12 Angry Developers A Qualitative Study on Developers' Struggles with CSP

Sebastian Roth †‡, Lea Gröber †‡, Michael Backes †, Katharina Krombholz †, and Ben Stock †
†CISPA Helmholtz Center for Information Security
‡Saarland University Graduate School of Computer Science
{sebastian.roth,lea.groeber,backes,krombholz,stock}@cispa.de

ABSTRACT

The Web has improved our ways of communicating, collaborating, teaching, and entertaining us and our fellow human beings. However, this cornerstone of our modern society is also one of the main targets of attacks, most prominently Cross-Site Scripting (XSS). A correctly crafted Content Security Policy (CSP) is capable of effectively mitigating the effect of those Cross-Site Scripting attacks. However, research has shown that the vast majority of all policies in the wild are trivially bypassable.

To uncover the root causes behind the 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.

CCS CONCEPTS

• **Security and privacy** → **Web application security**; **Usability in security and privacy**.

KEYWORDS

Usable Security, Web Security, Content Security Policy, Roadblocks

ACM Reference Format:
Sebastian Roth †‡, Lea Gröber †‡, Michael Backes †, Katharina Krombholz †, and Ben Stock †. 2021. 12 Angry Developers A Qualitative Study on Developers' Struggles with CSP. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3460120.3484780

1 INTRODUCTION

Nowadays, Web applications are getting more and more important for both business and private life. They evolved from documents presenting information into sophisticated fully-fledged office and entertainment applications. Thus, the Web as a platform plays an essential part of our daily lives. Since critical applications like online banking are implemented on the Web platform, attacks against its users are getting more severe. One of those attacks is Cross-Site Scripting (XSS), which is one of the most prevalent security issues present in Web applications [39], with over 750 CVEs in 2021 [13]. This investigation is supported by research that has shown that client-side XSS can be discovered on around 8-10% of the top Web sites [48, 50]. By abusing an XSS vulnerability, an attacker can steal session cookies to impersonate the victim, perform actions on behalf of the victim like issuing bank transactions, and a plethora of other devastating actions. To mitigate the effect of those attacks from the Web application itself, browsers support Content Security Policy (CSP). By deploying such a policy, a Web developer can specify a list of allowed JavaScript sources and prohibit the execution of inline scripts, making it hard or even impossible for an attacker to execute their malicious payload.

Although CSP may sound like the holy grail of Web Security, it suffers from several issues. Research has shown that the majority of all policies deployed by real-world Web sites are trivially bypassable because they either allow the execution of inline JavaScript or allow all resources of a specific scheme [44, 58]. If inline code is allowed, the attacker can directly inject a script tag or an event handler that executes the attack payload. In case of allowing a scheme, e.g., `https:`, the attacker can inject a script tag that loads a payload via `https:` that is under their control. Through free certificate authorities like Let's Encrypt [24], attackers can easily host resources under their control at `https:` URLs without any cost or travails.

The fact that policies are insecure in the wild has been documented by numerous works [7, 44, 49, 58, 59], but the reasons remained largely hidden. In order to understand the nature of such insecure practices, Roth et al. [44] investigated the HTML documents of Web applications. Here, they found high reliance on inline event handlers, which are not trivial to allow by a CSP. To understand why developers tend to use old and deprecated security mechanisms like `X-Frame-Options` instead of using the easy-to-use features of CSP for that use case, they also sent out a survey to those sites. Through this, they discovered 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

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484780

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 wrong directions 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.

2 BACKGROUND & RELATED WORK

Due to this work's focus on misconceptions about CSP and the usability of CSP deployment, this section describes details about the security mechanism itself, the threat model that is originally mitigated by CSP, as well as Web-related user studies, and a brief overview of qualitative methodologies used in usable security.

2.1 Cross-Site Scripting

The most basic security mechanism incorporated in every browser is the Same-Origin Policy (SOP). In essence, it protects distrusting pages from each other by ensuring that only documents with the same Origin [2], i.e., matching protocol, hostname, and port, can access each other's content. By performing a Cross-Site Scripting (XSS) attack, an adversary can execute JavaScript in the origin of a vulnerable Web page, and this code has the same capabilities as the document's legitimate one. It can modify the page content to the attacker's liking, exfiltrate sensitive information such as session cookies, steal login credentials, or perform any action on behalf of the victimized user. Since its initial discovery back in 1999 [42] a plethora of publications focus on these attacks [20–22, 25, 28, 33, 34, 41, 45, 48, 52], their different variants, and their individual impacts, showing that XSS is here to stay.

