Honey, I Cached our Security Tokens Re-usage of Security Tokens in the Wild

Leon Trampert
leon.trampert@cispa.de
CISPA Helmholtz Center for Information Security

Ben Stock
stock@cispa.de
CISPA Helmholtz Center for Information Security

Sebastian Roth
sebastian.roth@cispa.de
CISPA Helmholtz Center for Information Security

ABSTRACT

In order to mitigate the effect of Web attacks, modern browsers support a plethora of different security mechanisms. Mechanisms such as anti-Cross-Site Request Forgery (CSRF) tokens or nonces in a Content Security Policy rely on a random number that must only be used once. Notably, those Web security mechanisms are shipped through HTML tags or HTTP response headers from the server to the client side. To decrease the server load and the traffic burdened on the server infrastructure, many Web applications are served via a Content Delivery Network (CDN), which caches certain responses from the server to deliver them to multiple clients. This, however, affects not only the content but also the settings of the security mechanisms deployed via HTML meta tags or HTTP headers. If those are also cached, their content is fixed, and the security tokens are no longer random for each request. Even if the responses are not cached, operators may re-use tokens, as generating random numbers that are unique for each request introduces additional complexity for preserving the state on the server side. This work sheds light on the re-usage of security tokens in the wild, investigates what caused the static tokens, and elaborates on the security impact of the non-random security tokens.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Web Security, CSP Nonces, CSRF, Security Tokens

ACM Reference Format:
Leon Trampert, Ben Stock, and Sebastian Roth. 2023. Honey, I Cached our Security Tokens Re-usage of Security Tokens in the Wild. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3607199.3607223

1 INTRODUCTION

The modern Web is nowadays used for a myriad of different services, from governmental services, over entertainment, up to fully-fledged office applications that are running in browsers; nearly everything can be done on the Web. Due to this importance, Web sites need to deal with security- and privacy-sensitive data of their users, which is also the reason why they are one of the primary targets for attackers. While the attack surface of a Web application is quite large, there is a plethora of different security mechanisms available to defend a site against malicious actors.

One example of such an attack is a Cross-Site Request Forgery (CSRF), where the attacker performs an action on a Web application that the victim is authenticated on without the knowledge or consent of that victim. To mitigate this attack, a per page and user unique value (a so-called anti-CSRF token) can be generated and added to each form of the delivered page. This token is then sent to the server along with any submitted form, allowing the server to verify that the request was made by the same user that visited the page and not by an attacker. Because those secrets are supposed to be random for each request and user, an attacker can not guess those and, thus, not create valid state-changing requests to the server on behalf of the user.

Another vulnerability that is omnipresent in the OWASP Top 10 Web Application Security Risks [23] is Cross-Site Scripting (XSS), where an attacker can inject malicious code in the context of a Web site, such that this code has the same access rights as benign code from the site itself. To restrict this unintended code execution, a Web site's operator can deploy a Content Security Policy (CSP) [41]. One way of marking trusted JavaScript as executable is to add a *nonce* ("number used once") to the corresponding script tag and the header-delivered policy. Because those nonces are supposed to be random for each request, attackers can not guess them and thus not execute their malicious scripts.

Notably, both of the aforementioned mechanisms (Anti CSRF Tokens & CSP Nonces) rely on random secrets per request and user to work correctly. However, generating unique random numbers for each request introduces additional complexity for the operator as they must keep state on the server side. In addition to that, many Web sites use a Content Delivery Network (CDN) [9] to decrease the server load and the traffic that is received by their infrastructure. This feature is achieved by caching certain responses from the application server (mostly referred to as *the origin server*) and serving them to multiple requesting clients without forwarding the request to the origin server. This caching affects all parts of the response, so HTML forms, meta tags, and HTTP response headers. Those, however, are the parts where the random values for CSP nonces and Anti-CSRF Tokens are stored. Thus, delivering the same value to multiple clients undermines the protection offered by those security mechanisms. A Web operator must ensure that no dynamic content, such as content with embedded supposedly

RAID '23, October 16–18, 2023, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0765-0/23/10…$15.00
https://doi.org/10.1145/3607199.3607223

random tokens, is cached. Misconfigurations of server-side caches thus may compromise the security of a Web application.

