To hash or not to hash: A security assessment of CSP's unsafe-hashes expression

Peter Stolz ‡, Sebastian Roth †, and Ben Stock †
† CISPA Helmholtz Center for Information Security
‡ Saarland University & Bitahoy GmbH
peter.stolz@bitahoy.com
{sebastian.roth, stock}@cispa.de

***Abstract*—More and more people use the Web on a daily basis. We use it for communicating, doing bank transactions, and entertainment. This popularity of the Web has made it one of the main targets of attacks, most prominently Cross-Site Scripting (XSS). To mitigate the effect of those attacks, the prevalence of the Content Security Policy (CSP) is increasing. Such a policy allows developers to control the content that should be allowed on their Web applications precisely. Because this content includes JavaScript (via the script-src directive), it can also be an effective tool to mitigate the damage of markup injections such as XSS. Developers can specify fine-grained policies for scripts to only allow trusted third parties and disallow the usage of functions like `eval` and its derivatives that directly execute strings as code. As the whole Web is still evolving, so is CSP. The experimental source-expression `unsafe-hashes` aims to ease the adoption of secure CSPs, by allowing trusted scripts to be used as inline event handlers for HTML tags, which is currently only possible by blindly allowing all inline scripts to be executed. Our goal is to analyze if this expression is able to improve the security of a Web application or if it mainly provides a false sense of security because it still enables attackers to bypass the CSP. We built an automatic crawler utilizing dynamic JavaScript analysis using taint tracking and forced execution to detect security vulnerabilities of inline event handlers. This crawler visited 753,715 unique URLs from the Alexa Top 1,000 domains up to a maximum of 500 URLs per domain. We collected a total of 735,105 individual event handlers, where 443 of those had attribute values that flow into a dangerous JavaScript sink. Our manual analysis of the event handlers revealed that 370 of those handlers on 34 different domains are still vulnerable in presence of a CSP that contains the `unsafe-hashes` expression. We show that attackers can exploit these flows with only partial injections, such as adding new attributes to existing tags in most cases and discuss the impact of our findings on the future of the CSP standard.**

I. Introduction

Cross-Site Scripting (XSS) is still one of the most common vulnerabilities in modern web applications [24]. An attacker injects data into a page that is misinterpreted as code allowing the attacker to steal credentials, hijack sessions, or perform state-changing actions on behalf of the user like issuing bank transactions or sending emails. In order to get more fine-grained control over the JavaScripts that are executed in the context of a Web site, the operator of the site can deploy a Content Security Policy (CSP) via an HTTP header or HTML meta tag. By controlling the origins from which content can be loaded, e.g., of scripts, the attack surface of markup injections can be minimized. However, research has shown that the vast majority of policies deployed in the wild are trivially-bypassable because inline scripts are allowed via the `unsafe-inline` keyword, wildcards are used in the allow-list, or other insecure practices [39, 30]. However, some practices in HTML require the usage of those insecure source expressions. For example, inline event handlers only execute if `unsafe-inline` is present. Steffens et al. [37] have shown that many third-parties are injecting markup that contains inline events handlers, which forces the first party to deploy a trivially bypassable CSP to not lose functionality. In addition to that Roth et al. [31] also confirmed that third-party behavior is one of the major roadblocks for secure CSP deployment. In order to get rid of the inline event handler problem, `unsafe-hashes` [8] has been proposed as an addition to the CSP standard. This source expression allows the usage of inline event handlers (e.g., `onerror=alert("Error")`) when the hash of the event handlers code is present in the allow list for script content. In theory, this should be more secure than allowing all inline code via `unsafe-inline` because an attacker can not just inject his script directly but has to reuse already present event handlers to trigger unintended behavior. That is possible because its script code hash stays the same no matter where you use the event handler. Notably, the CSP standard does not recommend `unsafe-hashes` for modern web applications, as it may expose interesting capabilities, like issuing transactions in online banking. Although this might indeed happen, those cases are hard to identify automatically because the function itself might not reveal the state-changing action that might happen on the server-side. Therefore, our attacker model considers XSS or DOM manipulation by injecting an attacker-crafted HTML element with the allowed event handler. Additionally, the allowed handlers can be run as a standalone script, as the hash for event handlers is not treated differently from the hash of a script tag. At the time of writing, only chromium-based browsers support the experimental `unsafe-hashes` expression, while other major browsers such as Firefox or Safari are not supporting the expression. We want to help in understanding if real-world event handlers indeed suffer from those code-reuse attacks or if using `unsafe-hashes` instead of `unsafe-inline` might be the better option.

To answer this research question, we make the following contributions throughout this work:

- We utilize dynamic JavaScript analysis using taint tracking and forced execution, similar to PMForce [35], to detect if an XSS attacker could reuse allowed event handlers to trigger script execution in case of real-world event handler collected from the Alexa Top 1,000 domains from the 30th of August 2021 domains.
- We discuss the impact of our results on the ongoing discussion about the unsafe-hashes expression and elaborate on other mitigations for this problem that might be possible.

II. Background & Related Work

This section describes the technologies used in this work as well as related work. In particular, we outline Cross-Site Scripting (XSS) and explain the Content Security Policy (CSP) as a mitigation technique for this attack. Additionally, we present related work on JavaScript analysis, especially PMForce created by Steffens and Stock [35], which analysis approach is also used in this work.

A. Cross Site Scripting

Cross-Site Scripting (XSS) has been one of the most prevalent security vulnerabilities for over 20 years[4]. Since its initial discovery in 1999 [29] numerous papers showed different dimensions and attack vectors that can lead to this attack [28, 21, 19, 38, 32, 17, 22, 18, 16, 23, 36].

In essence, this content injection vulnerability allows an attacker to inject and execute their malicious code into a vulnerable Web site. This data is then misinterpreted as benign code and is executed inside the victim's browser context, which effectively bypasses the Same-Origin Policy (SOP). The adversary now has the same privilege as scripts that originate from the vulnerable site itself. It often is sufficient to exfiltrate the victim's cookies to the attacker's endpoint, to impersonate the user. In order to avoid this, authenticating session cookies should set the HttpOnly flag to prevent this session from being stolen.

