Scriptless Attacks – Stealing the Pie Without Touching the Sill

Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, Jörg Schwenk
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

ABSTRACT

Due to their high practical impact, Cross-Site Scripting (XSS) attacks have attracted a lot of attention from the security community members. In the same way, a plethora of more or less effective defense techniques have been proposed, addressing the causes and effects of XSS vulnerabilities.

As a result, an adversary often can no longer inject or even execute arbitrary scripting code in several real-life scenarios.

In this paper, we examine the attack surface that remains after XSS and similar scripting attacks are supposedly mitigated by preventing an attacker from executing JavaScript code. We address the question of whether an attacker really needs JavaScript or similar functionality to perform attacks aiming for information theft. The surprising result is that an attacker can also abuse Cascading Style Sheets (CSS) in combination with other Web techniques like plain HTML, inactive SVG images or font files. Through several case studies, we introduce the so called *scriptless attacks* and demonstrate that an adversary might not need to execute code to preserve his ability to extract sensitive information from well protected websites. More precisely, we show that an attacker can use seemingly benign features to build side channel attacks that measure and exfiltrate almost arbitrary data displayed on a given website.

We conclude this paper with a discussion of potential mitigation techniques against this class of attacks. In addition, we have implemented a browser patch that enables a website to make a vital determination as to being loaded in a detached view or pop-up window. This approach proves useful for prevention of certain types of attacks we here discuss.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security

Keywords

Scriptless Attacks, XSS, CSS, SVG, HTML5, Attack Fonts

1. INTRODUCTION

In the era of Web 2.0 technologies and cloud computing, a rich set of powerful online applications is available at our disposal. These Web applications allow activities such as online banking, initiating commercial transactions at the online stores, composing e-mails which may contain sensitive information, or even managing personal medical records online. It is therefore only natural to wonder what kind of measures are necessary to protect such data, especially in connection with security and privacy concerns.

A prominent real-life attack vector is *Cross-Site Scripting* (XSS), a type of injection attack in which an adversary injects malicious scripts into an otherwise benign (and trusted) website [11, 27]. Specifically, XSS supplies an attacker with an option of manipulating a Web page across different sites with the help of scripts. For this kind of attacks, JavaScript is typically employed as the language of choice; once the malicious script executes, it has full access to all resources that belong to the trusted website (e.g., cookies, authentication tokens, CSRF tokens). Because of their high practical impact, XSS attacks and related browser-security research have attracted a lot of attention from the security community during the recent years [20, 22, 29, 31, 32, 41, 46, 48, 51].

Preventing XSS by Preventing Executability of Code.

Following the developments and published work mentioned above, a plethora of more or less feasible defense techniques has been proposed. All these attempts have a clear goal: stopping XSS attacks [6, 26, 31, 41, 44]. In general, one can say that if an attacker manages to execute JavaScript on the target domain, then she can control the whole Web page navigated at by the victim. Therefore, a recommended mitigation strategy would be to deactivate/limit JavaScript code execution for security reasons, employing tools such as NoScript [33], Content Security Policy (CSP) [43], or, alternatively, making use of HTML5-sandboxed Iframes. This approach is reasonable if an application can function without external JavaScript, which is not always the case for modern Web 2.0 applications. Furthermore, a website increases its robustness and upgrades protection level against attacks – one example of such action being frame-busting code in order to mitigate classical clickjacking attacks [40]. As a result, limiting or disabling JavaScript synchronously disables the aforementioned protection mechanism.

Going back one step, we note that XSS attacks need to meet three preconditions guaranteeing their success:

1. *Injectability*: the attacker must be able to inject data into the Document Object Model (DOM) rendered by the Web browser.

2. *Executability*: if JavaScript (or any other code) is injected, it must be executed.

3. *Exfiltration Capability*: attacker-harvested data must be delivered to another domain or resource for further analysis and exploitation.

The fact that XSS recently replaced SQL injection and related server-side injection attacks as the number one threat in the OWASP ranking [36] indicates that these three preconditions are fulfilled by many Web applications. As observed above, several current mitigation approaches against XSS concentrate on the second precondition, mainly since injectability is often a desired feature in many Web 2.0 applications. Internet users are encouraged to contribute content and data exchange between different Web applications through the DOM is increasingly used. Thus, server- and client-side XSS filters try to remove scripts from the injected content, or, they try to modify/replace these scripts in a way that they are not executed in the browser's DOM. The typical advise reads: in case we successfully prevent injected JavaScript from being reflected or executed, a Web application can be considered secure against XSS attacks.

Note that a browser's rendering engine is often used in other tools, such as e-mail clients or instant messengers, to display HTML content. By default, scripting is disabled in these kinds of software to prevent attacks like XSS in the context of e-mail processing or instant messaging. Again, the defense approach is to temper the attacks by preventing the second precondition from occurring.

Beyond Script-based Attacks.

In this paper, we evaluate whether restricting scripting content is sufficient for attack mitigation by examining it in practice. We raise the question of an attacker actually needing JavaScript (or another language) to perform XSS attacks. The attack model that we use throughout the paper is as follows. First, we assume that precondition 1 remains fulfilled, which is reasonable in modern web applications as explained above. Secondly, we nonetheless assume that *scripting is completely disabled*, so that we can be sure that XSS attacks would not work because the precondition of executability is not met (i.e., JavaScript content will not be executed). Precondition 3 is granted by a vast majority of web applications, since extensive efforts are required to make sure that HTTP requests to arbitrary external domains are being blocked by the application itself.

It is important to note that this attack model enables an adversary to inject arbitrary markup such as *Cascading Style Sheet* (CSS) markup into a website. We show that CSS markup, which is traditionally considered to be only used for decoration/display purposes, actually enables an attacker to perform malicious activities. More precisely, we demonstrate that an adversary can use CSS in combination with other Web techniques such as inactive SVG images, font files, HTTP requests, and plain inactive HTML, all to achieve a partial JavaScript-like behavior. As a result, an adversary can steal sensitive data, including passwords, from a given site. For this work, our running example is a Web application that contains a form for entering credit card information. We introduce several novel attacks that we call *scriptless attacks*, as an attacker can obtain the credit card number by injecting markup to this page without relying on any kind of (JavaScript) code execution. We present several proof-of-concept scriptless attacks with increasing sophistication, illustrating the practical feasibility of our techniques. Neither of the discussed attacks depends on user interaction on the victim's part, but uses a mix of benign HTML, CSS and *Web Open Font Format* (WOFF [23]) features combined with a HTTP-request-based side channel to measure and exfiltrate almost arbitrary data displayed on the website.

It must be highlighted that traditional server- and client-side defense mechanism designed to prevent XSS, such as HTMLPurifier, NoScript or several other tested XSS filtering solutions, are not yet fully prepared to address our scriptless attacks. This is mainly due to the fact that we do not rely on injecting scripts or executing code.