Mirheidari et al. [21, 22] investigated the dangers of Web Cache Deception (WCD) attacks, where an attacker abuses the fact that CDNs may be configured to cache content purely based on the suffix of the request URL. Thus, this kind of attack can be abused to leak session IDs, CSRF tokens, or CSP nonces, but it requires an attacker to force their victim to first visit the crafted URL to populate the cache. We, however, want to investigate if this attacker step is necessary, by determining if, even without the presence of an attacker, Web sites (accidentally or intentionally) cache security-related tokens. To do so, we measure the prevalence of reoccurring CSP nonces and Anti-CSRF tokens for the 10,000 highest-ranking sites based on Tranco [27]. In particular, we find that 10-15 percent of sites employing such security tokens suffer from reoccurring tokens. We check if nonce re-usage is due to caching behavior by checking for response headers that indicate cache hits or misses and those that indicate the age of the resource. Here we noticed that cache or CDN misconfiguration is likely one of the root causes of reoccurring tokens in the wild. Finally, we quantify the security implications of reoccurring security tokens and also investigate guides and hints provided by CDN providers to assist during the configuration. Reoccurring CSP nonces effectively disable the CSP in the case of a client-side XSS vulnerability. Similarly, reoccurring anti-CSRF tokens either degrade the user experience or may lead to developers disabling server-side token validation to allow for proper functionality.

This paper presents the following contributions:

(1) We measure the prevalence of security token reoccurrence in the wild by analyzing the 10,000 highest-ranking sites based on Tranco [27].
(2) We investigate if the security tokens' reoccurrence is likely to result from a cache misconfiguration by checking cache-related header information.
(3) We quantify the security implication of using non-random security tokens in different scenarios.

2 BACKGROUND & RELATED WORK

In this section, we provide the necessary background for our work and survey the related papers. We first introduce XSS and its mitigation, CSP, followed by a discussion of CSRF attacks and countermeasures. Finally, we provide a high-level overview of CDNs.

2.1 Cross-Site Scripting and Content Security Policy

The most basic security mechanism built into modern browsers is the Same Origin Policy (SOP), which restricts access from Web sites to only those that share the same protocol, hostname, and port. Without this policy, an attacker's JavaScript code could simply read the victim's email in the context of their logged-in Gmail session. An attacker can, however, leverage a Cross-Site Scripting (XSS) vulnerability to circumvent this restriction. With XSS, the attacker finds a flaw in the target application (e.g., Gmail) and injects their code into the page. This way, the attacker-controlled code now operates in the origin of the vulnerable application, allowing it to access anything that legitimate code could.

Cross-Site Scripting is typically categorized into two-by-two dimensions: it can be caused by either insecure server- or client-side code, and it can persistent or might originate from the request itself (also called reflected). Arguably the most impactful class of XSS is persistent server-side XSS, as it targets all users of the application. In contrast, both classes of reflected XSS require the attacker to force the victim's browser to make an HTTP request which contains the payload within the URL, e.g., in the search parameters or the so-called URL fragment (the data after the # sign). Notably, the URL fragment is not sent to the server in the request, but can instead only be accessed by the client-side code. Finally, persistent client-side XSS requires a one-time infection of persistent client-side storage (e.g., cookies or localStorage). Once this injection has occurred, the payload is retrieved on the client for each page load [36].

According to the OWASP Top 10 Web Application Security Risks [23], XSS has been one of the most prominent security issues for Web sites for multiple years. Of particular interest on the Web have been studies of client-side XSS (initially dubbed DOM-based XSS by Klein [16]), primarily given the fact that these can be confirmed without having to send attack payloads toward servers [18, 20, 25, 36]. These works showed that around 10% of the tested sites were susceptible to the different types of client-side XSS. In addition, the ever-growing variety of XSS [11, 12, 15, 17, 33**?** **?**] as well as the lacking success of the different defense approaches [4, 10, 32, 35, 39], are indicating that XSS is here to stay.

A way to mitigate Cross-Site Scripting is the usage of a Content Security Policy (CSP), initially proposed by Stamm et al. [34]. The core idea of CSP is not to eradicate XSS flaws but instead limit the attacker's capabilities through an allowlist of scripting resources [41]. A Web server can specify such a policy through HTTP response headers or in HTML meta tags. This policy is then parsed and subsequently enforced by the browser. Hence, a secure CSP can limit the attacker to a point where the XSS flaw can no longer be meaningfully exploited.