The root cause for XSS is the misinterpretation of data and code. This can happen when the user input is not properly sanitized. As HTML is just a markup language, an attacker may break out of the context like an attribute name or plain text when given the opportunity to inject certain characters. An example of a reflected server-side XSS vulnerability in a python flask [5] server code is depicted in Listing 1.

Here the argument name that we supply to the web application is just echoed back by the webserver. Thus, visiting this site with a URL like http://example.com/?name=<script>alert(1)</script> opens an alert box. This is an elementary example of reflected server side XSS. There is also reflected client-side XSS where code like document.write("Hello, "+ location.hash); is executed inside the victim's browser. Location.hash refers to the part of the URL that comes after the # that is not sent along to the server. In these cases, the payload resides inside the URL and is therefore not persistent. When a vulnerable server builds a response based

```python
@app.route("/")
def hello():
        return f"Hello, {request.args.get('name')}!"
```

Listing 1: Example python flask code of an reflected server-side XSS vulnerability.

on a malicious database entry, it is called persistent server-side XSS. Cookies with a dangerous value that flow into, e.g., eval unescaped are an example of persistent client-side XSS.

The above-mentioned examples did not need a break out of context. When the injection point is inside a tag, the attacker may need to end that tag and then start a script tag. If it is inside a string, the payload usually starts by closing that string.

In this work, we focus on XSS that can be triggered by injecting a malicious tag that reuses a vulnerable event handler, as the handler code is only called when the specified event is triggered.

B. Content Security Policy

XSS is the execution of attacker-created scripts in the context of a vulnerable Web site. The Content Security Policy (CSP) is a mitigation for this kind of behavior. It enables the developer to explicitly allow certain trusted resources and disallow certain vulnerable JavaScript constructs like eval. The browser checks if the Content-Security-Policy header is present that contains the specified policy and enforces it. This means if the browser does not support it, users of the Web site are not protected. To ease deployment of a CSP, Web servers can specify the policy in the Content-Security-Policy-Report-Only header. This allows developers to receive violation reports if the specified policy was too strict or XSS was exploited.

1) Version 1: The first version of CSP[10] was proposed in 2010[33], adapted in 2012, and had minimal capabilities compared to more recent versions. Version 1 allows you to specify the source for web content for scripts, images, styles, fonts, objects, media, iframes, outgoing connections, and a default fallback if a type is not specified. The origin be can set to:

- Domains that may contain a path
- Domains containing a wildcard to allow certain subdomains
- Just https to enforce encrypted communication

If the script-src (or its fallback default-src) is specified, inline scripts and inline events are forbidden and will not be executed. The only way to enable those features again was, using the unsafe-inline expression in the directive. That means each inline script needs to be hosted on an allowed domain and then included using the src attribute for a script tag. Similarly also includes event handlers, as they are basically inline scripts, need to be programmatically added in those external scripts. Although there are tools that approach the automation of this task [14, 25], the vast majority of all CSP in-the-wild still contain unsafe-inline making

them trivially bypassable by an attacker [40, 39, 12, 30]. In addition to that string-to-code functions like `eval` are disallowed by default and need to be explicitly re-enabled with the `unsafe-eval` expression.

2) Version 2: As using `unsafe-inline` defeats the purpose of CSP as an XSS mitigation and refactoring Web sites to conform to these requirements requires massive engineering effort, CSP Level 2 [11] introduced nonces and hashes to allow specific inline scripts to be executed. Alongside with new directives such as `base-uri`, `form-action` and `child-src` this change should ease the deployment of CSP and give the operator even more fine-grained control over their resources. The inline scripts now do not need to be extracted and hosted somewhere, but the hash of the script's source code may be added to the CSP to allow it. Also, nonces can be used that are specified with the `nonce` attribute of script tags, which, however, requires the CSP and the source code always to carry a fresh nonce.

3) Version 3: A common practice in the Web is that one can dynamically add content, such as scripts, to a Web site. Especially in the context of advertisements, dynamically added new script hosts are common [37]. This practice, however, causes massive effort in maintaining CSPs because those dynamically added new scripts need to be allowed in the policy. In order to ease this the current version 3[41] of CSP additionally added the `script-dynamic` expression [39], which enables allowed scripts to propagate their trust to their new added scripts. If the new expression is present, scripts have to contain a nonce or a hash, and other source expressions like domains are ignored. This means that the allowed scripts can load all the necessary scripts they need without constraints. One exception is that the added scripts can not be parser-inserted. Thus, they can not be added by functions like `document.write`, but by functions like `appendChild`.

4) unsafe-hashes: The focus of this work `unsafe-hashes` was not mentioned before because it is not part of a standard yet. It allows developers to allow event handlers by adding `unsafe-hashes` to their CSP and adding the hash of their event handler to it. This could be used to adapt CSP for Web sites more efficiently, as they can just allow all used event handlers without needing major refactoring. As the hash does not rely on the location of the event handler, it can be reused anywhere on any event. An attacker may be able to inject a malicious tag where the event handler misinterprets data as code and executes it either by evaluating it or by appending it to the document. Firefox implicitly supports this behavior because it treats inline event handlers as regular scripts.

C. JavaScript Analysis

The usage of taint tracking[13, 20] to trace how a particular variable or object modifies the program state can ease the analysis of code to find vulnerabilities.