As a further contribution, we propose new protection mechanisms against this new class of attacks. Since filtering of the attack vectors may affect the regular content of the website, we focus on eliminating conditions under which the proposed attacks can be executed, essentially preventing requests to the attacker's server. We have implemented a browser patch that gives a website a capacity to determine whether it is being loaded in a detached view or pop-up window. This approach proves useful for preventing certain types of scriptless attacks and other attack vectors.

Contributions.

In summary, we make the following three contributions in this paper:

- We describe an attack surface that is resulting from a delimitation of scripting capabilities for untrusted content in modern web applications. We show how an attacker can deploy malicious code in a heavily restricted execution context. We label this class *scriptless attacks* because they do not need to execute (JavaScript) code.

- We discuss several novel attack vectors that are sophisticated enough to extract sensitive data (our running example pertains to obtaining credit card numbers) from a given website, doing so without executing script code. The attacks utilize a sequence of benign features which combined together bring about an attack vector causing data leakage. We demonstrate that proprietary features, as well as W3C-standardized browser functionality, can be used to concatenate harmless features to function as a capable and powerful side channel attack. The described attacks relate to Cross-Site Request Forgery (CSRF) and protection CSP and they are suitable for leaking almost arbitrary data displayed on a given website. Furthermore, we identify web- and SVG-fonts as powerful tools for assisting attackers in obtaining and exfiltrating sensitive data from injected websites. We have implemented proof-of-concept examples for all attacks.

- We elaborate on the existing defense mechanisms directed at scriptless attacks, specifically referring to protection techniques such as the *Content Security Policy* (CSP) [43]. Regrettably, we also identify gaps in the CSP-based protection and cover the limitations of `X-Frame-Options` header in regards to scriptless attacks. Furthermore, we introduce a new browser feature which we have implemented for the Firefox browser in a form of a patch helpful for mitigating scriptless attacks. As an additional upshot, this feature can also assist the mitigation of several other attack techniques, such as double-click-jacking and drag & drop attacks [21].

2. ATTACK SURFACE AND SCENARIOS

In the past few years, the bar for a successful attack has been significantly raised upon the introduction of many anew and sophisticated techniques preventing attacks against web applications. We speculate that this is mainly caused by a large number of published exploits, the rise of technologies connected to HTML5, and the ever-growing popularity of HTML usage in non-browser environments, i.e., a browser's rending engine is used in a wide variety of contexts such as instant messaging tools like Pidgin and Skype, e-mail clients such as Outlook, Thunderbird and Opera Mail, entertainment hard- and software, and ultimately operating systems such as Windows 8. As a result, all these environments require protection from HTML-based attacks. This has led to the steady development of numerous defense approaches (e.g., [6, 26, 31, 41, 44]). It is additionally worth noting that the number of users that install security extensions like NoScript is growing: NoScript blocks a large range of attacks against website users by simply prohibiting JavaScript execution [33]. Consequently, attacks against web applications have become more difficult and a website that deploys latest defense techniques can already resist a large number of attack vectors.

Given all these defense strategies, we expect that attackers will thrive towards developing techniques that function in rendering contexts that either do not allow script execution or heavily limit the capabilities of an executed script. For instance, HTML5 suggests using sandboxed Iframes for untrusted content; these essentially limit script execution up to fully blocking it and they will become crucial trust tokens for future web applications. A very basic question hence comes to mind: *can an adversary still perform malicious computations in such a restricted context?*

A continuously viable attack scenario is to develop techniques to retrieve and leak data across domains by (ab)using seemingly *benign* features and concatenating them into actual attack vectors. We presume that these scenarios will gain significance in the future, as a number of defense techniques discussed above carries on growing. The attacks here-introduced are based on this exact approach and target systems that are "injectable", yet they cannot execute any JavaScript (or another language) code. Thus, we term our approach *scriptless attacks*. During the creation of these attacks, our goal was to achieve data leaks similar to the ones possible for classical XSS attacks.

The following list briefly describes some scenarios where HTML is used in browsers or browser-like software, but JavaScript is either restricted or completely disabled for security and/or privacy reasons. Our attack techniques target these scenarios because scriptless attacks enable data leakage even in such heavily restricted environments:

1. **HTML5 Iframe sandbox**: The HTML specification describes a feature that allows a website to frame arbitrary data without enabling it to execute scripts and similar active content. The so called *Iframe sandbox* can be invoked by simply applying an Iframe element with a `sandbox` attribute. By default, the sandbox is strict and blocks execution of any active content, form functionality, links targeting different views and plugin containers. The restrictions can be relaxed by adding space-separated values to that attribute content. Thus, with these settings, a developer can for instance allow scripting but disallow access to parent frames, allow form functionality, or allow pop-ups and modal dialogs. Although sandboxed Iframes are currently only available in Google Chrome and Microsoft Internet Explorer, we predict their wider adoption as the described feature appears in the HTML5 specification. A reduced version of sandboxed Iframes, labeled *security restricted Iframes*, has been available in the very early versions of Internet Explorer, for example in MSIE 6.0.

2. **Content Security Policy (CSP)**: The Content Security Policy is a proposed and actively developed privacy and security tool. Specifically, it is available in Mozilla Firefox and Google Chrome browsers [43]. The CSP's purpose is the HTTP header and meta element based restriction of content usage by the website in question; a developer can for instance direct the user agent to ignore in-line scripts, resources from across domains, event handlers, plugin data, and comparable resources such as web fonts. In Section 4 we will discuss how CSP in its current state can help mitigating the attacks introduced in Section 3.

3. **NoScript and similar script-blockers**: NoScript is a rather popular Firefox extension composed and maintained by Maone, G. [33]. Aside from several features irrelevant for this work, NoScript's purpose is to block untrusted script content on visited websites. Normally, all script and content sources except for few trusted default origins are blocked. A particular user can decide whether to trust the content source and enable it, either temporarily or in a permanent manner. NoScript was in scope of our research: we attempted to bypass its protection and gain a capacity to execute malicious code despite its presence. Let us underline that scriptless attacks have proven to be rather effective for this purpose.

4. **Client-side XSS filters**: Several user agents provide integrated XSS filters. This applies to Microsoft Internet Explorer and Google Chrome as well as Firefox with the installed NoScript extension. Our scriptless attacks aim to bypass those filters and execute malicious code despite their presence. In several examples, we were able to fulfill our objective, despite the filter detecting the attack and blocking scripture execution in reaction.

5. **E-mail clients and instant messaging**: As noted above, a browser's layout engine is usually not exclusively used by the browser itself, as several tools such as e-mail clients and instant messengers equally employ the available HTML render engines for their purposes. Mozilla Thunderbird can be discussed as a

specific example. By default, scripting is disabled in this type of software: an e-mail client allowing usage of JavaScript or even plugin content inside the mail body could induce severe privacy implications. Scriptless attacks therefore supply a potential way for attackers to execute malicious code regardless.

