitchell, J. C. Robust Defenses for Cross-Site Request Forgery. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2008.

[9] Belshe, M., Peon, R., and Thomson, M. Hypertext Transfer Protocol Version 2 (HTTP/2). *Online at https://www.ietf.org/rfc/rfc7540.txt* (2015).

[10] Berners-Lee, T., Fielding, R., and Frystyk, H. RFC 1945: Hypertext Transfer Protocol–HTTP/1.0. *Online at https://www.ietf.org/rfc/rfc6454.txt* (1996).

[11] Bishop, M. HTTP/3. *Online at* `https://www.ietf.org/rfc/rfc9114.txt` (2022).

[12] Blandford, A., Furniss, D., and Makri, S. Introduction: behind the scenes. In: Morgan and Claypool Publishers, 2016.

[13] Braun, F., Heiderich, M., and Vogelheim, D. The Web Sanitizer API. *W3C Draft. Online at* `https://wicg.github.io/sanitizer-api/` (2022).

[14] Braun, V. and Clarke, V. Using thematic analysis in psychology. *Qualitative research in psychology* (2006).

[15] British Broadcasting Corporation (BBC). *British Airways boss apologises for 'malicious' data breach.* Online at `https://www.bbc.co.uk/news/uk-england-london-45440850`. 2018.

[16] Brooke, J. et al. SUS: A 'Quick and Dirty' Usability Scale. *Usability Evaluation in Industry* (1996).

[17] Bruno Scheufler. *Using security-related headers to secure your application against common attacks.* `https://tinyurl.com/y68c4lpp`. 2019.

[18] Bugliesi, M., Calzavara, S., Focardi, R., and Khan, W. Cookiext: patching the browser against session hijacking attacks. *Journal of Computer Security (IOS Press)* (2015).

[19] Calzavara, S., Rabitti, A., and Bugliesi, M. Content Security Problems?: Evaluating the effectiveness of Content Security Policy in the Wild. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[20] Calzavara, S., Rabitti, A., and Bugliesi, M. CCSP: controlled relaxation of content security policies by runtime policy composition. In: *USENIX Security Symposium (USENIX Security)*. 2017.

[21] Calzavara, S., Rabitti, A., and Bugliesi, M. Semantics-based analysis of content security policy deployment. *ACM Transactions on the Web (TWEB)* (2018).

[22] Calzavara, S., Urban, T., Tatang, D., Steffens, M., and Stock, B. Reining in the web's inconsistencies with site policy. In: *Network and Distributed System Security Symposium (NDSS)*. 2021.

[23] Campbell, J. L., Quincy, C., Osserman, J., and Pedersen, O. K. Coding in-depth semistructured interviews: problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research* (2013).

[24] CanIUse.com. *Can I use Content-Security-Policy: strict-dynamic.* `https://caniuse.com/mdn-http_headers_csp_content-security-policy_strict-dynamic`.

[25] CanIuse.com. *Can I use Content Security Policy 1.0.* `https://caniuse.com/#feat=contentsecuritypolicy`. 2019.

[26] CanIuse.com. *Can I use trustedTypes API?* Online at `https://caniuse.com/mdn-api_trustedtypes`. 2022.

[27] Chromium Blog. *No More Mixed Messages About HTTPS*. https://blog.chromium.org/2019/10/no-more-mixed-messages-about-https.html.

[28] Chromium Blog. *Protecting users from insecure downloads in Google Chrome*. https://blog.chromium.org/2020/02/protecting-users-from-insecure.html.

[29] Chromium Project. *HSTS preload list*. https://hstspreload.org/.

[30] Chromium Project. *HSTS preload list removal*. https://hstspreload.org/removal.

[31] Common Crawl. *So you are ready to get started*. http://commoncrawl.org/the-data/get-started/. 2019.

[32] content-security-policy.com. *CSP: Hashing*. https://content-security-policy.com/hash/.

[33] content-security-policy.com. *CSP: Nonces*. https://content-security-policy.com/nonce/.

[34] Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. Keep me updated: an empirical study of third-party library updatability on android. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[35] Django Software Foundation. *Security in Django*. Online at https://docs.djangoproject.com/en/4.0/topics/security/. 2017.

[36] Dong, X., Tran, M., Liang, Z., and Jiang, X. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In: *Annual Computer Security Applications Conference (ACSAC)*. 2011.