Lekies et al. [22] modified Chromium's JavaScript engine to taint attacker-controllable inputs, such that they can discover flows to dangerous functions that lead to XSS. Using this approach, they were able to identify 6,167 unique vulnerabilities among the Alexa top 5,000. While that approach offers a significant performance benefit over taint tracking directly in JavaScript, the major drawback is the complexity of modifying the JavaScript engine itself. Therefore Steffens and Stock [35] used the dynamic code analysis tool Iroh [1] to investigate the exploitability of postMessage event handler directly in the JavaScript context of the web application. They used the browser instrumentation framework puppeteer[7] to collect `postMessages` handlers from the Tranco [27] top 100,000 domains. By combining force execution with the taint tracking, they also investigated program paths that would not be reached and collected the necessary constraints to reach the vulnerable code. After that, constraint solving was applied to determine the satisfiability of reaching the sink along with their exploitability. This yielded 251 unique potentially exploitable data flows, of which the authors automatically verified 111. Notably, the authors open-sourced their code [6], which is why we rely on their dynamic analysis approach to collect the flows from our event handlers to the dangerous JS sinks.

Besides these dynamic analysis approaches, there exist sophisticated static analysis methods that are usable for conducting such research as well. Browser extensions were analyzed to discover suspicious flows[26, 34]. Having no runtime environment requires the reconstruction of program states through parsing the JavaScript code. The code is parsed into an Abstract Syntax Tree (AST). A graph is populated using the AST containing data and control flows. In addition to that pointer, analysis was used to determine the source and destination of variable changes. That knowledge was then used to discover flows ending in suspicious actions by the browser extension[15].

III. Methodology

In this section, we explain our Attacker Model as well as technical details and implementation of the crawler and the analysis.

A. Attack Model

As we want to evaluate the security of `unsafe-hashes`, we define our attack model around it. Therefore the pages we visit have a hypothetical CSP in place that allows all of their inline event handlers and inline scripts. The attacker has an injection point somewhere inside the HTML document where he can input unsanitized data; the typical setting for XSS. Even though he could inject just a script tag, the CSP would forbid executing it. Hence the attacker tries to reuse existing allowed event handlers to execute a malicious payload in the end. This can be done by injecting a new tag with his malicious payload. Since allowed event handlers can be reused on arbitrary events, it is relatively easy to trigger event handlers for arbitrary tags. There are even interactive cheat sheets that show which events can be triggered with and without user interaction for each tag [9]. Therefore executing the handler is not a problem, and the main challenge is crafting a tag that lets you partially run JavaScript or control input to particular sinks.

B. Implementation

The following subsections describe the architecture of our analysis pipeline and how it is used to find potentially vulnerable event handlers. We begin by describing our JavaScript analysis approach and then describing how to integrate it into an automated framework.

1) Analysis technique: Here we go into details on how the handlers were found, what method of analysis we used and what benefits and limitations they offer.

Scope: There are multiple ways a handler can influence the web application state. For this work we defined the following behavior as security critical:

1) Functions that directly execute code, namely `eval`, `setTimeout` and `setInterval`.
2) Functions that modify the raw HTML of the page, namely `document.write`, `insertAdjacentHTML`, `$.html`, Even though this may not lead to XSS in our attack scenario, since we have a hypothetical CSP, the attacker can still inject new HTML code at other positions. The jQuery variant, however, directly evaluates script tags, which could be exploited if `unsafe-eval` is allowed in the CSP config.
3) Functions that access the cookies through using `document.cookie` are also interesting for this work because the cookie itself may lead to another vulnerability [36]. Cookies are often not considered user input and, therefore, sometimes not correctly escaped when used in the application. This could lead to stored XSS vulnerabilities, which are hard for servers to detect. `LocalStorage` belongs to the same category and is monitored as well.

2) Dynamic JavaScript Analysis: The large-scale analysis relies on automated toolchains to find the desired behavior. Since JavaScript is a very dynamic language, it is a tough problem to analyze JavaScript statically. Although some advanced approaches exist that utilize Control Flow Graphs and pointer analysis [15] we decided to rely on dynamical analysis. This has the advantage that we can analyze the JavaScript as we collect it inside the context of the web page instead of recreating this setting during static analysis. Hence, our infrastructure does the analysis inside a real Chromium Browser instructed by puppeteer [7]. Our attack model assumes that an HTML tag can be injected to utilize a potentially allowed handler. This means that we want to find out if we can get code execution when using existing event handlers. We call an event handler with a special event object as a parameter and can define this as our source object.

Taint Tracking In order to find these flows, we make use of the dynamic JavaScript code analysis framework Iroh [1]. With code as input, Iroh builds a so-called stage where we can register hooks for events like function calls, variable assignments, if cases, and so on. This is done by parsing the JavaScript code and calling the functions to hook it.

```
1 // before instrumenting the code
2 function onclick(event) {
3     vuln(event)
4 }
5 // after instrumenting the code
6 const $$STx1 = Iroh.stages["$$STx1"];
7 var $$frameValue = void 0;
8 $$STx1.$49(3)
   ↪            // Program Enter
9 $$STx1.$48(
   ↪            // Frame Value
10     $$frameValue = $$STx1.$2(2, this,
       ↪            // Call
11         function onclick(event) {
12             $$STx1.$4(4, this, onclick,
               ↪  arguments);   // Function
               ↪  Enter
13             $$STx1.$2(1, this, vuln, null,
               ↪  [event]);  // Call
14             $$STx1.$5(4, this);
               ↪            // Function Leave
15         }, null, [ev])
16     );
17 $$STx1.$50(3, $$frameValue)
   ↪            // Program Leave
18 // example use case
19 let listener = stage.addListener(Iroh.CALL);
20 listener.on("before", (e) =>
   ↪  console.log("Calling: " + e.callee));
21 listener.on("after", (e) => console.log(
22     "Return value of " + e.callee + " was " +
       ↪  e.return
23     )
24 );
```

Listing 2: Iroh stage instrumentation

As depicted in Listing 2 the program is parsed into a syntax tree. Iroh then adds a corresponding function call around certain operations that invoke the callbacks to the registered listeners. Therefore the example snippet on line 18 can print out the called functions and their return value.