2.2 Content Security Policy

XSS attacks can cause severe harm to the users of a Web application. In order to mitigate the effect of those unintended code executions, Stamm et al. [47] introduced the Content Security Policy (CSP). By deploying such a policy, e.g., via an HTTP header, a developer can control which sources are allowed for certain types of resources, such as scripts or images. A policy consists of multiple (semicolon-separated) *directives*, each followed by a list of *source expressions*. Such an expression is a representation of an allowed content source for the directive. All the so-called *fetch directives*, such as `script-src` or `img-src` (for images) fall back to `default-src` if the more specific directive is missing. As an example, a policy that restricts scripts to be hosted on the same origin as the including page as well as `ad.com` and disallows any other content to be loaded from anywhere can be enforced as follows:

```
default-src 'none'; script-src 'self' ad.com;
```

Implicitly, when either `script-src` or `default-src` are deployed, inline scripts, event handlers, and functions that perform a string-to-code transformation (e.g., `eval`) are forbidden. In order to re-enable these dangerous constructs, a developer can specify the `'unsafe-inline'` or `'unsafe-eval'` source expressions, thereby explicitly opting out of security.

In CSP's initial version, inline scripts could only be enabled through a policy that contains `'unsafe-inline'` [54]. However, the presence of this expression undermines any protection capability of CSP, as the attacker can simply inject inline JavaScript into a vulnerable page. Hence, the only solution to this problem was to externalize inline scripts, requiring significant engineering effort. Even though academic approaches to automate this task exist [15, 40], policies remained insecure from the first analyses in 2014 through 2020 [7, 44, 58, 59].

In order to both ease the way of allowing inline script as well as improving the security of CSPs in the wild, the second version of CSP [55] added support for nonces and hashes to the `script-src` directive. This mechanism allows the developer to explicitly allow *their* scripts. Specifically, they can either add hashes of allowed scripts or a nonce to the policy. While the former implies that any script which matches an allowed hash is executed, the latter implies that scripts which carry the nonce in their `nonce` property can be executed. Since the nonce is randomly generated, it cannot be guessed by an attacker, leaving them unable to inject a script that would execute, yet allowing the developer to have their scripts execute. To still provide backward compatibility for CSP Level 1 browsers, the *'unsafe-inline'* expression is ignored if nonces are present, such that the inline scripts are still executed in old browsers that do not support nonces or hashes. Notably, event handlers cannot be allowed through nonces. However, the most recent draft of the CSP Level 3 living standard adds the `'unsafe-hashes'` keyword to enable hashed event handlers.

Once a script is loaded, it can add arbitrary other scripting resources. On the Web, this frequently occurs, particularly with newly added hosts, in the context of advertisements. This forces the CSP author to either regularly update their policy or risk breakage. To address this problem, Weichselbaum et al. [58] proposed the `'strict-dynamic'` expression, which is, in the meanwhile, present in the living standard of CSP Level 3 [56]. If this expression is present, a trusted script – explicitly allowed through its hash or through being nonced – can propagate its trust to *programmatically* added scripts, which do not have to be explicitly allowed in the `script-src` directive. Because deploying `'strict-dynamic'` only works if hashes or nonces are present, it also means that `'unsafe-inline'` has no effect if it is present.

Since developing and deploying a CSP is a tedious task which can cause breakage when not properly tested, CSP can also be delivered as a *report-only* policy. In that case, violating resources are not blocked, but instead, a report is generated to address the potential breakage before it occurs in *enforcement* mode. For both enforcement and report-only modes, CSP supports the `report-to` and `report-uri` directives, which specify a logging endpoint to which violation information is sent.

CSP has been the subject of many studies over the years, which all showed that policies in practice are mostly insecure [7, 44, 58, 59]. Moreover, even in cases where a CSP is secure, consistent deployment across the entire origin often lacks behind, opening the door for bypasses [9, 46]. Deploying a functional CSP is made increasingly harder through browser extensions that inject content, thereby causing breakage and/or false warnings [19].

As shown by Steffens et al. [49], the vast majority of sites have at least one third party that interferes with CSP through practices that require `'unsafe-inline'` or rely on APIs like `document.write` to add scripts, making them incompatible with `'strict-dynamic'`. To investigate the evolution of CSP over the course of time, Roth et al. [44] conducted a longitudinal analysis of deployed CSPs from 2012 until the end of 2018. They found the majority of CSPs were trivially bypassable, e.g., because of the usage of `'unsafe-inline'` or `http://*` expressions, for long periods of time, or sites gave up after experimenting with report-only mode.