[37] Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C., and Vigna, G. deDacota: Toward Preventing server-side XSS via automatic Code and Data Separation. In: *ACM SIGSAC Conference on Computer and Communications Cecurity (CCS)*. 2013.

[38] Drakonakis, K., Ioannidis, S., and Polakis, J. The cookie hunter: automated black-box auditing for web authentication and authorization flaws. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020.

[39] European Union. EU Commission Recommendation (2003/361/EC). *Online at https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32003H0361* (2003).

[40] Experiments with Google. *Content Security Policy*. https://csp.withgoogle.com/docs/strict-csp.html. 2019.

[41] Fass, A., Backes, M., and Stock, B. Hidenoseek: camouflaging malicious javascript in benign asts. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019.

[42] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. RFC 2616: Hypertext Transfer Protocol–HTTP/1.1. *Online at https://www.ietf.org/rfc/rfc2616.txt* (1999).

[43] Fielding, R., Nottingham, M., and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Caching. *Online at https://www.ietf.org/rfc/rfc7234.txt* (2014).

[44] Fielding, R. and Reschke, J. RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. *Online at https://www.ietf.org/rfc/rfc7230.txt* (2014).

[45] GitHub Blog. *GitHub CSP Blog Post.* https://blog.github.com/2013-04-19-content-security-policy/. 2013.

[46] Google. *WithGoogle: Content Security Policy.* https://csp.withgoogle.com/docs/strict-csp.html.

[47] *Google Analytics Legacy Documentation.* https://developers.google.com/analytics/devguides/collection/gajs.

[48] Guo, S.-y., Ficarra, M., and Gibbons, K. ECMAScript® 2023 Language Specification. *Online at https://tc39.es/ecma262/* (2022).

[49] Hayak, B. Same Origin Method Execution (SOME). *Online at http://www.benhayak.com/2015/06/same-origin-method-execution-some.html* (2015).

[50] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS).* 2012.

[51] Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., and Yang, E. Z. mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS).* 2013.

[52] Hodges, J., Jackson, C., and Barth, A. *RFC 6797: HTTP Strict Transport Security (HSTS).* https://tools.ietf.org/html/rfc6797. 2012.

[53] Hodges, J., Jackson, C., and Barth, A. RFC 6797: Strict-Transport-Security Response Header Field Processing (2012).

[54] Huang, S., Cuadrado, F., and Uhlig, S. Middleboxes in the internet: a http perspective. In: *Network Traffic Measurement and Analysis Conference (TMA).* 2017.

[55] Internet Archive. *About the Internet Archive.* https://archive.org. 2019.

[56] Internet Security Research Group (ISRG). *Let's Encrypt.* https://letsencrypt.org/.

[57] Ion, I., Sachdeva, N., Kumaraguru, P., and Čapkun, S. Home is safer than the cloud! privacy concerns for consumer cloud storage. In: *Symposium on Usable Privacy and Security (SOUPS).* 2011.

[58] Jakobsson, M., Ramzan, Z., and Stamm, S. JavaScript Breaks Free. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.3195&rep=rep1&type=pdf* ().

[59] Javed, A. and Schwenk, J. Towards Elimination of Cross-Site Scripting on Mobile Versions of Web Applications. In: *International Workshop on Information Security Applications (WISA)*. 2013.

[60] John Leyden. *XSS slip-up exposed Fortnite gamers to account hijack*. Online at https://portswigger.net/daily-swig/xss-slip-up-exposed-fortnite-gamers-to-account-hijack. 2019.

[61] Johnson, R. B. and Christensen, L. *Educational research: Quantitative, qualitative, and mixed approaches*. Sage publications, 2019.

[62] Jueckstock, J., Sarker, S., Snyder, P., Beggs, A., Papadopoulos, P., Varvello, M., Livshits, B., and Kapravelos, A. Towards realistic and reproducible web crawl measurements. In: *The Web Conference (WWW)*. 2021.

[63] Jueckstock, J., Sarker, S., Snyder, P., Papadopoulos, P., Varvello, M., Livshits, B., and Kapravelos, A. The blind men and the internet: multi-vantage point web measurements. *arXiv preprint arXiv:1905.08767* (2019).

[64] Kang, R., Dabbish, L., Fruchter, N., and Kiesler, S. “my data just goes everywhere:” user mental models of the internet and implications for privacy and security. In: *Symposium On Usable Privacy and Security (SOUPS)*. 2015.