In order to mitigate XSS attacks, the site's operator has to set a *script-src* (or alternatively a *default-src*) directive. By default, this directive forbids the execution of inline JavaScript, such as inline script tags or inline event handlers, and also disables the usage of JS functions that are doing a string-to-code conversion, such as *eval*. Moreover, the directive specifies a list of allowed URLs for script inclusion. When attempting to load a script from any URL, the browser, therefore, checks if that resource is permitted through CSP before attempting to execute it as JavaScript.

The original specification of CSP was quickly shown to lack adoption in the wild [40]. One of the primary reasons for this was the inflexibility dictated on developers by CSP. In particular, given CSP's default behavior to disallow inline scripts and event handlers, developers with applications full of these were faced with a conundrum: either they would have to conduct significant refactoring to rid their application of inline scripts (notwithstanding third-party code which might interfere [35]) or they would be required to add the `'unsafe-inline'` keyword to their policy. This explicitly allows *any* inline scripts again – irrespective of whether they are developer-intended or attacker-injected.

To overcome this shortcoming, CSP Level 2 introduced the concept of hashes and nonces. With hashes, developers can specify the cryptographic hash of their intended inline scripts in the CSP.

```
Content-Security-Policy:
  script-src 'nonce-ABCDEF0123456789'
             https://example.com/;
```

Figure 1: Example CSP that uses nonces to restrict script execution

```
<script nonce="ABCDEF0123456789">
  benignJSFunction();
</script>
```

Figure 2: Example script tag with a nonce attribute

Before executing an inline script, the browser then computes the hash of its source code and only executes the code if the hash match. While this can be beneficial to enable inline scripts that rarely change (and thus rarely require re-computing the hash), it suffers from drawbacks when inline scripts contain user-specific data (e.g., the user's name). Therefore, CSP also supports the use of nonces. As the name suggests, a nonce is a *number only used once*. This nonce is sent through the CSP and is supposed to be attached to any script intended by the developer. Figure 1 shows an example of a CSP that allows scripts that carry the specified nonce.

The browser then determines for a given script (both inline and external) if the nonce provided in the nonce attribute of the script tag matches with the CSP-specified one. Figure 2 shows an example of such a script tag. If the nonce matches the one from the CSP, the code is executed. Otherwise, and only for an external script, the URL is checked against the allowlist, and if there is no match, execution is refused. Notably, nonces cannot be used to allow inline event handlers. This is only possible through hashing the content of the event handler and specifying the experimental `'unsafe-hashes'` keyword [41].

Prior work has studied CSP in depth from multiple angles. Several works have documented the status quo for a specific point in time [4, 39, 40] or have studied the evolution over time [30]. To ease the deployment of CSP for developers, Doupé et al. [10], Pan et al. [24] have proposed automated systems to generate secure CSPs and perform code rewriting. Nevertheless, as documented by Steffens et al. [35], Web applications seldomly run purely on first-party code, and third parties play a crucial role in the lack of success of CSP. As a result, secure CSPs with nonces or hashes are still uncommon in the wild. This is also caused by the complexity of the mechanism itself, which was recently documented by Roth et al. [32] through an interview study.

2.2 Cross-Site Request Forgery

Authentication on the Web is usually implemented through the usage of cookies. These small snippets of data are associated with a domain or host and sent along in the headers of an HTTP request toward matching URLs. These cookies are attached irrespective of who is the initiator of the request [1] (unless SameSite cookies are activated, which currently is only true by default for Chrome derivatives). This behavior enables a class of attacks commonly referred to as Cross-Site Request Forgery (CSRF). Here, a malicious actor causes a victim's browser to perform an unwanted action on a vulnerable site when the victim is authenticated. As an example, let us consider a simple banking application. This likely features a form with the target bank account number, an amount, and a description for the transaction. An attacker can thus simply set up their own form (on their site), for which the form's `action` field (which specifies the target URL to post the data to) points to the banking site. Upon loading the page, the attacker's pre-filled form (with their own bank details in place) can be automatically submitted through JavaScript, i.e., simulating a user clicking the submit button. If a victim is logged into their bank application and visits the attacker's page, their browser will automatically make the request to the bank, causing the unwanted action.