In summary, there are a lot of attack scenarios in which an adversary is either unable to execute scripts or she is heavily limited by the capabilities of an executed script.

3. BEYOND SCRIPT-BASED ATTACKS

In this section, we discuss the technical details of the attacks we developed during our investigation of the attack surface related to scriptless attacks. As we will see, scriptless attacks can grant a feasible solution to nevertheless exfiltrate and steal sensitive information in the contexts described in the previous section, bypassing many of the available defense solutions such as sandboxed Iframes, script-blockers (i.e. NoScript), or client-side XSS filters. For the rest of the paper, we assume an attacker has the following capabilities:

1. The attacker can inject arbitrary data into the DOM rendered by the browser – such as for instance an HTML mail body in a webmail application. This is a viable assumption for modern Web 2.0 applications that encourage users to contribute content. Furthermore, the fact that XSS attacks are ranked as number one threat according to the OWASP ranking [36] indicates that injection vulnerabilities are present in many web applications.

2. We assume that scripting is completely disabled (e.g., our user has NoScript installed or similar defense solutions are in place, preventing an attacker from code injection and subsequent execution). Note that traditional XSS attacks would not be feasible in this set-up because there is no way for executing JavaScript (or any other language) content.

We illustrate our attacks with the help of a simple web application that processes credit card numbers – it can be compared to the Amazon web store or similar websites applied with a back-end suitable for processing or delegating credit card transactions. This web application allows us to demonstrate our attack vectors in a proof-of-concept scenario. We specifically chose credit card numbers' processing for they consist of only sixteen digits such as for example `4000 1234 5678 9010`. This enables us to exfiltrate information in a short amount of time. Note that our operations are applicable to other attack scenarios as well and we will for example explain how one can steal CSRF tokens and other kinds of sensitive information with our method. Furthermore, we implemented a *scriptless keylogger* [18] that allows remote attackers to capture keystrokes entered on a web page, even when JavaScript is disabled (this vulnerability is being tracked as *CVE-2011-anonymized*).

3.1 Attack Components

The attacks described in the following sections take advantage of several standard browser features available in modern user agents and defined in the HTML and CSS3 specification drafts. We list and briefly explain these features before moving on to demonstrating how they can be combined to comprise the working attack vectors. More specifically, we show how legitimate browser features can be abused to exfiltrate content or establish side channels functional to obtain specific information from a web browser. We found the following browser features to be useful building blocks in constructing attacks:

1. **Web-fonts based on SVG and WOFF**: The HTML and CSS specifications recommend browser vendors to provide support for different web-font formats [23]. Among those are *Scalable Vector Graphics* (SVG) fonts and *Web Open Font Format* (WOFF). Our attacks employ these fonts and utilize their features to vary the properties of displayed website content. SVG fonts allow an attacker to easily modify character and glyph representations, change appearance of single characters, and diversify their dimensions. It is possible to simply use attributes such as width to assure that certain characters have no dimensions by assigning "zero width", whereas other attributes may have distinct and attacker-controlled dimensions. WOFF in combination with CSS3 allows using a feature called *discretionary ligatures* or *contextual alternatives*. By specifying those for a WOFF font, arbitrary strings of almost any length can be represented by a single character (again given distinct dimensions for eventual measurement purposes).

2. **CSS-based Animations**: With CSS based animations, it is possible to over time change a wide range of CSS and DOM properties without using any script code [14]. The properties allowing change via CSS animations are flagged by specification as `animatable`. An attacker can use CSS animations to change the width or height of a container surrounding DOM nodes that hold sensitive information, to name one example. By being able to scale the container, the contained content can be forced to react in specific ways to the dimension changes. One reaction would be to break lines or overflow the container. In case those behaviors are measurable, animation can cause information leaks based on the timing parameters of that specific behavior.

3. **The CSS Content Property**: CSS allows to use a property called `content` to extract arbitrary attribute values and display the value either before, after, or instead of the selected element [8]. The attribute value extraction can be triggered by the property value function's use `attr`. For a benign use-case of this feature, consider the following situation: A developer wishes to display the link URL of all or selected links on her website by simply rendering the content of the `href` attribute after displaying the link, but only for absolute link URLs. This is feasible by utilizing the following CSS code:

```
a[href^=http://]:after{content:attr(href)}
```

This powerful feature can also be used to extract sensitive attribute values such as CSRF tokens, password-field-values and similar data. Subsequently, they could be made visible outside the attribute context. Combining the extracted information with a font injection provides a powerful measurement lever and side channel. In fact, this combination constitutes a substantial aspect of the attacks discussed in Section 3.2 and Section 3.3.

4. **CSS Media Queries**: CSS Media Queries provide website developers with a convenient way to deploy device-dependent style-sheets [49]. A user agent can use a media query to for instance determine whether the device visiting the website has a display with a view-port width greater than 300 pixels. If this is the case, a style-sheet optimized for wider screens will be deployed. Otherwise, a style-sheet optimized for smartphones and generally smaller screens and view-ports will be chosen. The example code shown in Listing 1 illustrates the general technique; If the device visiting the website deploying this CSS snippet has a view-port width larger than 400 pixels, the background turns green; if the screen only allows a smaller view-port width, the background will be red.

Note that these different components are all legitimate and benign features within a browser. Only in combination they can be abused to establish side channels and measure specific aspects of a given website.

```
<style type="text/css">
  @media screen and (min-width: 401px){
    *{background:green;}
    body:after{content:'larger view-port'}
  }
  @media screen and (max-width: 400px) {
    *{background:red;}
    body:after{content:'smaller view-port'
        }
  }
</style>
```

Listing 1: CSS Media Queries determining screen width and deploying style-sheets accordingly

3.2 Measurement-based Content Exfiltration using Smart Scrollbars

Initially, we have decided to focus our analysis on Webkit-based browsers, since this browser layout engine is widely deployed. This includes, among others, Google Chrome and Safari, which in turn means that we cover desktop computers, laptops, iPhones and iPads, as well as the whole range of Android browsers, Blackberry, and Tablet OS devices. The Webkit project operates as open source and is known for very short development cycles and fast implementation of novel W3C and WHATWG feature suggestions. Alongside those specified and recommended features, Webkit also ships a wide range of non-standard features that are exclusively available in browsers using this particular layout engine.

One of the proprietary features enables attackers to deliver a tricky exploit, working against websites permitting submission of user-generated styles. It is possible to extract almost arbitrary information that is displayed by the website, including text content like credit card number, element dimensions, and even HTML/XHTML attribute values such as CSRF tokens used to protect non-idempotent HTTP requests [3]. The latter becomes possible once one uses the CSS `content` feature described in Section 3.1.

We have developed a demonstration exploit [17] capable of extracting detailed information about CSRF tokens; to name one example, a test showed that reading a 32 character CSRF token requires less than 100 HTTP requests.