[65] Kern, C. *Preventing Security Bugs through Software Design*. OWASP AppSec California. 2016.

[66] Kirda, E., Kruegel, C., Vigna, G., and Jovanovic, N. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In: *ACM Symposium on Applied Computing (SAC)*. 2006.

[67] Klein, A. DOM Based Cross Site Scripting or XSS of the Third Kind. *http://www.webappsec.org/projects/articles/071105.shtml* (2005).

[68] Kotowicz, K. Trusted Types - mid 2021 report. *Online at https://research.google/pubs/pub50512.pdf* (2021).

[69] Kotowicz, K. and West, M. Trusted Types. *W3C Standard. Online at https://w3c.github.io/webappsec-trusted-types/dist/spec/* (2021).

[70] Krippendorff, K. *Content Analysis: An Introduction to Its Methodology*. Sage, London, 2004.

[71] Krombholz, K., Busse, K., Pfeffer, K., Smith, M., and Zezschwitz, E. von. "If HTTPS Were Secure, I Wouldn’t Need 2FA"-End User and Administrator Mental Models of HTTPS. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019.

[72] LaToza, T. D., Venolia, G., and DeLine, R. Maintaining mental models: a study of developer work habits. In: *International Conference on Software Engineering (ICSE)*. 2006.

[73] Lawrence, E. *This page frames a victim page in myriad ways*. http://www.enhanceie.com/test/clickjack. 2019.

[74] Lazar, J., Feng, J. H., and Hochheiser, H. *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.

[75] Lekies, S., Kotowicz, K., Groß, S., Vela Nava, E. A., and Johns, M. Code-reuse attacks for the web: breaking cross-site scripting mitigations via script gadgets. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[76] Lekies, S., Stock, B., and Johns, M. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2013.

[77] Lerner, A., Kohno, T., and Roesner, F. Rewriting history: changing the archived web from the present. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[78] Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting. In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[79] Michael, K. and Joseph, B. Upgrading https in mid-air: an empirical study of strict transport security and key pinning. In: *Network and Distributed System Security Symposium (NDSS)*. 2015.

[80] Microsoft. *CSP Level 3 strict-dynamic source expression*. https://tinyurl.com/y3d6ljjk. 2019.

[81] Mike West et al. CfC to publish as an FPWD. Issue No. 342. *GitHub.com. Online at https://github.com/w3c/trusted-types/issues/342-is-still-hard-d2b5c7ad9412* (2018).

[82] MITRE Common Vulnerabilities and Exposures. *Vulnerability Details: CVE-2014-1473*. Online at https://www.cvedetails.com/cve/CVE-2014-1473/. 2014.

[83] Mozilla Developer Network (MDN). *Content Security Policy (CSP)*. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. 2019.

[84] Mozilla Developer Network (MDN). *Data URIs*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs. 2019.

[85] Mozilla Developer Network (MDN). *Referrer-Policy*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy. 2019.

[86] Mozilla Developer Network (MDN). *X-Frame-Options*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options. 2019.

[87] Mozilla Mobile Applications. *Releases of Firefox-iOS*. https://github.com/mozilla-mobile/firefox-ios/releases.

[88] Mozilla Security Blog. *Firefox 83 introduces HTTPS-Only Mode*. https://blog.mozilla.org/security/2020/11/17/firefox-83-introduces-https-only-mode/.

[89] Network, M. D. *CSP: frame-ancestors.* https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors.

[90] Open Web Application Security Project (OWASP). *OWASP Top 10 Web Application Security Risks.* Online at https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS). 2017.

[91] Open Web Application Security Project (OWASP). *HTTP Strict Transport Security Cheat Sheet.* https://www.owasp.org/index.php/HTTP_Strict_Transport_Security_Cheat_Sheet. 2018.

[92] Open Web Application Security Project (OWASP). *Manipulator-in-the-middle attack.* Online at https://nakedsecurity.sophos.com/2011/11/15/facebook-hardcore-porn-violence-and-animal-abuse-images/. 2022.

[93] Opera. *Opera is leading the digital revolution in Africa with nearly 120 million users.* https://blogs.opera.com/mobile/2019/08/opera-is-leading-the-digital-revolution-in-africa/. 2019.