Other classes of CSRF attacks include login CSRF [2], where the attacker forces the victim to log into the attacker's account on some service. While this may sound counterintuitive initially, it allows the attacker to track their victim's activities, e.g., following their search history, since that history is now tied to the attacker's account. Moreover, CSRF attacks can be leveraged to exploit reflected server-side XSS flaws, e.g., in search fields, which insecurely reflect the search query.

There are different ways of mitigating CSRF attacks. First, the SameSite directive in cookies can be used to mitigate the issue (in supporting browsers). If set to `Lax`, browsers will not attach cookies to requests that originate from a different site, except for top-level navigation through safe HTTP methods such as GET. If set to `Strict`, cookies will not be sent on any type of cross-site request. However, SameSite cookies come with their own drawbacks: first, legacy browsers do not support them, making them prime targets for CSRF attackers. Second, they do not protect from a cross-origin, yet same-site attacker. For example, an attacker capable of compromising a bank's test system (at `https://test.bank.com`) can leverage this to force the victim's browsers to make requests to `https://banking.bank.com`. Third, and most importantly on the modern Web, restricting cookies to SameSite may hurt third-party business models. As an example, if SameSite cookies are enabled, like/share buttons cannot be customized to the user, as these frames are typically sub-resources loaded across site boundaries.

Given the limitations of SameSite cookies in terms of functionality on the modern Web and their lacking support in legacy browsers, the best practice to defend against CSRF is the usage of random CSRF tokens. Usually, there are two types of mitigation techniques. First, applications can rely on the so-called *Synchronizer Token Pattern* [19]. Here, the server generates a token for each user's session,

```
<form action="/login" method="POST">
  <input type="text" name="email">
  <input type="text" name="password">
  <input type="text" name="csrfToken" hidden
   value="35a762fec5e963934ce06bb16c0528af">
</form>
```

Figure 3: Example HTML form tag that includes an anti-CSRF token.

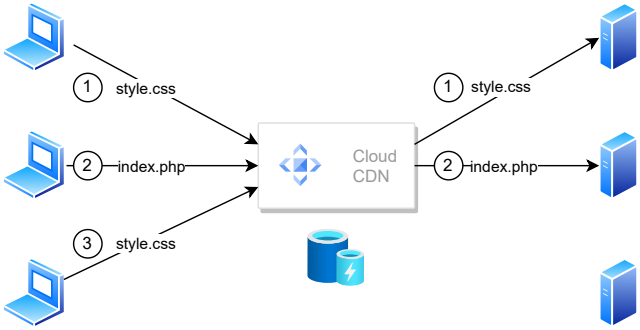

Figure 4: Simplified CDN architecture

stores it in its database, and adds it to the forms presented to the client. Figure 3 shows a login form that includes such a token in a hidden input field. When a user manually fills the form within the application, the token is posted alongside the other field. The server can then determine if the provided token matches the one it generated for that user earlier. Since the token is only accessible from within the application itself (protected through the Same-Origin Policy), an adversary cannot learn the token, thus stopping the CSRF attack. Notably, the token needs to *(i)* be tied to a specific user session and *(ii)* should change on every request to avoid replaying stolen tokens. Moreover, the token must also not be guessable by an attacker, e.g., simply a timestamp combined with the user's name.

The second mitigation technique are so-called *Double Submit Cookies*. Here, a pseudorandom token is generated either on the server or the client side. The token is sent in a request to the server as a cookie and as part of the posted data. The server then determines if the cookie value matches the value in the posted data. While this approach offers the same protection regarding the cookie being inaccessible to the attacker, making it impossible for them to ascertain the token, it does not require the server to keep any state. However, double submit cookies are prone to attacks from the site, yet different origin, as an attacker would then be able to simply set the anti-CSRF cookie to their liking. Thus, site operators should instead rely on the Synchronizer Token Pattern.