We pick up on this notion of a CSP which is not trivially bypassed, and refer to it as a *sane* CSP. While other bypasses exist, e.g., through JSONP endpoints [58] or open redirects [43], such a sane policy nevertheless indicates that it is not trivial to bypass, but rather requires certain preconditions to be insecure. While prior work has documented the technical struggles in deploying CSP from an empirical and quantitative point-of-view, they fall short of understanding the reasons behind this failure. Roth et al. [44] conducted a short survey about CSP in the context of their notification around framing control but could only generally report that developers have misconceptions about CSP.

In this paper, through a qualitative study with developers with CSP experience, we dive much deeper into the roadblocks of CSP and outline how they align with technical challenges and human misconceptions.

2.3 Qualitative Methods

Qualitative and quantitative research methods complement and benefit from each other. This is due to the different perspectives that the two approaches offer [1]. Quantitative research follows a "top-down" approach [26] that requires well-founded hypotheses. These may be constructed based on theories from qualitative research, among other sources. As a "bottom-up" approach [26], qualitative research is particularly useful to explore new areas where prior research cannot be relied upon, or to explore the origins of behaviors and misconceptions, for example. In our case, we use the strengths of qualitative methodology to uncover root causes of the phenomena that previous quantitative work identified [44]. One of the most prominent qualitative methods is interviews. In particular, semi-structured interviews can be used to understand the problems, perspectives, and needs of administrators [30], developers [31], and end-users [23, 27, 30] better. While open-ended questions allow exploring a topic broadly, the interviewer can deviate from the interview guideline as soon as an interesting concept emerges. Upon appropriate follow-up questions, interviewees provide insights into their thoughts, reflect and share their insights and ideas. This depth and data quality distinguishes interviews, and is difficult to obtain with other methods such as surveys. However, interview studies are costly, which is reflected in a relatively small sample size [32].

Another methodology that is well-suited for collecting qualitative data is a lab study. A particular advantage of lab studies is the controlled setting, where researchers can change factors at will to observe possible effects. Thus, lab studies are very versatile and, depending on the study design, quantitative and/or qualitative data can be collected. For example, study participants' actions can be recorded in video and audio, and they can be encouraged to think-aloud. This helps to understand better the motivations behind decisions, misunderstandings, and mistakes [32]. Gorski et al. [18] conducted a between-subjects lab study to investigate if the enforcement of a CSP per default affects the usability of a Web framework. They also evaluated the effectiveness of CSP warnings in the developer console of Chrome and Firefox. They recruited 30 students from their local university and made them use the Java-based Play framework on Windows via the IntelliJ IDEA IDE to create a Web site embedding a specific point on Google Maps. After completing this coding task, the students also took part in a semi-structured interview. Their findings suggest that enforcing CSP usage per default in IDEs does not lead to increased security. Notably, there is a discussion on how far students can be recruited instead of developers for developer-centric topics [36, 37]. In our case, this was not an option, as we heavily rely on real-world experience to uncover root causes for CSP's poor adoption rate. However, as Lazar et al. [32] point out, it can be challenging to recruit "*specialized populations, such as highly trained domain specialists*", as is the case with CSP knowledgeable Web developers. Hence, we decided to conduct a controlled online coding task to mitigate the impact of geographic location on our recruitment.

Drawing tasks can be powerful to visualize participants' understanding of a system or concept [3]. 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 [27], and end users' and administrators' understanding of HTTPS [30].

3 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 [44]. Our findings can be used to improve the mechanism and ultimately remove roadblocks to its successful adoption. Accordingly, we answer the following research questions:

RQ1: What are the root causes of insecure practices when deploying a CSP?
RQ2: What strategies do developers adopt when creating a CSP?
RQ3: How well do developers understand the associated threat models of CSP?
RQ4: 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.

3.1 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 Appendix C). 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 Appendix D) 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.

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 Appendix E) 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 [32].

3.2 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 Appendix A). 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 Appendix A.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 Appendix A.2). Last, we ask for demographics covering age, gender, profession, education, home country, and the company size [53] (see Appendix A.3).

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 Appendix F) 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 coding task, choice of video conferencing software, and choice of the study date [35].

3.3 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 1). The drawing is done either via Zoom's Annotate feature or alternatively with enabled screen sharing and diagrams.net [14]. After finishing the drawing, we ask the participants at which point of this attack a CSP could successfully stop exploitation.

While Roth et al. [44] 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

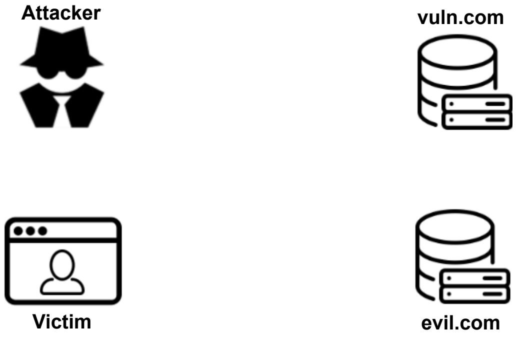


Figure 1: Template for XSS/CSP Drawing Task

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 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 [44] 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 B.