This API allows us to inspect and modify the state of the program before and after interesting operations. Now we need a way to track the flow of our input (the event that is passed to the event handler code). We, therefore, build a proxy around our event, which is the argument of the handler. A `Proxy` is a built-in object [2] that wraps around an existing object and can be used to intercept and redefine its fundamental operations. Therefore we can return new Proxies when operations involve a Proxy object by overwriting them and returning a new Proxy as a result. Said Proxy is seen as nothing special by the analyzed program, but the handlers in our stage can recognize, track and modify it. Therefore it is possible to hook function calls and report them to our database if that function is a defined sink and the argument is a Proxy meaning it is somehow tainted by our input. Furthermore, it is even possible to pick up constraints on the Proxies based on their history of operations, allowing for better analysis. The advantages of this approach are that we can run the analysis inside a normal browser and don't need to modify internal JavaScript engine

code.

This dynamic analysis would already find some vulnerable handlers, but only in program paths that are actually reached. To find potential flows to sinks like `eval` we are missing branches of the program all the time. If we consider an `if` statement that checks for the presence of an attribute on our tag, we would not execute it as it does not exist. However, our taint tracking allows us to evaluate the satisfiability of constraints from our input afterward. Hence we can force execute all branches and check later if it is even possible to reach those paths.

Forced Execution Since some conditions may need to be met in order to reach a sink, like having a custom attribute or a certain global state, we may not find that path by simply executing our staged code. We can, however, just force the execution of branches as we please since we control the conditions for them using registered handlers inside Iroh. This means that we will explore all of the programs and check later if we can reach that part somehow. A drawback of this approach is that conditions may not depend on our input at all or are nonetheless unsatisfiable. Since we can still alter the global state of the program in our staged code and we execute all paths, it is possible that we trigger unintended behavior modifying the results of the rest of our analysis. This is somewhat mitigated by replacing destructive calls like `removeChild` with NOOPs. However, this can still lead to unwanted side effects as non-obvious destructive calls may alter the global program state.

Now that we have a proper method for analysis, we need a crawling infrastructure that allows us to inject our analysis code into the web pages.

3) Crawling the Web: Since we can run the analysis inside a browser, we can use existing frameworks to inject the code into each frame of the browser. Thus we use Google's Puppeteer [7] that controls a real Chromium browser through its developer API. This has the significant benefit that we can access browser internal functionality like modifying a page's source code and even expose functions to the browser window from within our Node.js program. Therefore communication with the database or accessing the local file system can be done inside of each browser window. We, unfortunately, have a significant performance drawback through this dynamic approach compared to just statically collecting HTTP responses, but have the advantage of running inside the most popular browser that renders the page accurately, compared to requiring reconstruction of this context using the static approach.

C. Building the framework

As PMForce [35] released its analysis pipeline [6], we adapted it to fulfill our needs. This project contained the code to set up the database and perform the previously described dynamic analysis for postMessage handlers. The first problem was that recursive crawling was not implemented in the open-sourced version. This feature is essential as one would rely on third parties for subdomains and other paths without it. Therefore we implemented an additional module that collects all references within the `a` tags. Puppeteer can be used to evaluate a function inside the frames of the page, allowing to easily collect the references to other pages. These are fed into our database with additional metadata, so we can keep track of features like crawling depth.

1) Finding event handlers: As we care about all handlers, the hook needed to change that triggers when the handlers are analyzed. Event handlers that are dynamically attached by the function `addEventListener` are not interesting as they do not reside inside the raw HTML document. We can abuse Chromium's behavior to find all the event handlers that were present during the creation of the tag. This is important as only these handlers would be relevant to be allowed in the CSP corresponding CSP directive. Chromium returns `null` using `getAttribute` for handler names that were added afterwards.

```
1  <button onclick="handle_buttonclick()">
2          Click me!
3  </button>
4  <script>
5      button = document.getElementsByTagName('button')[0];
6      button.onmouseover = function(){console.log('Hover')};
7      button.addEventListener('onmousedown', function(){console.log('MouseDown')})
8      console.log(button.getAttribute('onclick')) // handle_buttonclick()
9      console.log(button.getAttribute('onmouseover')) // null
10     console.log(button.getAttribute('onmousedown')) // null
11 </script>
```

Listing 3: getAttribute behavior for different handler declarations

When running the example above in Chromium, the output in the comments is printed to the console. Firefox handles this differently, so it is a special browser quirk that enables us to distinguish these scenarios. Therefore we can query all attributes of the tag with `getAttribute`, and if the result is not `null` and the queried key is an event handler, we can pass it to our analysis function.

2) Analyzing handlers: The analysis function creates a `ForcedExecution` state of the function and calls the instrumented Iroh code within it. When the browser executes an event handler, it does not just evaluate the value of the attribute but rather creates a wrapper function around it with the `event` as the argument. Additionally, that function is bound to the event's source Element. Binding a function in JavaScript replaces the `this` object that usually points to the `globalThis` variable that is the same object as `window` inside a browser context. To mimic this behavior, we create our initial proxy event as the argument of the event handler and bind the handler to `event.srcElement` which is

another way of accessing the HTML element that is called the handler. As Iroh builds the stage from the string representation of the event handler, the information to which object the function is bound is lost. Therefore the instrumented Iroh code needed to be modified to bind the handler. Functions like `srcElement` and `currentTarget` are defined on the `srcElement` of the event to mimic a real `HTMLElement`. To ease debugging when verifying candidates, indentation was added to the existing logging of the Iroh listeners giving a better overview of the program execution.

IV. Results

With the above-described crawling infrastructure, we crawled the Alexa Top 1,000 domains, ending up in 753,715 unique URLs that used 735,105 unique inline HTML event handler on their Web sites. As crawling uses bandwidth and computing power for the server owners without gaining anything, we do not want to create significant traffic for each domain. Therefore, we limited the number of pages visited per domain to 500. Our crawling depth was set to 50. Starting URLs have a depth of zero, and when they encounter a new link, it is incremented. Hence the maximum distance from a link to the start can be 50. However, only a maximum depth of 4 was reached on 15,917 URLs. Evidently, the pages we visited contained too many URLs to reach a higher depth within the web applications due to the per domain limit.