As noted above, CSRF tokens are used by websites that wish to protect possibly harmful GET requests from being guessable. In case an attacker can discover the link to initiate modification of stored items, harm can be done by simply issuing a HTTP request to that link from a different browser navigation tab. An unguessable link – applied with a long and cryptographically safe token – prevents this kind of attack. The token has to be known in order to perform the request successfully. In an attack scenario that allows the adversary to execute arbitrary JavaScript, it is easy to extract the token by simple DOM traversal to one of the protected links and subsequent utilization of a side channel for sending the token to an off-domain location for later re-usage. But in our attack scenario, the adversary cannot execute JavaScript, and thus token extraction and exfiltration (aside from using open `textarea` elements and form submissions) is complicated. Vela et al. accomplished creating a demonstrative heavy-load CSS-only attribute reader by using attribute-selectors back in 2009 [45]. Unfortunately, this approach is unsuitable to read high-entropy 32+ character CSRF tokens.

To enable a purely CSS-based data exfiltration attack, we utilize all of the available features listed in Section 3.1, additionally combining them with one of the proprietary Webkit features. The following outline presents the steps we undertake to move from initial CSS injection to full stack data exfiltration of sensitive CSRF tokens:

1. An attacker injects a style element containing a set of CSS selectors and a `font-face` declaration. These CSS selectors choose the *CSRF-token-protected links* (CTPL) and their container elements. The `font-face` declaration imports a set of SVG fonts that has been carefully prepared: for each character that can appear in the CSRF tokens, one font file is imported. Any other character, except for the one the font has been imported for, has zero width. A single specific character per font that does have a width is applied with a distinctive width value.

2. A CSS animation block is injected alongside the aforementioned CSS. This animation targets the container of the CTPL and shrinks it from an initial large size to a specific final size. Determining this final size is crucial; the attacker needs to find out what is the right pixel size for the animation to stop to leak information about the content enclosed by the shrinking container.

3. The injected CSS contains a content property embedded by a `::before` pseudo-selector for the CTPL. This content property is applied with the value `attr(href)`. Thereby, the attacker can map the value of the `href` attribute to the DOM and make it visible. By doing so, the injected SVG fonts can be applied. For every occurrence of a CTPL, a different SVG font can be chosen. In the first selected link, the font that only gives dimension to the character `a` will be selected. For the second CTPL occurrence, the font crafted to give dimension only to the character `b` will be chosen and so on. Successively, all CTPL can be applied with an individual font, while all CTPL void of the character connected to the assigned font will have no dimension at all. Finally, all CTPL containing the characters dimensioned by the chosen font will have dimension of *character-width* × *occurrences* in pixels.

4. By decreasing the box size of the container element of the CTPL from 100% to one pixel, the attacker can evoke an interesting behavior: The box will be too small for the CTPL, so the characters applied with dimension will break to the next line. In case the box is then given a distinct height and no horizontal overflow properties, a scrollbar will appear. The moment when scrollbar appears constitutes an opening for the attacker to *determine locally* what character is being used: specific SVG font, zero width characters and scrollbars forced via pixel-precise animation decreasing the box size are sufficient for that.

Eventually an attacker can locally determine whether a character is uniquely dimensioned and therefore present in the CTPL. The only obstacle for not being able to remotely determine this character is the lack of a *back-channel* applicable for scrollbars. There is no standardized way to apply background images or similar properties to scrollbars. Webkit – an exception among all other tested browser layout engines – provides this feature. A developer can select any component of a window's or HTML element's scrollbar and apply almost arbitrary styles. This includes box shadows, rounded borders, and background images. However, our investigation showed that typical scrollbar background images are requested directly after page-load. Therefore, this property is seemingly uninteresting for timing purpose and development of a side channel obtaining information about the time of appearance or sheer existence. Nevertheless, further investigation of the Webkit-available pseudo-classes and state selectors unveiled a working way to misuse scrollbar states combined with background images for actual timing and measuring attacks. Several of the state selectors allow assignment of background images and based on that fact the specific state (such as an incrementing scroll affecting the background of the scrollbar track) have to actually occur. Here and then, the background will be loaded on *entering this CSS-selected state* and not on page-load. This allows an adversary to indeed use the measuring of scrollbar appearance for timing and side channel data exfiltration.

The CSS code sample shown in Listing 2 demonstrates one of the state selectors capable of working as a side channel. During our tests based on the Webkit scrollbar feature, determination of sensitive content took only few seconds. The victim would not necessarily notice the malicious nature if the performed CSS animation.

```
<div id="s">secret</div>
<style type="text/css">
  div#s::-webkit-scrollbar-track-piece
    :vertical:increment {
    background:red url(//evil.com?s);
  }
</style>
```

Listing 2: A working side channel: Scrollbar CSS for track-piece incrementing vertically.

We created a public test-case available at `http://html5sec.org/webkit/test` to demonstrate this side channeling attack after the issue was disclosed responsibly to the Google Chrome development team. To mitigate this attack, we recommended to treat scrollbar backgrounds and scrollbar state backgrounds equally; all background images and similar external resources should be loaded during page-load and *not* on appearance or state occurrence. These two aspects create an attack window allowing side channel attacks and appearance-probing usable for leaking sensitive data and page parameters as demonstrated in the attack explained above.

Connecting the general attack technique with the running example of having a credit card number displayed on an attacked website, the injected font will provide one ligature per digit-group of the credit card number. To create a WOFF font containing all possible groups of numbers necessary to brute-force a credit card number, an amount of no more than 9,999 or 999,999 distinct ligatures is necessary, depending on the credit card manufacturer. Every digit-group will then have a distinct width and can thus be exfiltrated through a determination as to when the scrollbar appears during the size-decreasing animation process. We successfully tested this approach in our example scenario and found that we could reliably determine and exfiltrate this information.

3.3 Content Exfiltration using Scrollbar Detection and Media Queries

During our research of the Webkit specific scrollbar data leakage capabilities, we attempted to develop a technique that can accomplish similar results in any other browser through standardized features. Additionally, extraction of single characters can turn out to be a long lasting task not optimal for effective targeted attacks. Our goal was therefore to continue research on attack techniques that have larger impact, on the whole being more efficient and more generic in comparison to the rather specific "Smart Scrollbar" approach presented above. Beware that without deep understanding of the attack surface and possible impact, as well as the involved features and adversaries, effective defense as discussed in Section 4 is complicated if not impossible.