3.4 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 pre-configured virtual machine for virtual box. In addition to that, we offer the opportunity to remote control this VM using Zoom, or alternatively use TeamViewer [16]. 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 [7, 58, 59] and more explicitly by Roth et al. [44] and Steffens et al. [49], 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 [1].

3.5 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

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

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.

3.6 Data Analysis

We manually transcribed all collected data, including the coding and drawing task. Afterward, we unitized [10] the transcript and conducted open coding according to Strauss and Corbin [51] 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 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 Appendix I.

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* [6] to the coded data to identify emerging themes and patterns. Then we conducted *axial coding* [51] 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.

3.7 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.

4 RESULTS

In this section, we present participant demographics and results of our *thematic analysis*. The *inter-coder reliability* Krippendorff's α [29] 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).

4.1 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 Appendix G. 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.

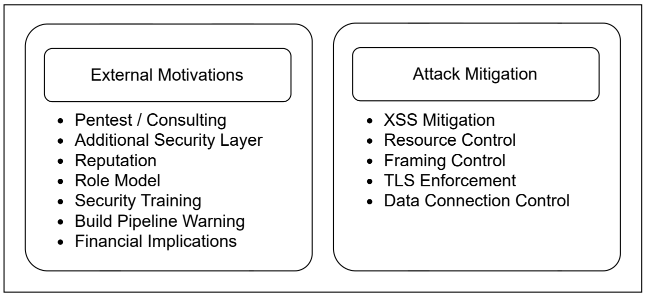


Figure 2: Categories of Motivations

4.2 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 2 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 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.

4.2.1 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, for example, "*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.*"(P5). 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 [38]. 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.*"(P3). 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 [8]. 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 [4] and Firefox is automatically upgrading HTTP connections [5]. 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.

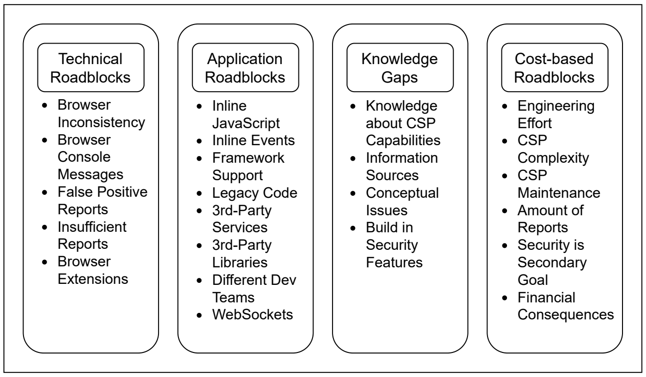


Figure 3: Categories of Roadblocks

4.3 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 3. Notably, these roadblocks are results from both the semi-structured interview and the coding task.

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.

4.3.1 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.*" (P4)). 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.

4.3.2 Application Roadblocks. Some of the application-based roadblocks are those that were hypothesized by previous work on CSP [44]. 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.*"(P12)). 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 [49]. 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 [56], others require to allow the own domain with the WebSocket protocol explicitly. This issue was already discussed in the CSP GitHub repository [57], leading to a change in the living standard [56]. 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.

4.3.3 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.

4.3.4 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 deployment 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 4.3.1. 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.*"(P5)).

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.

4.4 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 4 gives an overview about the strategies that the participants mentioned during the interview or strategies they have taken into account for the coding task.

4.4.1 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 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.*"(P10)). 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.

4.4.2 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

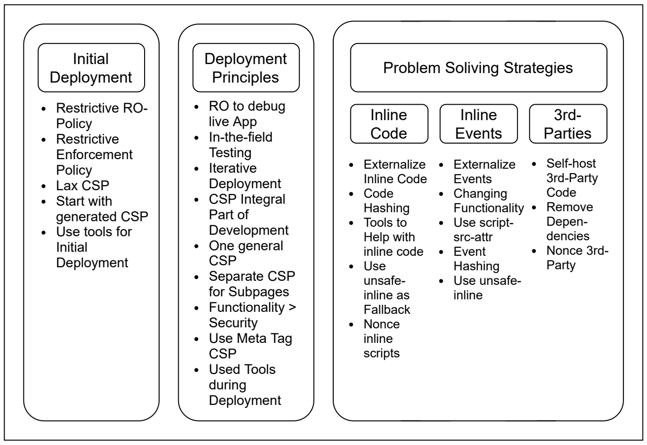


Figure 4: Categories of Strategies

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.*"(P3)).