A. Handler analysis

There are two common ways to pass the element containing an event handler to the handler function. The most common way was to pass `this` as an argument to a function. Normally `this` inside the JavaScript Browser context refers to the global `window` object, however the `this` in an HTML event handler is bound to the actual HTML element. Notably, 201,255 of the event handlers used this way to pass the tag to the handler or modify it directly and not in a helper function. Compared to that, only 71,837 used the event parameter inside the handler that contains the source element in `event.srcElement` or `event.currentTarget`. This leaves another 462,013 event handlers that do not obviously make use of the element the handler was invoked on. As JavaScript is a very dynamic language, we want to point out that this is not an exhaustive way to access the called event. If we have an inline handler defined as `handle()` that takes no arguments one can still access the event through invoking `arguments.callee.caller.arguments[0]`. This works because every function has the special object `arguments` that contains references to the caller. Usually, the arguments object is just used for functions with a dynamic attribute count, as you can iterate over the arguments object to get all the function parameters. Unfortunately, Iroh requires running JavaScript inside the `strict mode`[3]. It is a special way to disable the implicitly used `sloppy mode` and was introduced in ECMAScript 5. `Strict mode` raises some errors that would usually be ignored and additionally forbids certain JavaScript syntax like `with`. Along with some other

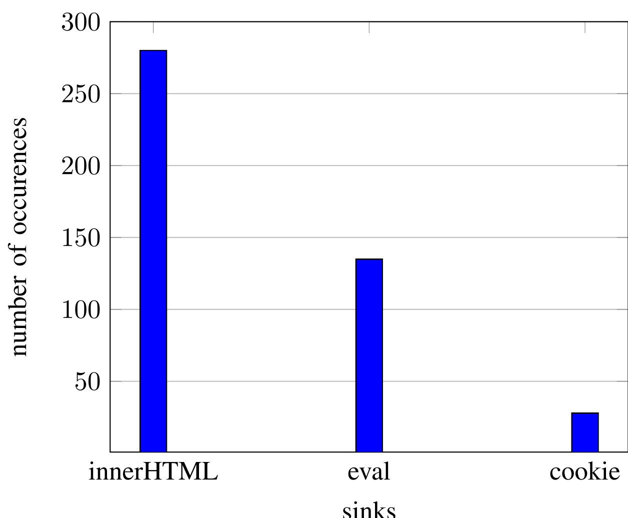

Fig. 1: Distribution of discoverd flows across distinct sinks

changes, it also forbids using the `arguments.callee` object. Our analysis pipeline would not catch handlers utilizing these flows as the analysis context is set to strict. Since this is an edge case and Iroh requires strict mode to prevent unexpected behavior and to provide a correct function trace, we can not further explore this behavior in this analysis.

Another possible way to bypass our analysis is to pass a unique identifier to the element and look it up from a global function. A basic example would be to pass the `id` of a tag as a String and then the handler uses `document.getElementById`. Since we only track access to `this` and `event`, it is impossible to find such behavior for us. However, it is unlikely that such code is common, especially among the Alexa Top 1,000 domains, due to maintainability and readability.

Due to the complexity of HTML elements, we cannot rely on the exploit generation inherited from the PMForce project, which worked because postMessages have a simpler structure.

1) Manual Analysis and Verification of Discovered Flows: Manual analysis of the handlers is a tedious process. By adding logging capabilities to our framework, we could trace the execution flow and see where the sink access occurred. In most cases, the call depth was below three allowing for quick verification. Three Web sites using anti-debugging protection, like blocking debugger commands and clearing console logs, were encountered and only partially analyzed due to time constraints.

From the 735,105 distinct handlers, our analysis pipeline identified 443 handlers from 84 different domains as potentially problematic flows. An example of such a real-world handler is presented in Listing 4. This snippet is used by major advertisement companies, such as `googlesyndication.com`, `ad-srv.net` and `adform.net`. If an attacker is able to inject a malicious tag containing the `data-onload` attribute, XSS can be

```
function onload(event) {
    eval(this.getAttribute('data-onload'))
}
```

Listing 4: Vulnerable event handler used by ad companies

```
function onblur(event) {
    deactiveMe(this);
}

function deactiveMe(obj) {
    if (obj.name.indexOf("sisDetail") == 0) {
        var objName = obj.name.substr(0, obj.name.length - 1);
        objName = objName.substr(objName.lastIndexOf("[") + 1);
        eval("var re = /\\\[" + objName + "\\\]/g");
        objName = obj.name.replace(re, "[_op_]");
        if (document.getElementsByName(objName).length > 0) {
            var hidOp = document.getElementsByName(objName);
            hidOp[0].value = 1;
        }
    }
    return;
    setTimeout("deColor4('"+obj.id+"')",100);
}
```

Listing 5: Vulnerable event handler using eval for regex

```

function onclick(event) {
    return bls(this)
}

function bls(a) {
    var b = gf("FBD");
    sd(b, blo);
    if (-1 === blo) {
    a = a.href;
    a = a.match(/\.html/) ?
        a.replace(".h", "-frame.h")
        : a.match(/\?/) ?
            a.replace("?", "-frame?")
            : a + "-frame";
    var c = document.body.offsetWidth;
    600 < c && (c = 600);
    ih(b, '<iframe id=FBF onload="siv(this, true);" src="' + a +
    '" width=' + c + " height=800 frameborder=0>webmaster@timeanddate.com</iframe>")
    }
    blo = !blo;
    0 == blo && (b = gf("FBF")) && b.scrollIntoView();
    return !1
}
```

Listing 6: Vulnerable event handler using innerHTML

triggered by reuse the body of the handler for evaluating arbitrary code.

Among the 443 handlers, we manually analyzed and verified all handlers with an exploitable source object. We also discuss some techniques used to build exploits for certain sink objects.

Across all the handlers we can observe that only flows to three distinct sinks were found. Namely `innerHTML`, `eval` and `cookie`.