We utilized the aforementioned technique [16] of deploying CSS Media Queries to elevate the scrollbar-based data-leakage and made it applicable to all modern browsers. It also helped separating the core problem, moving from a small implementation quirk into representing an actual design-based security issue. Media Queries, as described in Section 3.1, allow determination of a device's view-port size. Based on this judgment process, they deploy various and most likely optimized CSS files and rules. To have a scrollbar be a source for data-leakage problems as described in the aforementioned attack in Section 3.2, the attacker needs to find out when and why the scrollbars appear. More specifically, the adversary can resize elements up to a certain point and use the scrollbar to determine if the element contains a certain other element or text node of distinct value. The distinct size is the actual part where CSS Media Queries will help unveiling if a scrollbar is there or not. The following steps demonstrates how detecting scrollbar existence with CSS Media Queries works in detail:

1. A website deploys an Iframe embedding another website. A maliciously prepared CSS injection is part of this embedded website. The Iframe is set to a width of 100%, therefore fills the whole embedding window in regards to width-feature. The height of the Iframe can be set to an arbitrary value depending on what data should be leaked.

2. The embedding website is set to a specific width. This will make sure that, given the 100% width of the Iframe,

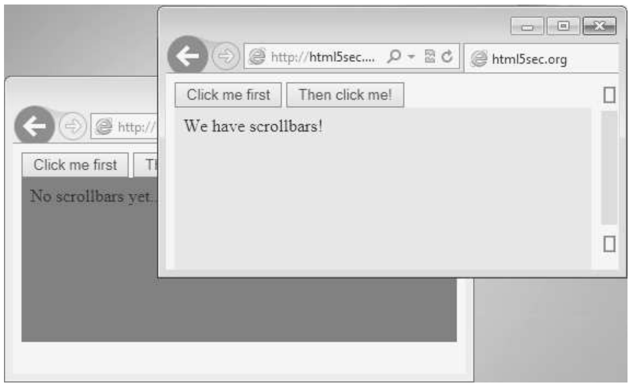

Figure 1: Decreasing vertical view-size leads to a scrollbar appear – which decreases the horizontal view-size and causes a different media query to execute

the embedded site will obey to that width and set its view-port dimensions accordingly. The framed/embedded website uses injected CSS Media Queries that deploy two states. The first state uses almost the same width as the embedding page. Consider the framing view-port having a width of 430px, then the framed website's first media query will listen for a device view-port width of 400px. A second CSS Media Query will now listen for a device view-port width of 390px. Note that once the Iframe decreases width by only ten pixel, the media query for 400px will not match anymore. At the same time, the second media query shall be activated and deploy its assigned styles, including background image requests and alike.

3. As a next step, the height of the Iframe embedding the injected site will be changed. This can be performed by a CSS animation and the Webkit-specific information leak, a script running on the website hosting the Iframe, or a manual size change in case the attacker generated a pop-up or an Iframe displayed in the edit-mode; if the hosting site displays the Iframe in edit-mode, a click-and-drag action will accomplish the resize (consider a browser-game scenario for social engineering).

   The CSS animation persists to be the most likely case not requiring any user interaction. Once the height of the Iframe is reduced, the size change will force its contents to line-break. By itself, this breaking line will generate a vertical scrollbar forced by the injected overflow-behavior or simply the window default.

   The scrollbar will occupy about 10-15 pixels and thereby reduce the view-port size from 400 to 390 or less pixels in width. This will trigger the second media query and a background image can be displayed, in parallel leaking the exact position and time of the line-break, the scrollbar appearance and thereby the width and nature of the information contained by the box. This finalizes the attack and classifies the combination of aforementioned features with CSS Media Queries as yet another potential information leak. The screenshot in Figure 1 illustrates this case.

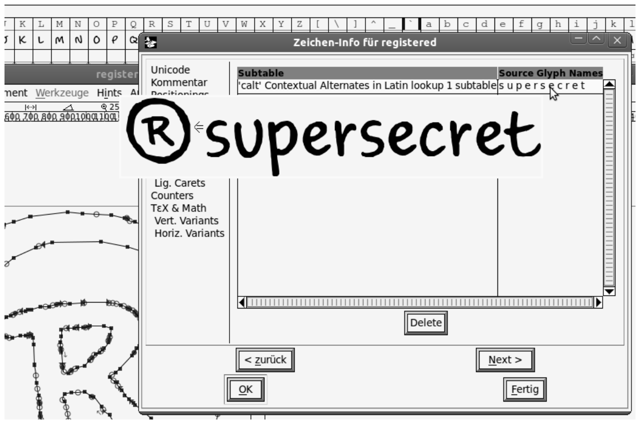

Figure 2: Assigning the contextual alternative string "supersecret" to a specific character with the help of the *FontForge* tool

Again, we created a public test-case available at `http://html5sec.org/scrollbar/test` to demonstrate scriptless determination of scrollbar existence. To initiate the test, the window has to be initially resized, and then manually reduced in height in a manner of dragging its lower boundary towards the upper boundary. Note that this can of course be accomplished automatically across domains.

To combine this attack technique with our running example, we simply use the size-decreasing pop-up window or Iframe to determine when the visible content-size is being undercut and causes the scrollbar to appear. At this moment, the overall view-size will decrease as well and cause appearance of a side channel by having the CSS Media Query initiate a HTTP request via (for example) background images. Note that this time we do not need to utilize timing attacks: The media query CSS provides detailed information on the pixel width that the scrollbars appeared at. Combining that information with the known distinct width of the contextual ligature replacing the credit card number creates a verbose and precise side channel attack.

3.4 Building Dictionary Fonts using Contextual Alternatives

To accelerate the process of identifying and determining particular strings and sub-strings on an injected website, an attacker might need a large number of different fonts and requests. The aforementioned attack samples are described as capable of exfiltrating single characters from an injected website. To be more efficient, the adversary can employ the *Discretionary Ligatures* or *Contextual Alternatives* provided by SVG and WOFF fonts [14]. By injecting a cross-domain font containing a dictionary of several hundreds of thousands of string combinations, one can greatly accelerate the detection process.

Note that the character information for each string representation can be small in size: Fonts use vector graphics and all that is necessary to deliver the detection feature of a distinct width can be contained by a path comprising of two single points. Within a single font file of one megabyte in size, an attacker can store vast amounts of contextual alternatives that depend on the nature of the represented string. As for data leakage of numerical values (for instance for being able to leak credit card numbers or similar information), the attack font can be even smaller in size and still easily discover and represent the single blocks a credit card number consists of. The tools necessary to create attack fonts are freely available for legitimate use; for creating SVG fonts containing dictionaries a simple text editor suffices. Compressing the font to the SVGZ (compressed SVG) format to be optimized in size requires a simple `gzip` implementation. For editing and abusing WOFF fonts, the free and open textttFontForge tool available at `http://fontforge.sourceforge.net/` can be easily well-used.

The results of our research signify that font-injections might actively contribute to the future attack landscape. While CSP and NoScript protect against cross-domain font injections by default, we need to monitor public font APIs use. That is because they can be abused and deliver attack fonts, bypass white-list-based filters and protection tools. By doing so, they will be breaking the trust users put into providers such as *Google Web Fonts* and *TypeKit*, both of which are free web-font deployment services.