4.4.3 Problem Solving Strategies. In order to solve some of the application issues mentioned in Section 4.3.2, 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 [12]. 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 [11]. 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 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.

5.1 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. [46] 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.

5.1.1 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 programmatically, 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* [17], 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) 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 [34, 43] or vulnerable JSONP endpoints [58]. Also, the common practice of third parties to programmatically add scripts [49] 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.

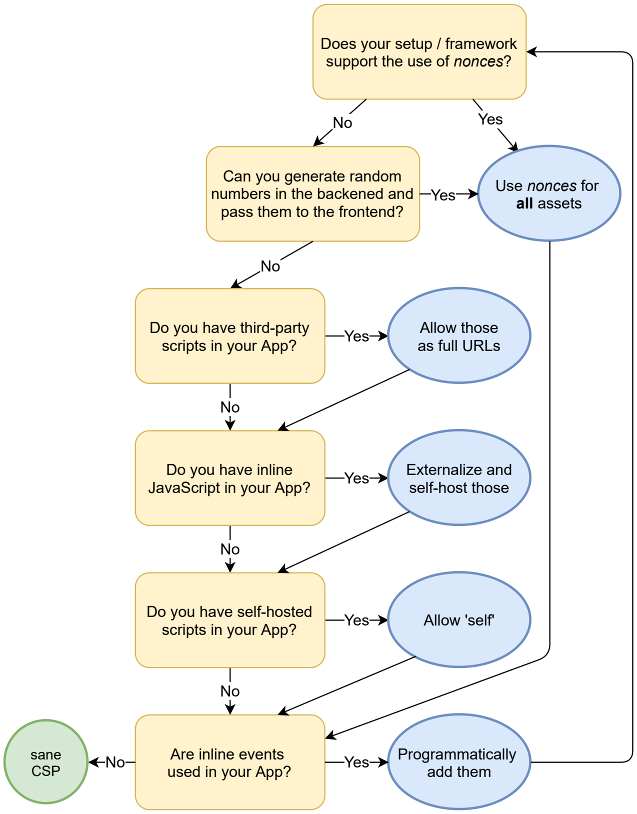

Does your setup / framework support the use of *nonces*?
No
Yes
Can you generate random numbers in the backend and pass them to the frontend?
Use *nonces* for **all** assets
Do you have third-party scripts in your App?
Allow those as full URLs
Do you have inline JavaScript in your App?
Externalize and self-host those
Do you have self-hosted scripts in your App?
Allow 'self'
Are inline events used in your App?
Programmatically add them
sane CSP

Figure 5: Developer decision tree to create a sane CSP.

As mentioned in Section 4.4.1 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 4.4.2, 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 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.

5.2 Drawing Task

As mentioned in Section 3.3, 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." (P9). 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 Appendix H. 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.

5.3 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 numbers 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 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 4.1 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.

6 CONCLUSION

In this paper, we present the first qualitative study involving 12 real-world Web developers to evaluate the usability of the CSP. Throughout our interview that 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 mechanism in Web security, but we hope our insights will inform the design process of new mechanisms to avoid such problems.

ACKNOWLEDGEMENTS

We want to thank the reviewers for their insights and constructive feedback regarding our paper's presentation. Furthermore, we want to thank our colleagues Marius Steffens, Carolyn Guthoff, and our student helpers Moritz Wilhelm, Daniel Emmel, Simon Anell, Priyasha Chatterjee, Xin Yun, Richard Bergs, Florian Romann, and Tim Recktenwald for their feedback during the prestudy and their assistance in the transcription process. Finally, we would like to thank all of our participants for their time and insights that ultimately made this project possible.

REFERENCES

[1] Vishal Arghode. Qualitative and quantitative research: Paradigmatic differences. *Global Education Journal*, 2012(4), 2012.

[2] A. Barth. RFC 6454: The Web Origin Concept. *Online at https://www.ietf.org/rfc/rfc6454.txt*, 2011.

[3] A Blandford, D Furniss, and S Makri. Introduction: Behind the scenes. 2016.

[4] Chromium Blog. Protecting users from insecure downloads in google chrome. https://blog.chromium.org/2020/02/protecting-users-from-insecure.html, .

[5] Mozilla Security Blog. Firefox 83 introduces https-only mode. https://blog.mozilla.org/security/2020/11/17/firefox-83-introduces-https-only-mode/, .

[6] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[7] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *CCS*, 2016.

[8] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: a formal analysis of inconsistent click-jacking protection on the web. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 683–697, 2020.

[9] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. Reining in the web's inconsistencies with site policy. In *NDSS*, 2021.

[10] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research*, 42(3):294–320, 2013.