As depicted in Figure 1, flows to `innerHTML` are by far the most prevalent with 280 occurrences. It is followed by `eval` having less than 50% of `innerHTMLs` with 135 occurrences. As setting `cookies` within event handlers seemed unlikely as `localStorage` seems more fitting we surprisingly found 28 flows to `cookies`. In the following subsubsections, we dive deeper into each of these categories of findings by highlighting case studies and discussing their necessities.

eval Even though flows to `innerHTML` are more prevalent than `eval` exploiting such flaws is not as impactful since the premise of our attack is the capability to inject some form of HTML. Flows to eval are the most problematic as they lead to direct XSS. It is not clear to us why web developers use eval so frequently. In most cases, this could be trivially replaced, for example, in the real-world code snippet presented in Listing 5. The last character of the name attribute is removed in line seven. If "[" is not contained in the name, the following line does nothing, and the string is thrown into eval between a variable assignment and a regex. There is no reason to do this inside eval, as global variables can be defined inside a function as well. This pattern occurred very often. Expressions are evaluated that could just be executed inline and instead are passed to eval. No manually analyzed event handler contained actual JavaScript code that would require dynamic evaluation. This just shows again that client-side security is not a priority for most Web sites, and `eval` should rarely be used. Out of the analyzed flows to eval, only seven out of 135 can not be abused.

innerHTML The impact of innerHTML flows is not as relevant as eval. Since our attacker can already inject HTML somewhere, the innerHTML could end up in a place where the attacker previously had no control over. The majority of flows to innerHTML can be bypassed by escaping it properly, like in the example depicted in Listing 6: The HTML element is passed along by the event handler to the bls function. In line 9, the tag's href attribute is selected, and in the following line, it is matched against some regex. We could simply leave out a `".html"` to not trigger the replacement. In line 13, `ih` points to `innerHTML`, and we create an iframe by concatenating the parameters with their tag names. By starting with a quotation mark, we can end the src attribute and begin writing a new tag or modifying the iframe.

This behavior is observed for most handlers and is considered bad practice. The correct way

of implementing this feature would be to use `document.createElement('iframe')` and set the properties there. An attacker could not escape out of the attribute, and the tag can be added by, e.g., `document.appendChild`.

document.cookie Most of the flows to cookies allow an attacker to influence the cookie value partially. However, sanitization was not used. If the unescaped cookie value is read and flows into other sinks like, e.g., eval, this would lead to XSS again as demonstrated by Steffens et al. [36]. Since this cookie string loses our Proxy property upon setting and we only apply taint tracking on the handler, this behavior was not further analyzed in this work due to the complexity of the web applications for further manual analysis.

Unique tricks to break out of context We found 14 unique source objects (see Table I) and we investigate how and if an attacker could exploit them.

Trivial in the context of the table means that the HTML parser does not replace characters, and therefore breaking out of context is easy. Difficult means that the parser expects the input to match a particular format like a URL or that the parser modifies that attribute's value. Setting the sink object `event.srcElement.childNodes.0.src` to escape out of context is not easy, as the src attribute is sanitized and can not be set to arbitrary values. In most cases, the src attribute contains a URL or a path. Using escape characters in that scenario does not work as chrome URL encodes them. The needed " becomes a %22. A source attribute can also be of other format like `data:image/gif;base64` followed by the base64 encoded image. For unknown reasons the parser does not check for illegal base64 characters and we can escape out of context. So one discovered example where the page constructs an HTML element via string concatenation, we can use some base64 value and after it specify the payload for the exploit like

```
<div><img src="data:image/gif;base64,
R0lGcGVzdG8K "onsmth="alert(1)"></div>.
```

Because of the `childNodes.0`, we need a tag containing at least one child, and the first elements `src` property needs to include the payload.

The `href` attribute is not parsed like `src` and can be set to arbitrary values, even though it is a reference like `src`.

Html elements cannot have the attribute `menu` as a direct attribute, as it was used in the pages we found. Even when accessing it via `getAttribute`, attributes are strings meaning you can not set the id attribute. A helper function was used inside the handler that would check if the argument is an object of a different "type", but our Proxy did not know that property cannot exist.

The `select` tag can have children called `option`. Therefore `event.srcElement.options.0.value` and `event.srcElement.options.0.text` is exploitable if the attacker can inject an additional tag and the injection point is inside a `select` tag, or he can inject two new tags.

The `event.srcElement.files.0.name` is a bit tricky. The element `input` can have the attribute `files` if it has the attributetype="file". For security reasons, the default value can not be set as Web sites could leak files from the visitor's file system. This means the user would need to upload a file with a name that breaks out of context and contains a malicious payload. As this is an implausible scenario, we do not consider it exploitable.

Flows where the sink object is `event.srcElement` can not be exploited to our knowledge as they simply return the HTML element. These make up 53 of the found flows. All of them are flows to `eval`.

V. Discussion

We found 735,105 unique event handlers, and among those 443 from 84 domains were identified as problematic by our analysis suite. Based on the source objects 62 handlers are not exploitable. For the remaining 381 handlers, the manual analysis revealed that only 11 of them are not exploitable. Thus, we found 370 event handlers on 34 different domains that are exploitable in the presence of `unsafe-hashes` in the deployed CSP.

A. *unsafe-inline vs. unsafe-hashes*

Research has shown that the vast majority of policies in the wild are trivially bypassable due to the presence of `unsafe-inline` [40, 39, 12, 30]. Steffens et al. [37] have shown on a technical level that this behavior is caused by the inclusion of third-party code, which mandates the use of unsafe-inline by adding inline events for 75% of the Web sites. This technical observation was later confirmed in a developer study by Roth et al. [31], where third-party code was documented as one of the major roadblocks for CSP deployment.

Therefore, we argue that although we found out that there are 370 event handlers, that would cause 34 different domains to be still exploitable, the advantages of `unsafe-hashes` over `unsafe-inline` would still improve the security of all other sites that are using event handlers. However, in some cases, e.g., dynamically added code with ever-changing hashes, or the sheer amount of code snippets that need to be hashed and entered into CSPs allow-list can cause the `unsafe-hashes` directive to not be usable in practice. Notably, although `unsafe-hashes` is able to improve the current situation, it will not be the savior of CSP, which is why we should also discuss and elaborate on other alternatives to ease the deployment of the security mechanism.