4. MITIGATION TECHNIQUES

In this section, we analyze existing attack mitigation techniques to determine to what extent website owners and developers can protect against scriptless attacks. Acknowledging the wide range of possibilities for scriptless attacks (this publication only discusses two of potentially many more attacks' variations), we conclude that several layers of protection are necessary to effectively and holistically defend against CSS-, SVG- and HTML-based data leakage.

4.1 Content Security Policy (CSP)

CSP was originally developed by Mozilla and it is now specified as a draft by the W3C Web Application Security working group. The primary goal of CSP is to mitigate content injection vulnerabilities like cross-site scripting by determining at least one domain as a valid source for scripting code. To achieve this goal, one can use a directive like `frame-src` or `sandbox`. To provide an example, in the case of `frame-src` it is possible to let a supporting user agent check which frames can be embedded in a website. It is therefore possible to gain a fine granularity about the allowed content on a controllable website. Thus, CSP is capable of reducing the potential harmful effects of malicious code injection attacks. Note that CSP considers both arbitrary styles, inline CSS, and web fonts as possibly harmful and therefore provides matching rules.

In the context of our scriptless attacks, it would be desirable to restrict fundamental prerequisites to prevent a Web page (or rather the user) from being attacked. Therefore, we analyzed the given CSP directives with respect to the attacks we introduced in this paper. First, we have found that nearly all directives of the W3C draft, except for the directive `report-uri` for reporting policy violations, are helpful in preventing a website and its users from being affected by adversaries. The directive `default-src` enforces the user agent to execute – with one exception – the remaining directives of the draft with the given default source of the directive value. Before going into detail regarding the `default-src` influenced directives, it is important to know that pure injections with script or style sheet code into a vulnerable Web page cannot be detected by CSP. Thus, it is only possible to block the content of a file that is loaded from an external resource.

This leads to the ability of blocking malicious content that is included within an external file. A look at our attacks shows that it makes sense to use at least `style-src` and `img-src` of CSP to further reduce the attack surface. By specifying the style of the protected Web page with `style-src`, it is possible to restrict the access to undesirable CSS files. Therefore, CSS-based animations for reading DOM nodes or a usage of the CSS `content` property will no longer work in this case as an attacking tool. The same applies to `img-src`; as mentioned before, SVG files can be used to carry out scriptless attacks and intercept events, keystrokes and similar user interaction without using scripting technologies. In consequence, blocking SVG files from another site and especially another domain is recommended for achieving a better level of security. Based on our example attacks, we also propose to use `frame-src` to restrict the resources of embedded frames as well as `font-src` for limiting external font sources.

Once the possibility of increasing the security by restricting external file resources has been made clear, we are left with a following consideration: *can one restrict possible attack vectors inside the protected site?* This is exactly the case when we use `sandbox` as a directive which is not controlled or set by `default-src`. It restricts the available content based on the HTML5 `sandbox` attribute values. This directive can therefore be used to for example deactivate the execution of scripts; hence, JavaScript-based attacks will not function. What was not considered to be dangerous is scriptless code. In our case, `sandbox` is just helpful if one is facing a typical scripting attack.

In summary, we conclude that CSP is a small and helpful step in the right direction. It specifically assists elimination of the available side channels along with some of the attack vectors. In our attack model described in Section 1, CSP therefore contributes to mitigating precondition 1 and eliminating precondition 3. Nevertheless, it is insufficient to fully cover a wide array of scriptless attacks. What we recommend is to increase the range of CSP settings, so that one at least has an option to forbid the execution of style sheets or – even better – selected style sheet properties. One thing will still remain out of CSP's coverage: a behavior related to double-clickjacking [21]. The scrollbar detection we have discussed in Section 3.3 relies on a pop-up window in case the attacked website uses a frame-buster. Contrary to available frame detection and busting features, no reliable way to achieve the same security for pop-up windows and detached views is present in modern browsers. In Section 4.2 we therefore propose additional protection mechanisms against scriptless attacks and similar threats.

4.2 Detecting Detached Views

Several of the attacks we described in Section 3 can be leveraged by using Iframes and similar content framing techniques. Nevertheless, a website can easily deploy defensive measurements by simply using proper `X-Frame-Options` headers. Attackers, aware of that defense technique, have since started utilizing a different way and leverage pop-up windows and detached views to accomplish data leakage exploits and even clickjacking attacks without being affected by *frame-busting code* [40] and `X-Frame-Options` headers. Some of these attacks have been documented under the label *double-clickjacking*, while other techniques involve drag & drop operations of active content such as applets, or copy & paste operations into editable content areas across domains. Due to the extended attack surface, we want to stress that as far as modern browsers are concerned, there is no feasible way for a website to determine if it is being loaded in a detached view respective pop-up window or not.

In order to fix this problem, we created a patch for a recent version of the Web browser Firefox (*Nightly 14.0a1*, available as of April 2012), providing a possible solution to prevent the described attacks. The patch extends the well-known DOM `window` object by two additional properties: `isPopup` and `loadedCrossDomain`. Both properties are represented by a boolean value and can be accessed in a read-only manner by any website at any time. As the naming already suggests, `window.isPopup` is true only if the actual GUI window represented by the current DOM `window` object is a detached view. Likewise, `window.loadedCrossDomain` is true only if the current DOM `window` object was loaded cross-domain. These features enable websites to check their own status with the use of simple JavaScript code.

Subsequently, in case of unsafe circumstances, appropriate countermeasures can be taken. For instance, a website could protect itself against the attacks described in Section 3 by restricting itself from being loaded inside a detached view in a cross-domain manner or inside an Iframe. While the latter can already be accomplished in modern browsers out of the box (by setting the `X-Frame-Options` header to either `SAMEORIGIN` or `DENY`), the former cannot. Luckily, it becomes easily possible with our custom extension of the Firefox browser, as we have demonstrated in Listing 3 below.

```
if(window.isPopup &&
    window.loadedCrossDomain) {
  // stop loading
  window.close();
}
// continue loading
```

Listing 3: JavaScript usage example for the two additional properties exposed by the modified Firefox version

The patch consists of changes in the C++ classes *nsGlobalWindow* and *nsWindowWatcher* as well as in the interfaces *nsIDOMWindow* and *nsIWebBrowserChrome* of the Firefox code base. While the `isPopup` property could directly be implemented by examining a certain already existing internal window-flag, the introduction of the `loadedCrossDomain` property required additional code. Whenever a website tries to open a new window, this code compares the host name of the URI of the invoking website to the host name of the website-to-be-loaded (including ports). If the host-names differ, a newly introduced internal flag is set to indicate this condition, and vice versa, this flag is unset in the opposite situation. Thus the `loadedCrossDomain` property is also updated correctly in case that an already existing popup window is reused by the Firefox browser to display a new website in a popup-mode.

Allowing a website to determine whether it is being loaded in a detached view, one can mitigate several attack techniques at once. This includes several of the aforementioned scriptless attacks, double-clickjacking, drag & drop as well as several copy & paste attacks. We plan to discuss this patch with different browser development teams and evaluate how this technique can be adopted by several browsers to protect users against attacks.