[11] content-security policy.com. Csp: Hashing. https://content-security-policy.com/hash/, .

[12] content-security policy.com. Csp: Nonces. https://content-security-policy.com/nonce/, .

[13] MITRE Common Weakness Enumeration (CWE). Cve search for security vulnerabilities (cross site scripting (xss)). https://www.cvedetails.com/vulnerability-list.php?vendor_id=0&product_id=0&version_id=0&page=15&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=1&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=0&cvssscoremax=0&year=0&month=0&cweid=0&order=1&trc=19379&sha=e3bb5586965f5a13bfaa78233a10ebc3f9606d12.

[14] diagrams.net. Diagrams. https://www.diagrams.net/.

[15] Adam Doupé, Weidong Cui, Mariusz H Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1205–1216, 2013.

[16] TeamViewer Germany GmbH. Teamviewer. https://www.teamviewer.com/.

[17] Google. Withgoogle: Content security policy. https://csp.withgoogle.com/docs/strict-csp.html.

[18] Peter Leo Gorski, Luigi Lo Iacono, Stephan Wiefling, and Sebastian Möller. Warn if secure or how to deal with security by default in software development?. In *HAISA*, pages 170–190, 2018.

[19] Daniel Hausknecht, Jonas Magazinius, and Andrei Sabelfeld. May i?-content security policy endorsement for browser extensions. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 261–281. Springer, 2015.

[20] Ben Hayak. Same Origin Method Execution (SOME). *Online at http://www.benhayak.com/2015/06/same-origin-method-execution-some.html*, 2015.

[21] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

[22] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[23] Iulia Ion, Niharika Sachdeva, Ponnurangam Kumaraguru, and Srdjan Čapkun. Home is safer than the cloud! privacy concerns for consumer cloud storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security*, pages 1–20, 2011.

[24] Internet Security Research Group (ISRG). Let's encrypt. https://letsencrypt.org/.

[25] Markus Jakobsson, Zulfikar Ramzan, and Sid Stamm. Javascript breaks free. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.3195&rep=rep1&type=pdf*.

[26] Burke Johnson and Larry Christensen. *Educational research: Quantitative, qualitative, and mixed approaches.* Sage, 2008.

[27] Ruogu Kang, Laura Dabbish, Nathaniel Fruchter, and Sara Kiesler. "my data just goes everywhere:" user mental models of the internet and implications for privacy and security. In *Eleventh Symposium On Usable Privacy and Security ({SOUPS} 2015)*, pages 39–52, 2015.

[28] Amit Klein. Dom based cross site scripting or xss of the third kind. *http://www.webappsec.org/projects/articles/071105.shtml*, 2005.

[29] Klaus Krippendorff. *Content analysis: An introduction to its methodology.* Sage, London, 2004.

[30] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel von Zezschwitz. "If HTTPS Were Secure, I Wouldn't Need 2FA"-End User and Administrator Mental Models of HTTPS. *IEEE Security & Privacy*, 2019.

[31] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006.

[32] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction.* Morgan Kaufmann, 2017.

[33] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *CCS*, 2013.

[34] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[35] Calendly LLC. Calendly. https://calendly.com/.

[36] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 297–313, 2018.

[37] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[38] Mozilla Development Network. Csp: frame-ancestors. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors.

[39] Open Web Application Security Project (OWASP). Owasp top 10 web application security risks 2017. https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS).

[40] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 653–665, 2016.

[41] Phil Ringnalda. Getting around IE's MIME type mangling. http://weblog.philringnalda.com/2004/04/06/getting-around-ies-mime-type-mangling.

[42] David Ross. Happy 10th birthday cross-site scripting. *Online at https://blogs.msdn.microsoft.com/dross/2009/12/15/happy-10th-birthday-cross-site-scripting/*, 2009.

[43] Sebastian Roth, Michael Backes, and Ben Stock. Assessing the impact of script gadgets on csp at scale. 2020.

[44] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *NDSS*, 2020.

[45] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[46] Dolière Francis Some, Nataliia Bielova, and Tamara Rezk. On the content security policy violations due to the same-origin policy. In *Proceedings of the 26th International Conference on World Wide Web*, pages 877–886, 2017.

[47] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *International Conference on World Wide Web (WWW)*, 2010.

[48] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. 2019.

[49] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. Who's hosting the block party? studying third-party blockage of csp and sri. In *Network and Distributed Systems Security (NDSS) Symposium 2021*, 2021.

[50] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1419–1430, 2015.

[51] Anselm Strauss and Juliet M Corbin. *Grounded theory in practice.* Sage, London, 1997.

[52] Michael Sutton. The dangers of persistent web browser storage, 2009.