Prior work in the area of CSRF has developed approaches to find potential CSRF flaws [26] in open-source projects. Moreover, Likaj et al. [19] investigated the implementation (or lack thereof) of CSRF countermeasures in popular frameworks. Finally, Calzavara et al. [3] developed a machine-learning system dubbed Mitch to find CSRF vulnerabilities in the wild, finding 35 previously undisclosed flaws in 20 major Web sites. Notably, though, none of these works have attempted to understand the implications of improper randomness of such tokens in the wild.

2.3 Content Delivery Networks

A content delivery network (CDN) is a service which provides many geographically distributed servers which can be used to serve assets of a Web site or even entire Web pages to fasten the content delivery and lower the load of the actual origin server. In order to do so, it caches the assets (or the pages) and delivers them to clients such that the original server needs to handle fewer requests. Content will be cached depending on their file extensions, the presence of query strings, cache-control headers sent by the origin server, and the presence of other origin headers that indicate dynamic content (e.g., `Set-cookie`). A simplified architecture is depicted in Figure 4. Here, the first client requests `style.css` from the CDN server. Since this file has not been cached yet, the CDN picks one of the configured origin servers and requests the file. Subsequently, given it is a static resource, the file is cached. The second client instead accesses `index.php`, which is then retrieved from another origin server. This resource is dynamic based on the user's session and should, therefore, not be cached. Finally, a third client requests the previously cached `style.css` file, which is served from the cache without requiring additional round-trips to any of the origin servers.

Prior work has already investigated the security implications of such architectures. Two recent works, Mirheidari et al. [21, 22] highlighted the dangers of Web Cache Deception (WCD). Here, the attacker abuse the fact that CDNs may be configured to cache content based purely on the suffix of the request URL rather than the content or headers from the origin server. This way, an attacker can trick the CDN into caching a file, which is actually specific for a user's session. Still, because the attacker forces the victim to request it with a cacheable suffix (e.g., `index.php/style.css`), it matches the caching rules. Their work primarily highlighted that attackers can use WCD to leak personally identifiable information of a user. In the second work [22], they discussed that WCD can also be used to steal session IDs, CSRF tokens, or CSP nonces. This, however, requires the attacker to force their victim to first visit the crafted URL to populate the cache, steal the token in a separate request from the attacker to the CDN, and then attack their victim. In contrast, our work aims to determine if, even without the presence of an adversary, CDNs (accidentally or intentionally) cache security tokens.

3 THREAT MODEL

At the core of our research lies the question: *how is randomness used on the Web to aid sites in mitigating Cross-Site Scripting and stopping Cross-Site Request Forgery attacks*? To answer this, we first survey the usage of token-based security for highly ranked sites. This allows us to ascertain the pervasiveness of such defense mechanisms in general, as well as to then study the security implications of improper usage of randomness. Second, in case the tokens are not entirely randomly chosen, how do sites implement the security mechanisms instead? Generally speaking, such security tokens can fall into three cases:

(1) *static* tokens are those that are fully static and do not change.
(2) *reoccurring* tokens are re-used for a certain amount of time or from a rotating pool but are not *static*.
(3) *predictable* tokens are based on predictable information such as timestamps, but not *reoccurring*.

3.1 Implications for CSP Nonces

For CSP nonces, the usage of a static value is effectively the same as relying on `'unsafe-inline'`. To attack a server, the adversary builds their payload (assuming they have found an XSS vector, of course), visits the site once to extract the static "nonce" and adds this

to their script. Hence, while such a configuration seemingly avoids the dreaded `'unsafe-inline'` keyword, it offers no protection beyond it.

Similarly, if nonces are reoccurring, be it they are valid for some time or come from a rotating pool, the attacker can simply collect the currently used tokens and opportunistically attack their victim. Concretely speaking, for a set of possible nonce candidates, the attacker simply adds one iframe pointing to the vulnerable page (assuming a reflected XSS) per nonce candidate. This way, even though many of the requests will likely fail to contain the correct nonce, exploitation is still feasible.

Finally, for nonces that are predictable, the attacker does not have to guess anything. Instead, if the algorithm of nonce generation is deterministic and known to the attacker (e.g., through simply using a timestamp), they build the payload with the predetermined nonce and force their victim to visit that page.