4.3 Miscellanneous Defense Techniques

Scriptless attacks can occur in a plethora of variations and are often based on a malicious concatenation of otherwise benign features. We so far elaborated on ways to harden the browser and provide new levers for website owners to strengthen their applications with minimal effort. Furthermore, we shed light on how CSP helps preventing scriptless attacks by defining strict origin policies for images, fonts, CSS and other resources potentially causing information leakage by requesting data from across origins.

Zalewski et al. discussed yet another aspect of scriptless attacks in 2011, pointing at *dangling open tags* and, more specifically, elements such as button, textarea and half-open image `src` attributes to be used for data leakage [52]. These attacks are simple yet effective and require a web application and eventual HTML filtering techniques to apply grammar validation and enforce syntactical validity of user generated (X)HTML content. An open `textarea` can easily turn the rest of a website into its very own content and thereby leak sensitive data and CSRF tokens. Note that even image maps and similar deprecated technologies can be used for scriptless data leakage by sending click-coordinates to arbitrary sinks across domains. Aside from the aforementioned protection techniques and mechanisms, classic HTML content and grammar validation is of equal importance for, as Zalewski coined it, protection from attacks in the "post-XSS world" [52]. Note that this is an attacker model similar to the one we have examined in this paper. Eliminating the side channel rather than the attack vector is again of greater importance for solving this specific problem.

5. RELATED WORK

Members of the security community have granted a lot of attention to the attacks against web applications. We will now review related work in this area and discuss the novel aspects and contributions of scriptless attacks.

History Sniffing.

From a conceptual point of view, *CSS-based browser history sniffing* is closely related to our work. This technique enables an adversary to determine which websites have been visited by the user in the past. History sniffing is documented in several browser bug reports for many years now [2, 9, 39]. This method has been used in different attack scenarios [22, 24, 32, 47, 50]. In an empirical study, Jang et al. found that several popular sites actually use this technique to exfiltrate information about their visitors' browsing behavior [25]. Given the prevalence of this attack vector, the latest versions of common web browsers have implemented certain defenses protecting users from CSS-based history sniffing.

We also use CSS as part of our attacks, yet we refrain from using the actual concept behind history sniffing. More specifically, we demonstrate how CSS-based animations, the CSS `content` property, and CSS Media Queries can be abused by an adversary to access and gather specific information. As a result, our attacks also work against the latest versions of popular web browsers. One must be aware that while many documented history sniffing attacks are significantly faster when using JavaScript to exfiltrate data, these attacks can also be implemented solely based on CSS and no active scripting code, which in turn distinguishes them as scriptless attacks according to our definition.

Timing Attacks.

A more general form of history sniffing attacks in the context of web security was presented by Felten and Schneider who analyzed timing difference related to whether or not a resource is cached [15]. In a similar attack, Bortz and Boneh [7] showed how timing attacks can be implemented to recover private information from web applications. Recently, Chen et al. demonstrated different side channel leaks related to popular web sites and also based on timing information [12]. In other domains, timing attacks are a well-established technique and were used to exfiltrate information from many different kinds of systems (e.g., OpenSSL [10], SSH [42], or virtual machine environment [38]).

While timing measurements are used as part of the attacks we covered in this paper, we take advantage of other kinds of timing attacks and use this general concept to determine specific information in the context of a web browser.

Client- and Server-Side XSS Detection or Prevention.

Due to their high practical prevalence, XSS attacks have been covered by a dedicated large body of research. We will now briefly discuss different client- and server-side approaches to discovering and preventing such attacks. Note that their effectiveness is limited in the context of scriptless attacks due to their differing ground principles.

Bates et al. [4] investigate client-side filtering approaches capable of preventing XSS. They have found flaws in noXSS, NoScript and the IE8 XSS filter, and showed that some attack vectors were only activated *after* XSS filtering. In contrast to other approaches, they are inclined to put XSSAUDITOR between the HTML parser and the JavaScript engine. This design will however not prevent scriptless attacks as they do not target the JavaScript engine.

Curtsinger et al. [13] put forward a browser extension called ZOZZLE to categorize malicious JavaScript code with Bayesian classification. It remains an open question if such learning-based defense mechanisms will work against scriptless attacks.

Pietraszek et al., introduced *context-sensitive string evaluation* (CSSE), a library to examine strings of incoming user-generated data by relying on a set of meta-data [37]. Depending on the context derived from the attached meta-data, different filtering and escaping methods were being applied for the protection of the existing applications. This low-level approach is described as operational for existing applications, requiring few to no application developer implementation effort.

Kirda et al. proposed a client-side XSS prevention tool called *Noxes* [30]. By keeping the browser from contacting URLs that do not belong to the domain of the web application, this tool prevents an adversary from leaking sensitive data to his server. From a conceptual point of view, such an approach can also be used to limit what an adversary can achieve with scriptless attacks, since it prevents side channels from exfiltrating stolen information. Furthermore, the authors elaborate on the difficulties of server-side XSS detection and prevention based on the manifold of encoding and obfuscation techniques an attacker can choose from. In a similar way, we argue that scriptless attacks cannot be prevented at the server-side.

Jim et al. introduced *Browser-Enforced Embedded Policies* (BEEP) [26], a policy-driven browser extension capable of controlling whether a certain script may execute or not. More specifically, BEEP enables a user to whitelist legitimate scripts and disable scripts for certain regions of the web page. The whole concept represents another foundation for CSP [43]. Nadji et al. proposed a similar approach: *document structure integrity* (DSI) [35] ensures that dynamic content is separated from static content on the server-side, while both are combined at the client-side in an integrity-preserving way. *Blueprint* by Louw and Venkatakrishnan follows a similar approach [31]: a server-side application encodes the content into a model representation that can be processed by the client-side part of the tool. Saxena et al. presented *ScriptGuard*, a context-sensitive XSS sanitation tool capable of automatic context detection and accordant sanitation routine selection [41]. Note that all these approaches focus on preventing code scripting, which implies that scriptless attacks can potentially bypass such protection mechanisms, for we do not use dynamic content.

Heiderich et al. published on XSS vulnerabilities caused by SVG graphics bypassing modern HTML sanitizers [20] as well as DOM-based attacks detection in the context of browser malware and complex cross-context scripting attacks [19].

Martin and Lam [34] as well as Kieyzun et al. [29] introduced tools capable of automatically generating XSS and SQL injection attacks against web applications. XSSDS [28] is a system that determines if an attack is actually successful by comparing HTTP requests and responses. In recent papers, different approaches to discovering parameter injection [1] and parameter tampering vulnerabilities [5] were offered. These types of tools are not yet available for automated discovery and creation of scriptless attacks, although we expect that similar notions can be identified and applied to appropriately consistent tools' development in the future.