B. *Alternatives to unsafe-hashes*

One different mitigation of XSS would be to give each tag a NONCE similar to script tags in combination with certain CSPs. If the HTML parser encounters an element with an invalid nonce or no nonce it is discarded. The idea is to prevent attackers from injecting their own element into the page. Experiments show that the positioning of the nonce is important to provide security. Imagine a server sending a response like

source object	#domains	#occurences	satisfiyablity
event.srcElement.childNodes.0.src	1	257	needs new tag, difficult
event.srcElement.name	9	72	trivial
event.srcElement	50	53	no
event.srcElement.id	2	26	trivial
event.srcElement.innerHTML	5	12	needs new tag
event.srcElement.options.0.value	5	5	needs new tag
event.srcElement.value	3	4	trivial
event.srcElement.href	3	3	trivial
event.srcElement.menu.id	2	3	no
event.currentTarget.menu.id	1	2	no
event.srcElement.files.0.name	1	2	requires user interaction
event.srcElement.value.length	2	2	no
event.currentTarget.src	1	1	needs new tag, difficult
event.srcElement.options.0.text	1	1	needs new tag

TABLE I: Source objects and their exploitability

```
'<a nonce=1234 href=' + query['href'] + '></a>
<a nonce=1234></a>'.
```

An attacker can freely add new attributes to the tag but cannot create new HTML tags because he does not know the NONCE. In this scenario, the NONCE from the second a tag cannot be stolen as well. With a server code like:

```
'<a href=' + query['href'] + 'nonce=1234 ></a>
<a nonce=1234></a>'
```

the attacker could create a new valid tag because the nonce comes after the malicious input. The first benign a tag would have no NONCE and be discarded by the parser. Even though the tag can not be properly closed, the HTML parser will happily read it and close the tag for you. Therefore the nonce should be at the beginning of a tag. Otherwise, we do not gain much security. This would also prevent the attacker from adding additional tags as required by sink objects like event.srcElement.options.0.value. In a third scenario like this:

```
'<a nonce=1234 href=' + query['href'] + '></a>
<a href="asdf"nonce=1234></a>'
```

the attacker can hijack the NONCE from the second tag to create a new Element. With the current lax HTML parsers an injection like "213"><a test=" would end up with a new tag <a href="asdf"=""nonce= "1234">. If the nonce would be at the start, however, this could likely not be exploited. Therefore correctly noncing elements would prevent attackers from injecting new tags but still allows new attributes to be set. One could overcome this by setting default values for all the attributes, so an attacker can not overwrite them again. This, however, seems infeasible as it would drastically increase HTML file size and requires more refactoring than adding dynamic attributes. Therefore, we conclude that noncing tags would improve client-side security if placed as the first attribute. But because this requires heavy modification of web applications, it is not deployable in the real world.

C. Assisting Developers

Given our results, we currently have no secure and easy-to-use way to deploy a sane CSP while at the same time using inline events handlers. Therefore we argue that the best way to solve this problem is to work together with the Web developers and provide them with the tools to create CSP compliant code and deploy a sane CSP.

The best way is to eradicate the problem before it actually occurs. Therefore, the IDEs need to display warnings during the creation of a Web application. Developers know that using inline code will result in an application that a CSP can not protect. Even coding suggestions to automatically convert inline events to programmatically added events might be possible. This automatic conversion could also enable developers of existing applications to make their applications CSP compliant. Notably, however, all these efforts can not work without third parties getting compliant with a secure CSP at their client's applications. Notably, their incompatibility with CSP only hurts their clients but not the services themself. Thus, encouraging the third parties to make their code CSP compliant will require a "political" solution. Organizations such as the World Wide Web Consortium (W3C) or the browser vendors themself need to work together with the third-party providers on a solution for the general problem of third parties that are blocking the usage of secure CSPs.

To secure legecy Web applications, we reccommend using CSPs report-only mode to collect data about policy violations without enforcing the policy. Depending on the amount of unique violations by event handlers teh developer could then use unsafe-hashes, if only a few unique violation occur. If however, to many uniquie violations happen, we reccommend the refactoring of inline events into nonced inline scripts, such that the amount of allowed event hashed does not lead to an hard to read, and thus hard to maintain, CSP.

D. Beyond injecting event handlers

One premise of our attack model is the capability to inject HTML code into a page, but not JavaScript as a hypothetical CSP exists. The question to answer in this subsection is whether we can loosen this requirement by exploring what exactly we need to inject and what some potential mitigations could look like. One possibility is not to inject new tags but rather hijack ones when unsanitized code is injected into

attributes. This is a realistic scenario since certain attributes are dynamically filled in by the server with potentially unsanitized data. If an attacker can break out of the context with escape characters like ” or ’ we end up with a partial tag injection. When the HTML parser encounters duplicate attributes, only the first occurrence is kept, and the following attributes of the same name are removed. Capitalization of attributes is ignored when parsing and removed by the browser as well. We tested this behavior on Chromium Version 92.0.4512.0 and verified it with Chromium Version 94.0.4606.50 and Firefox 92.0. For an attacker, this means he can not replace attributes within a tag defined before his injection point. If a certain attribute value is needed and it is already set, an attacker needs to create a new tag, or exploitation will not be feasible.

E. Limitations

The conclusion we draw is based on the results of our analysis pipeline and manual verification of exploit candidates. We believe that the number of problematic handlers is higher than our measurement. There are only five false positives that we picked up, namely `event.srcElement.menu.id` as the source element. False positives, in this case, mean that the attacker cannot modify the value of the attribute that has flows to a sink. This occurred because a called function takes different types of objects as parameters and checks if these are present. Thus, our automated analysis found flows to sinks that are not necessarily exploitable by an attacker.