Notably, for nonces that are not entirely static, but instead are caused by CDN caches, an attacker may be limited. Specifically, for server-side Cross-Site Scripting flaws, any change in the requested URL would lead to a new request towards the origin server. If this is configured to randomly generate a nonce, the attacker cannot build a payload, which contains the correct nonce, as a new payload inevitably triggers a new origin server response; which in turn carries a new nonce. Nevertheless, in the CDN scenario, client-side XSS threats are still valid, since the payload may originate from insecurely using client-side storage (such as LocalStorage) or parts of the URL fragment (which is not sent in the request).

3.2 Implications for Anti-CSRF tokens

Considering CSRF tokens, having entirely static values may cause two types of issues: first, if the token is actually static, an attacker can simply extract it once and run their attacks. This also applies to tokens used in the double submit cookie technique, since an attack page can issue a hidden request to the victim page before performing the state-changing action to load the known cookie for the victim. Such hidden requests can be issued using hidden iframes, images, or even XHR requests, as long as the response contains the `set-cookie` header used in the double submit cookie technique.

If, instead, the lack of randomness comes from caching, any validation of the CSRF token on the server side should work, as the server would not even have generated a token for the user's session in the first place. Hence, having cached CSRF tokens either leads to functionality breakage or insufficient validation. Note that some stateless token validation mechanisms (incl. standard double submit cookie technique) will also provide insufficient validation since they are detached from user sessions. Here, hidden requests can be used to set the known anti-CSRF value in the cookie for a victim.

If tokens are predictable or caused by origin servers using tokens from a fixed pool, the adversary can again launch an opportunistic attack. Similar to the attack outlined before for CSP, the adversary creates a set of iframes with prefilled forms, one frame per candidate token. When the victim then visits the attacker's page, all the forms are automatically submitted to the target server. Notably, contrary to XSS where the attacker's injected code can directly provide feedback of successful execution (e.g., by sending a PostMessage to the attacker's page), the effect of CSRF might not be immediately visible. Nevertheless, the opportunistic attack is sufficient to undermine the security guarantees intended by Anti-CSRF tokens.

4 METHODOLOGY

Now that we have an understanding of both the considered security mechanisms and the potential threats, we outline our methodology for studying this problem at scale. In this section, we thus first explain details of our data collection procedure, i.e., how we determine a set of candidate sites for further inspection. Subsequently, we outline how we perform the validation to confirm the nature of the token re-usage and, finally, discuss how we detect the usage of a CDN.

4.1 Data Collection

We crawled the highly ranked Top 10,000 sites from the Tranco list [27] generated on February 22, 2023[1]. In addition to loading the site's landing page, we also loaded all subpages that the landing page references up to a limit of 100 subpages. For each page load, we used a clean Chromium instance instrumented by Puppeteer such that browser caches would not interfere with our data collection. To see if a token is re-used, we loaded each of the pages five times to get multiple security tokens from the same request, but also make sure that we do not miss tokens due to randomly missing security headers as shown by prior work [31]. Note that it is not problematic if the first request is already served from a web cache, since we only want to detect reoccurrences.

While extracting CSP nonces from the respective HTTP header or the meta tag is trivial, detecting Anti-CSRF tokens is not. This is because the developer of the application or framework can freely choose the names of the token fields. Therefore, we decided first to investigate how popular Web frameworks such as Django, Drupal, ASP.NET, Spring, etc., name their anti-CSRF tokens. Here we noticed that in each of those cases, the string `token` or `csrf` were present in the name of the token. Hence, we collect input fields from all form tags in the crawled HTML where the name case-insensitively contained either `token` or `csrf`. `token` is, however, frequently used in various scenarios. For example, CAPTCHAs, which are embedded on many sites, often use `captcha_token` as a name. Thus, we restrict all following steps to the subset of token names that case-insensitively contain `csrf` or match the name of tokens employed by a popular framework.