6. CONCLUSION AND OUTLOOK

In this paper, we introduced a class of attacks against web applications we call *scriptless attacks*. The key property of these attacks is that they do not rely on the execution of JavaScript (or any other language) code. Instead, they are solely based on standard browser features available in modern user agents and defined in the current HTML and CSS3 specification drafts. In a way, this kind of attacks can be seen as a generalization of CSS-based history stealing [22,32] and similar attack vectors [52]. We discussed several browser features useful for scriptless attacks, covering a variety of ways in which an adversary can access information or establish a side channel. Furthermore, we presented several scriptless attacks against an exemplary web application and demonstrated how an adversary can successfully obtain sensitive information such as CSRF token or user-input by abusing legitimate browser concepts. In addition, we showed that an adversary can also exfiltrate specific information and establish side channels that make this attack feasible.

While the attacks discussed in this paper presumably do not represent the entirety of ways to illegitimately retrieve sensitive user-data, we believe that the attack components discussed by us are of great importance to other attack vectors. Therefore, a detailed analysis and further elaborated investigation pertaining to possible defense mechanisms will likely yield more attack vectors. We hope that this paper spurs research on attacks against web applications that are not based on the execution of JavaScript code.

As another contribution, we introduced a browser patch that enables a website to determine if it is being loaded in a detached view or pop-up window, showcasing mitigation technique for several kinds of attacks. Within our future work, we will examine more ways for dealing with and preventing scriptless attacks.

Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research (BMBF grant 01BY1205A JSAgents).

7. REFERENCES

[1] M. Balduzzi, C. Gimenez, D. Balzarotti, and E. Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[2] D. Baron. :visited support allows queries into global history. `https://bugzilla.mozilla.org/147777`, 2002.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[4] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100. ACM, 2010.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. NoTamper: Automatic Blackbox De- tection of Parameter Tampering Opportunities in Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[6] P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2008.

[7] A. Bortz and D. Boneh. Exposing Private Information by Timing Web Applications. In *16th International Conference on World Wide Web (WWW)*, 2007.

[8] B. Bos, T. Çelik, I. Hickson, and H. Wium Lie. Generated content, automatic numbering, and lists. `http://www.w3.org/TR/CSS21/generate.html`, June 2011.

[9] Z. Braniecki. CSS allows to check history via :visited. `https://bugzilla.mozilla.org/224954`, 2003.

[10] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. In *USENIX Security Symposium*, 2003.

[11] CERT Coordination Center. Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`, 2000.

[12] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy*, 2010.

[13] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, 2011.

[14] J. Daggett. CSS fonts module level 3. `http://www.w3.org/TR/css3-fonts/`, Oct. 2011.

[15] E. W. Felten and M. A. Schneider. Timing Attacks on Web Privacy. In *ACM Conference on Computer and Communications Security (CCS)*, 2000.

[16] M. Heiderich. Content exfiltration using scrollbar detection and media queries. `http://html5sec.org/scrollbar/test`, June 2012.

[17] M. Heiderich. Measurement-based content exfiltration using smart scrollbars. `http://html5sec.org/webkit/test`, June 2012.

[18] M. Heiderich. Scriptless SVG Keylogger. `http://html5sec.org/keylogger`, June 2012.

[19] M. Heiderich, T. Frosch, and T. Holz. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. In *Recent Advances in Intrusion Detection (RAID)*, 2011.

[20] M. Heiderich, T. Frosch, M. Jensen, and T. Holz. Crouching Tiger – Hidden Payload: Security Risks of Scalable Vectors Graphics. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[21] D. Huang and C. Jackson. Clickjacking Attacks Unresolved. `https://docs.google.com/document/\\pub?id=1hVcxPeCidZrM5acFH9ZoTYzg1DOVjkG3BDW_oUdn5qc`, June 2011.

[22] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting Browser State From Web Privacy Attacks. In *15th International Conference on World Wide Web (WWW)*, 2006.

[23] D. Jackson, D. Hyatt, C. Marrin, S. Galineau, and L. D. Baron. CSS animations. `http://dev.w3.org/csswg/css3-animations/`, Mar. 2012.

[24] A. Janc and L. Olejnik. Web Browser History Detection as a Real-World Privacy Threat. In *European Symposium on Research in Computer Security (ESORICS)*, 2010.

[25] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[26] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-enforced Embedded Policies. In *16th International Conference on World Wide Web (WWW)*. ACM, 2007.

[27] M. Johns. *Code Injection Vulnerabilities in Web Applications – Exemplified at Cross-Site Scripting*. PhD thesis, University of Passau, Passau, July 2009.

[28] M. Johns, B. Engelmann, and J. Posegga. XSSDS: Server-side Detection of Cross-site Scripting Attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[29] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[30] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks. In *ACM Symposium on Applied Computing (SAC)*, 2006.

[31] M. Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy*, 2009.

[32] M. Jakobsson and S. Stamm. Invasive Browser Sniffing and Countermeasures. In *15th International Conference on World Wide Web (WWW)*, 2006.

[33] G. Maone. NoScript :: Firefox add-ons. `https://addons.mozilla.org/de/firefox/addon/722/`, July 2010.

[34] M. Martin and M. Lam. Automatic Generation of XSS and SQL Injection Attacks With Goal-directed Model Checking. In *USENIX Security Symposium*, 2008.

[35] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[36] OWASP. Top Ten Project. `https://www.owasp.org/index.php/Category:OWASP\_Top\_Ten\_Project`, Jan. 2012.

[37] T. Pietraszek and C. Berghe. Defending Against Injection Attacks Through Context-sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2006.

[38] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[39] J. Ruderman. CSS on a:visited can load an image and/or reveal if visitor been to a site. `https://bugzilla.mozilla.org/57351`, 2000.

[40] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *Web 2.0 Security and Privacy (W2SP) Workshop*, July 2010.

[41] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical report, Technical Report MSR-TR-2010-128, Microsoft Research, 2010.

[42] D. X. Song, D. Wagner, and X. Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium*, 2001.

[43] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *19th International Conference on World Wide Web (WWW)*, 2010.

[44] M. Van Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-site Scripting Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[45] E. Vela. CSS Attribute Reader Proof Of Concept. `http://eaea.sirdarckcat.net/cssar/v2/`, Nov. 2009.

[46] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[47] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks. In *IEEE Symposium on Security and Privacy*, 2011.

[48] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[49] H. Wium Lie, T. Çelik, D. Glazman, and A. van Kesteren. Media queries. `http://www.w3.org/TR/css3-mediaqueries/`, July 2010.

[50] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A Practical Attack to De-anonymize Social Network Users. In *IEEE Symposium on Security and Privacy*, 2010.

[51] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. SWAP: Mitigating XSS Attacks Using a Reverse Proxy. In *ICSE Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2009.

[52] M. Zalewski. Postcards from the post-XSS world. `http://lcamtuf.coredump.cx/postxss/`, 2011.