[94] OWASP Security Header Project Statistics. *Global usage of header 'strict-transport-security'.* Online at https://github.com/oshp/oshp-stats/blob/80c05796ba0983bb920b34cd48b956ab2592d1c4/stats.md#global-usage-of-header-strict-transport-security. 2022.

[95] Paicu, A. *CSP3: unsafe-hashed-attributes.* https://www.chromestatus.com/features/5867082285580288. 2017.

[96] Paicu, A. *CSP 'navigate-to' directive.* https://www.chromestatus.com/feature/6457580339593216. 2018.

[97] Paicu, A. *CSP: 'script-src-attr', 'script-src-elem', 'style-src-attr', 'style-src-elem' directives.* https://www.chromestatus.com/features/5141352765456384. 2018.

[98] Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., and Zhou, T. CSPAutoGen: Black-box Enforcement of Content Security Policy upon real-world Websites. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS).* 2016.

[99] Patnaik, N., Hallett, J., and Rashid, A. Usability smells: an analysis of developers' struggle with crypto libraries. In: *Symposium on Usable Privacy and Security (SOUPS).* 2019.

[100] Pellegrino, G., Johns, M., Koch, S., Backes, M., and Rossow, C. Deemon: Detecting CSRF with dynamic analysis and property graphs. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS).* 2017.

[101] Perris, R. Avoiding XSS in React is Still Hard. *Medium.com. Online at https://medium.com/javascript-security/avoiding-xss-in-react-is-still-hard-d2b5c7ad9412* (2018).

[102] Pochat, V. L., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. Tranco: a research-oriented top sites ranking hardened against manipulation. *Network and Distributed System Security Symposium (NDSS)* (2019).

[103] Porter Felt, A., Barnes, R., King, A., Palmer, C., Bentzel, C., and Tabriz, P. Measuring https adoption on the web. In: *USENIX Security Symposium (USENIX Security)*. 2017.

[104] *Prevent nonce stealing by looking for "<script" in attributes of nonced scripts*. https://github.com/w3c/webappsec-csp/issues/98.

[105] Raggett, D., Le Hors, A., and Jacobs, I. HTML 4.01 Specification. *Online at https://www.w3.org/TR/html401/* (1999).

[106] Ringnalda, P. *Getting around IE's MIME type mangling*. http://weblog.philringnalda.com/2004/04/06/getting-around-ies-mime-type-mangling.

[107] Ristic, I. *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Feisty Duck, 2013.

[108] Ross, D., Gondrom, T., and Stanley, T. RFC 7034: HTTP Header Field X-Frame-Options. *Online at https://www.ietf.org/rfc/rfc7034.txt* (2013).

[109] Ross, D. Happy 10th Birthday Cross-Site Scripting. *Online at https://blogs.msdn.microsoft.com/dross/2009/12/15/happy-10th-birthday-cross-site-scripting/* (2009).

[110] Salem Alashwali, E., Szalachowski, P., and Martin, A. Exploring https security inconsistencies: a cross-regional perspective. *arXiv e-prints* (2020).

[111] Saxena, P., Hanna, S., Poosankam, P., and Song, D. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In: *Network and Distributed Systems Symposium (NDSS)*. 2010.

[112] Scheitle, Q., Hohlfeld, O., Gamba, J., Jelten, J., Zimmermann, T., Strowes, S. D., and Vallina-Rodriguez, N. A long way to the top: significance, structure, and stability of internet top lists. In: *ACM Internet Measurement Conference (IMC)*. 2018.

[113] Scott Helme. *Security Headers*. https://securityheaders.com. 2019.

[114] Singh, K., Moshchuk, A., Wang, H. J., and Lee, W. On the incoherencies in web browser access control policies. In: *IEEE Symposium on Security and Privacy (S&P)*. 2010.

[115] Sivakorn, S., Keromytis, A. D., and Polakis, J. That's the way the cookie crumbles: evaluating https enforcing mechanisms. In: *ACM Workshop on Privacy in the Electronic Society (WPES)*. 2016.

[116] Smith, J., Do, L. N. Q., and Murphy-Hill, E. Why can't johnny fix vulnerabilities: a usability evaluation of static analysis tools for security. In: *Symposium on Usable Privacy and Security (SOUPS)*. 2020.

[117] Some, D. F., Bielova, N., and Rezk, T. On the content security policy violations due to the same-origin policy. In: *The Web Conference (WWW)*. 2017.