Additionally, we defined the `length` attribute not to be exploitable as it would always be an integer. Other than that, only the `event.srcElement` is not exploitable because it returns the element itself. As these cases can easily be filtered, we conclude that our approach has a very low false-positive rate. Chromium also gives us the benefit to detect in-HTML event handlers, meaning that we found no handlers that were added dynamically to the element. Compared to that, false negatives are likely higher. One reason for it is the force execution that may break the environment. Code that calls a variable pointing to a function could be set to a different function inside a force-executed branch that would not be possible during normal execution. Therefore we analyze a different function or no function at all, missing out on potential flows. We did observe one such case during manual analysis, leading to a false positive. However, we cannot quantify the impact of this regarding false negatives. An additional undesired side effect is the modification of the global state in a way that the standard handler would not have, leading to different analysis outcomes for the following handlers on that page. Therefore a more sophisticated approach could be used that can restore program states after force executing branches. It would yield better results regarding false positives and negatives.

Due to the internals of the Iroh framework, we could not analyze JavaScript that makes use of ECMAScript 6. While the code is still executed, we can not observe the outcome or manipulate these statements. Additionally, it required running JavaScript in the strict mode that breaks some web applications. Implementing variable tracking could be of value, as it would allow tracking how an attacker may modify the global state of the program. The sites we visited are also not crawled using the session cookies of a logged-in user. We, therefore, miss out on web applications that are hidden behind an Authentication mechanism.

PMForce, the basis for our crawler, utilized constraint solving to generate payloads for vulnerable handlers automatically. The event of a `postMessage` is more straightforward than an HTML element and was therefore discarded in our research. However, based on the source objects we found, one could create exploit templates that could verify some cases. Further work may utilize this method for larger crawls that have too many candidates to verify manually.

Lastly, we only checked supported Chromium event handlers. There are other browsers like Firefox, Safari and Opera that implement browser-specific event handlers. Therefore our hook may have missed existing event handlers, that even though they are not executed in Chromium, could be reused in our attack scenario.

Weighting the above-discussed limitations, we conclude that further work can find more problematic flows by discovering more features of web applications using sessions and looking for more handlers. Additionally, the analysis itself has some drawbacks that can be improved. Still, it is improbable that their findings would change our conclusion as the results would not vary by order of magnitude.

VI. Conclusion

Throughout our analysis, we identified 370 event handlers on 34 different domains that are exploitable in the presence of the `unsafe-hashes` expression in the deployed CSP. Although the re-usage of event handlers that are allowed via their hashes is seemingly a problem that occurs on real-world Web applications, we argue that still `unsafe-hashes` can ease the deployment of a CSP to effectively mitigate XSS. All domains that are using inline events only have the option to use `unsafe-hashes`, or to effectively allow all script executions by not deploying CSP or by deploying a trivially bypassable CSP that includes `unsafe-inline`. Thus, `unsafe-hashes` is currently the lesser of the evils that emerge during the deployment of a CSP. Notably, other possible solutions, such as the possibility to nonce all HTML nodes, might have similar issues because nonces could be stolen depending on the injection point, which results in a bypass of the deployed CSP.

The only way to securely get rid of the problem of inline JavaScript code is to not use it. To achieve that, we have to encourage developers not to use inline code from the beginning of a project, and we need to support developers during the migration process from inline event handlers to dynamically added ones. While `unsafe-hashes` is a first step towards an easier-to-use CSP, there is still work to be done to help CSP effectively mitigate XSS attacks.

REFERENCES

[1] Dynamic javascript code analyzer iroh, 2021. URL https://github.com/maierfelix/Iroh.

[2] Javascript proxy objects, 2021. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

[3] Javascript strict mode, 2021. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode.

[4] Advisory ca-2000-02 malicious html tags embedded in client web requests, 2021. URL https://resources.sei.cmu.edu/asset_files/whitepaper/2000_019_001_496188.pdf.

[5] Python3 flask web development framework, 2021. URL https://flask.palletsprojects.com/en/2.0.x/.

[6] Pmforce soucre code, 2021. URL https://github.com/mariussteffens/pmforce.

[7] Puppeteer souce code, 2021. URL https://github.com/puppeteer/puppeteer.

[8] W3c. usage of unsafe-hashes, 2021. URL https://w3c.github.io/webappsec-csp/{#}unsafe-hashes-usage.

[9] Xss cheat sheet, 2021. URL https://portswigger.net/web-security/cross-site-scripting/cheat-sheet.

[10] Adam Barth and Brandon Sterne. Content security policy 1.0. W3C note, W3C, February 2015. https://www.w3.org/TR/2015/NOTE-CSP1-20150219/.

[11] Adam Barth, Mike West, and Daniel Veditz. Content security policy level 2. W3C recommendation, W3C, December 2016. https://www.w3.org/TR/2016/REC-CSP2-20161215/.

[12] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems? evaluating the effectiveness of content security policy in the wild. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[13] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[14] 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 *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[15] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

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

[17] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.

[18] 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 and Communications Security (CCS)*, 2013.

[19] 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*.

[20] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 2018.

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

[22] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[23] 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)*, 2017.

[24] MITRE. Common vulnerabilities and exposures - the standard for information security vulnerability names.

[25] 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 *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[26] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[27] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *Network and Distributed Systems Symposium (NDSS)*, 2018.

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

[29] 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.

[30] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[31] Sebastian Roth, Lea Gröber, Michael Backes, Katharina

Krombholz, and Ben Stock. 12 angry developers-a qualitative study on developers' struggles with csp. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[32] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Network and Distributed Systems Security Symposium (NDSS)*, 2010.

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

[34] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *International Conference on World Wide Web (WWW)*, 2017.

[35] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[36] 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.

[37] 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 Symposium (NDSS)*, 2021.

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

[39] 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 *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[40] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is csp failing? trends and challenges in csp adoption. In *International Workshop on Recent Advances in Intrusion Detection*, 2014.

[41] Mike West. Content security policy level 3. W3C working draft, W3C, June 2021. https://www.w3.org/TR/2021/WD-CSP3-20210629/.