In particular, we match the following token names:

- `authenticity_token` (Ruby on Rails[2])
- `__RequestVerificationToken` (ASP.NET[3])
- `form_token` (Drupal[4])
- `_xfToken` (XenForo[5])

The tokens of other popular frameworks, such as Java Spring (`_csrf`), are already covered, as they contain the substring `csrf`. Other frameworks, such as Laravel or Symfony use more generic

[1] Available at https://tranco-list.eu/list/X57NN
[2] https://rubyonrails.org/
[3] https://dotnet.microsoft.com/en-us/apps/aspnet
[4] https://www.drupal.org/
[5] https://xenforo.com/

Token Name	Framework	#Sites
`authenticity_token`	Ruby on Rails	345
`__RequestVerificationToken`	ASP.NET	190
`_csrf`	Java Spring	130
`csrfmiddlewaretoken`	Django	103
`csrf_token`	-	100
`_csrf_token`	Symfony	33
`form_token`	Drupal	25
`_xfToken`	XenForo	24
`csrf`	-	24
`csrfToken`	cahePHP	23

Table 1: Number of sites that used a certain anti-CSRF token name and which prominent Web framework uses this name as the default.

names such as `_token`. As we want to report a lower bound of the problem and this token name is too ambiguous, we discarded the results for the 163 sites that use this token name. The token names were restricted after manual investigation of the values hinted at not only anti-CSRF tokens being captured. The manual investigation was unable to detect tokens not used for CSRF mitigations in the final selection.

Table 1 shows the top ten anti-CSRF token names used in the top 10k domains that match our pattern. In particular, we can see from the distribution that Ruby on Rails, ASP.NET Core, and Django appear to be popular frameworks. The default behavior of those frameworks is to deploy those tokens random for each request and user. Thus, a reoccurring CSRF token would lead to errors, as the token can not be validated multiple times.

Although our crawl is performed with a real browser, anti-CSRF token values could be injected dynamically upon submission of the form. We would need to submit all forms with an empty value to collect those. To not cause state-changing actions and to not put unnecessary load on the server, we then need to intercept all requests that result from the form submission. Identifying those requests is not trivial as Web sites can use the `event.preventDefault` function in the `onsubmit` handler to abort the form submission in JavaScript and send XMLHTTPRequest (XHR) requests instead. Another problem is that some forms in the wild require CAPTCHAs to be solved before they can be submitted, which we cannot do automatically. Therefore, we only consider name-value pairs that are delivered with the token in the value of the HTML input tag and exclude all occurrences of anti-CSRF tokens where the input tag's value is empty.

4.2 Analysis

As outlined in Section 3, we consider three types of token re-usage: *static*, *reoccurring*, and *predictable* tokens.

According to the CSP standard [41], CSP nonces should follow the base64 alphabet. However, in practice, this requirement is not checked in modern browsers. In general, randomness has to be encoded somehow to deliver it as printable characters. Thus, both, CSP nonces and anti-CSRF tokens are often delivered encoded.

To detect *predictable* tokens we applied common decoding procedures on the Web to see if meaningful output can be recovered. We used the following algorithms for decoding the gathered tokens:

- Standard Base64 decoding
- URLsafe Base64 decoding
- Base32 decoding
- Base16 decoding
- Hexadecimal decoding
- Binary decoding

As soon as one of those decoding functions returned a string that only contained printable characters (punctuations, digits, ASCII letters, and whitespaces), we manually investigated those values to see if they followed a particular pattern (e.g., if a pattern appeared to be a UNIX timestamp).

To detect *reoccurring* tokens, we further investigate all domains that featured which for n different requests to a URL showed n CSPs with a nonce, yet had less than n distinct CSP nonces. In other words, a site has *reoccurring* nonces if we received at least one nonce multiple times.

In the case of CSRF tokens, we apply the same logic. Here, we only look at values for the same parameter name and identify cases where a token appeared more than once. Naturally, we would expect that for each URL on each visit we receive a different token. If we now have fewer distinct tokens per URL across each visit, we received at least one token multiple times for different visits. Thus, at least one token is *reoccurring*.

Last but not least, the final case relates to static tokens. If we repeatedly see a token-based approach being used (i.e., either all responses have a CSP nonce or CSRF token field), we check to see if this token is the same every time. In that case, we consider the token fully *static*.

4.3 Validation

We performed our initial crawl again for all domains where the number of distinct nonces that we found during our initial crawl was less than the total number of nonces for a given domain. Notably, our validation crawl did not gather any URLS but only visited the URL where we encountered the problem. Also, to avoid burdening too much load on the servers, we only pick one single URL per page. To be consistent, we use the shortest URL. This should not be an issue under the assumption that developers apply the same principle across different forms and across different nonce-based CSPs of the application. If the token on the URL we visit shows the same behavior as in the original crawl, we mark them as validated.