[53] European Union. EU Commission Recommendation (2003/361/EC). *Online at https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32003H0361*, 2003.

[54] W3C. CSP 1.0. *Online at https://www.w3.org/TR/CSP1/*, 2015.

[55] W3C. CSP Level 2. *Online at https://www.w3.org/TR/CSP2/*, 2016.

[56] W3C. CSP Level 3. *Online at https://www.w3.org/TR/CSP3/*, 2016.

[57] GitHub W3C webappsec csp. Issue 7: Csp: connect-src 'self' and websockets. https://github.com/w3c/webappsec-csp/issues/7.

[58] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In *CCS*, 2016.

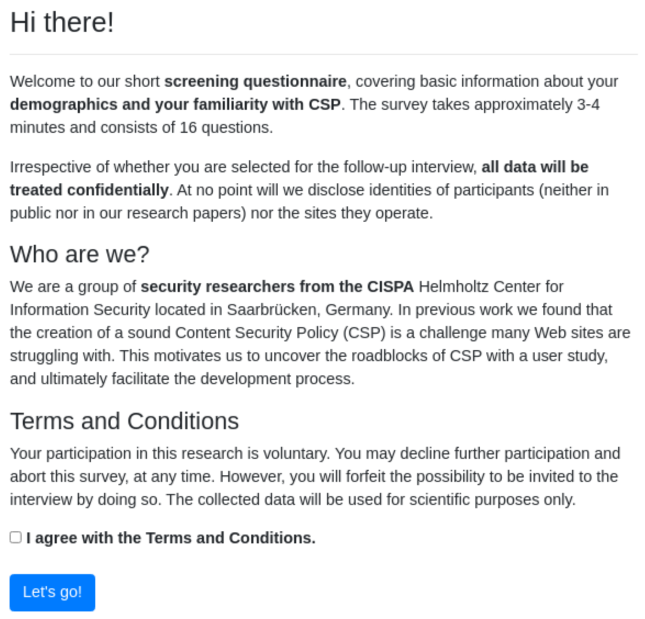
[59] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is CSP failing? Trends and challenges in CSP adoption. In *RAID*, 2014.

A SCREENING QUESTIONNAIRE:

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

A.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*



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 6: The landing page of our screening questionnaire.

- 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?

A.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 ...

A.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) [53]*
- Is the Web presence your company's main business?
- Email address *Note: Mandatory for contact reasons.*

B INTERVIEW PROTOCOL:

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

B.0.1 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.

B.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.)

B.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?

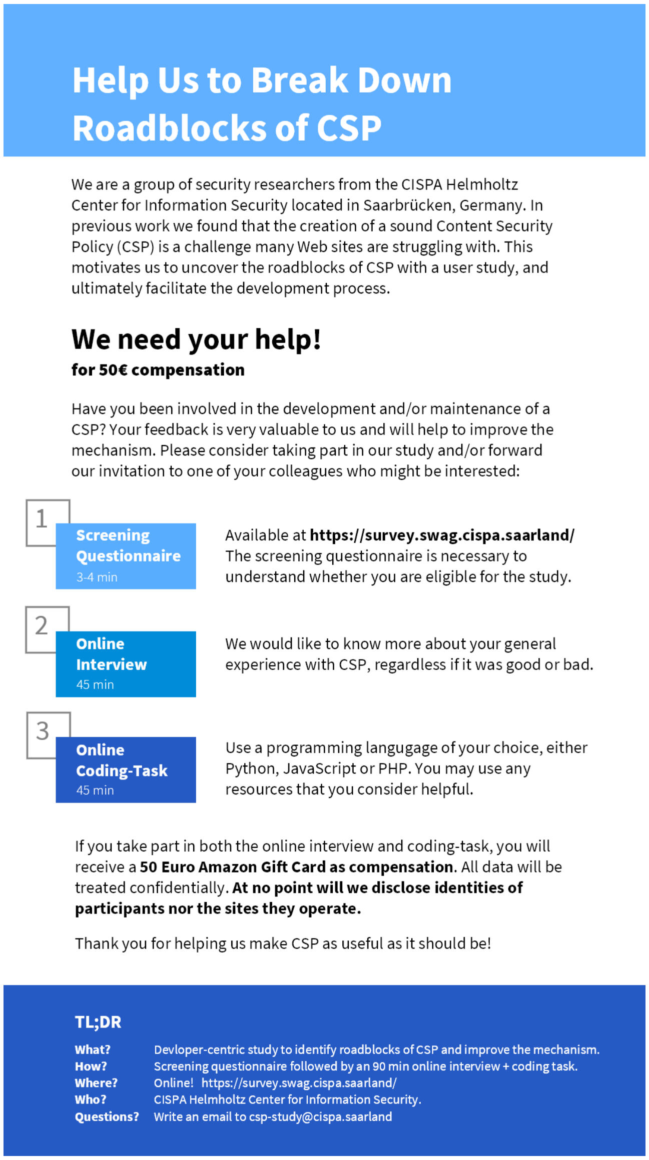
Ask all Groups: Have you used any tools / consulting that helped you to create the Policy? If yes, which ones: ...