[118] Son, S. and Shmatikov, V. The Hitchhiker's Guide to DNS Cache Poisoning. In: *International Conference on Security and Privacy in Communication Systems (SecureComm)*. 2010.

[119] SOPHOS nacked security. *Facebook users hit by hardcore porn, violence and animal abuse images*. Online at https://nakedsecurity.sophos.com/2011/11/15/facebook-hardcore-porn-violence-and-animal-abuse-images/. 2011.

[120] Soussi, W., Korczynski, M., Maroofi, S., and Duda, A. Feasibility of large-scale vulnerability notifications after gdpr. In: *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020.

[121] Stamm, S., Sterne, B., and Markham, G. Reining in the web with content security policy. In: *The Web Conference (WWW)*. 2010.

[122] Steffens, M., Musch, M., Johns, M., and Stock, B. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In: *Network and Distributed Systems Security (NDSS)*. 2021.

[123] Steffens, M., Rossow, C., Johns, M., and Stock, B. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In: *Network and Distributed Systems Symposium (NDSS)*. 2019.

[124] Stock, B., Johns, M., Steffens, M., and Backes, M. How the web tangled itself: uncovering the history of client-side web (in) security. In: *USENIX Security Symposium (USENIX Security)*. 2017.

[125] Stock, B., Pellegrino, G., Li, F., Backes, M., and Rossow, C. Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications. In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[126] Stock, B., Pellegrino, G., Rossow, C., Johns, M., and Backes, M. Hey, you have a problem: on the feasibility of {large-scale} web vulnerability notification. In: *USENIX Security Symposium (USENIX Security)*. 2016.

[127] Stock, B., Pfistner, S., Kaiser, B., Lekies, S., and Johns, M. From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[128] Strauss, A. and Corbin, J. M. *Grounded theory in practice*. Sage, London, 1997.

[129] Sudhodanan, A., Carbone, R., Compagna, L., Dolgin, N., Armando, A., and Morelli, U. Large-scale analysis & detection of authentication cross-site request forgeries. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017.

[130] Tang, S., Dautenhahn, N., and King, S. T. Fortifying web-based applications automatically. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2011.

[131] Toomey, P. *GitHub's CSP Journey*. https://githubengineering.com/githubs-csp-journey/. 2016.

[132] Twitter Community. *Reasons to move from same-origin iframes to third-party iframes?* Online @ https://twittercommunity.com/t/reasons-to-move-from-same-origin-iframes-to-third-party-iframes/179321. 2022.

[133] Van Goethem, T., Chen, P., Nikiforakis, N., Desmet, L., and Joosen, W. Large-scale security analysis of the web: challenges and findings. In: *International Conference on Trust and Trustworthy Computing (TRUST)*. 2014.

[134] Wang, P., Gudmundsson, B. A., and Kotowicz, K. Adopting Trusted Types in Production Web Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study. In: *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2021.

[135] Weichselbaum, L. and Spagnuolo, M. *CSP - A Successful Mess Between Hardening and Mitigation*. LocomocoSec, Online at https://locomocosec2019.sched.com/event/Ixt3/content-security-policy-a-successful-mess-between-hardening-and-mitigation. 2020.

[136] Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[137] Weissbacher, M., Lauinger, T., and Robertson, W. Why is CSP failing? Trends and challenges in CSP adoption. In: *Recent Advances in Intrusion Detection (RAID)*. 2014.

[138] West, M. *Upgrade Insecure Requests*. https://www.chromestatus.com/feature/6534575509471232. 2018.

[139] West, M. CSP Level 3. *W3C Standard. Online at https://www.w3.org/TR/CSP3/* (2021).

[140] WhatIsMyBrowser.com. *Latest user agents for Web Browsers & Operating Systems*. https://www.whatismybrowser.com/guides/the-latest-user-agent/whatismybrowser.com.

[141] WHATWG community. HTML Living Standard. *Online at https://html.spec.whatwg.org/multipage/* (2022).

[142] WICG. *Explainer: Trusted Types for DOM Manipulation*. https://github.com/WICG/trusted-types#limiting-policies. 2018.

[143] World Wide Web Consortium. *Fetch Metadata Request Headers*. W3C Working Draft, Online at https://www.w3.org/TR/fetch-metadata/. 2021.

[144] World Wide Web Consortium - GitHub - webappsec-csp. *Issue 7: CSP: connect-src 'self' and websockets*. https://github.com/w3c/webappsec-csp/issues/7.