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

C OWASP POSTER:

D LINKEDIN ADVERTISEMENT:

E RECRUITING MAIL:



Help Us to Break Down Roadblocks of CSP

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.

We need your help!

for 50€ compensation

Have you been involved in the development and/or maintenance of a CSP? Your feedback is very valuable to us and will help to improve the mechanism. Please consider taking part in our study and/or forward our invitation to one of your colleagues who might be interested:

1 **Screening Questionnaire** 3-4 min — Available at **https://survey.swag.cispa.saarland/** The screening questionnaire is necessary to understand whether you are eligible for the study.

2 **Online Interview** 45 min — We would like to know more about your general experience with CSP, regardless if it was good or bad.

3 **Online Coding-Task** 45 min — Use a programming langguage of your choice, either Python, JavaScript or PHP. You may use any resources that you consider helpful.

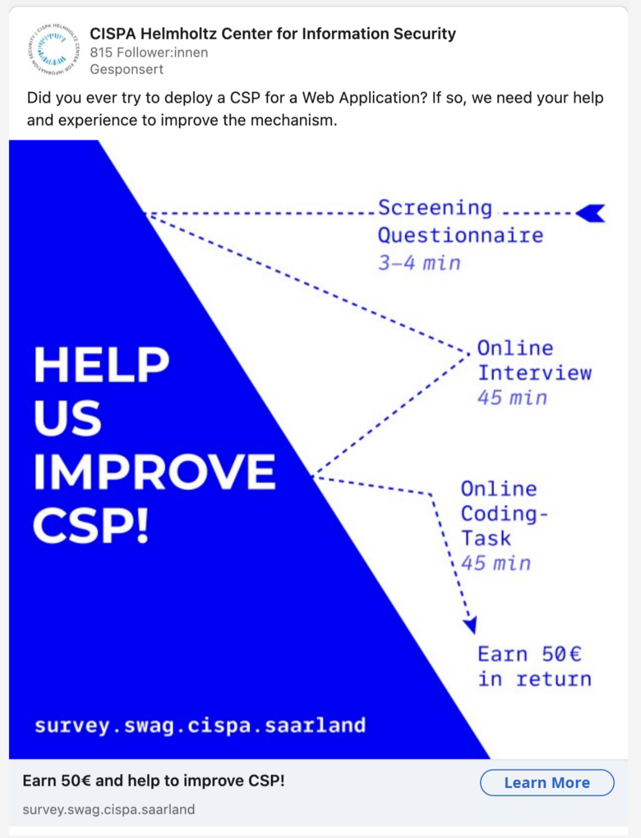
If you take part in both the online interview and coding-task, you will receive a **50 Euro Amazon Gift Card as compensation**. All data will be treated confidentially. **At no point will we disclose identities of participants nor the sites they operate.**

Thank you for helping us make CSP as useful as it should be!

TL;DR

What?	Devloper-centric study to identify roadblocks of CSP and improve the mechanism.
How?	Screening questionnaire followed by an 90 min online interview + coding task.
Where?	Online! https://survey.swag.cispa.saarland/
Who?	CISPA Helmholtz Center for Information Security.
Questions?	Write an email to csp-study@cispa.saarland

Figure 7: Poster that was used by the OWASP to advertise our study.



CISPA Helmholtz Center for Information Security
815 Follower:innen
Gesponsert

Did you ever try to deploy a CSP for a Web Application? If so, we need your help and experience to improve the mechanism.

HELP US IMPROVE CSP!
Screening Questionnaire 3–4 min
Online Interview 45 min
Online Coding-Task 45 min
Earn 50€ in return
survey.swag.cispa.saarland

Earn 50€ and help to improve CSP!
survey.swag.cispa.saarland

Figure 8: Advertisement that was used in our LinkedIn campaign.

```
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 $MONETARY_UNIT (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 address.

F 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_NAME

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

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

Table 1: Demographics of the 11 participants that completed the survey. (One participant only took part in the interview.)

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

Table 2: Drawing task results with concepts and number of participants. (One participant did not draw an XSS attack.)

```
Footer with full name, exact address, phone number, and email address.
```

G DEMOGRAPHICS OF PARTICIPANTS

H DATA FROM THE DRAWING TASK

I 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
```