4.4 Detecting CDN & Cache usage

To assess if the usage of caching is the reason for the token reoccurrences, we also gathered the response headers of all responses from our validation crawl. CDN providers and other caches usually add additional headers to the responses to indicate cache hits or misses. As the name of those headers is not standardized, we checked for the occurrence of the string `hit` or `miss` in the lowercase value of the response headers. Also, other headers such as *last-modified*, *date*, or *age* can indicate the usage of cached responses.

In the case of CDN caches, the CDN can be configured not to send this information. Similar settings can be done for custom solutions

such as *Squid*, *nginx*, and *Apache HTTP Server*. Thus, in addition to the header data, we also detect the usage of a CDN via DNS. Using a CDN for the caching of HTML requires changes to the DNS setup. Thus, we employ the following DNS-based CDN detection for a domain: First, we query all A records for the domain. Afterward, we use the IP to ASN mapping service by *Team Cymru*[6] to retrieve the autonomous system number (ASN) for each of those IP addresses. The ASN provides us with hints information that hints towards a CDN provider, as we have a mapping between AS name and the relevant providers we consider. We can conclude that a domain uses a CDN with a very high probability if at least one of the A records points to a CDN.

In order to check if cache usage is the root cause of the reoccurrence, we checked if the cache headers are indicating hits on each reoccurrence or if specific headers are changing their value if reoccurring nonces are changing. With the headers, we have a strong indication that caching is the issue since we can find cases where for example, the `last-modified` header indicated the freshness of the response. Thus, if we have multiple reoccurring nonces that are reoccurring with the same `last-modified` header, we have a strong indicator that the nonce is reoccurring due to a cache and thus not a fresh response. Therefore, those numbers will serve as lower bounds here, while the usage of a CDN according to the ASN is an upper bound for the usage of CDNs as a root cause.

4.4.1 Global vs. Local Cache Detection. Roth et al. [31] have shown that different vantage points can lead to different security header configurations because they cache responses from different origin servers or some cache a fresher version of the site than other countries.

In order to observe the difference between localized and global caches, we use multiple vantage points to see how and which kinds of caches interfere with the freshness of security tokens. Here, we classify a site as having a global cache if token overlap exists between some pair of vantage points.

To programmatically issue requests from different vantage points, we use different TOR proxies and perform the requests through those via Python Requests. We configured the proxy using SOCKS5h, such that python requests will also resolve its DNS via the proxy [7]. This way, we will get the IP of the CDN, which is closest to the configured vantage point.

Notably, some sites check for a valid runtime environment to mitigate the impact of DDoS attacks and to prevent simple crawlers from accessing the site. [8] This is, for example, commonly done by checking if cookies are supported by the engine before redirecting to the actual content. The validation was conducted six times from three different vantage points. We use two TOR proxy vantage points with exit nodes in Singapore and the US, in addition to the vantage point used for our initial crawl. Some sites, however, specifically block requests originating from TOR proxies. In our set of domains, we were blocked by nine sites due to the requests originating from a TOR proxy. Interestingly, two out of these sites showed us an error page without sending an appropriate HTTP status code. We noticed this as these error pages did not feature any CSP. If other sites did not send appropriate error codes, their sites at least still featured a CSP with a nonce.

[6] https://www.team-cymru.com/ip-asn-mapping
[7] https://docs.python-requests.org/en/latest/user/advanced/#socks
[8] https://www.cloudflare.com/ddos/

	Total	CDN
Sites	7,210	4,500
Sites with CSP	3,489	2,730
Sites with Script-restricting CSP	1,855	1,479
Sites with Nonces in CSP	446	384
Sites with reoccurring Nonces in CSP	33	28
Sites with Static Nonces in CSP	41	37
Sites with Anti-CSRF Tokens	1058	837
Sites with reoccurring Anti-CSRF Tokens	152	137

Table 2: Overview of Token Reusage and CDN Usage in the Top10k Sites

5 RESULTS

In the following sections, we present the results of our crawl of the top 10k sites. Since many domains only redirected (e.g., `youtu.be`), did not serve any Web content (`windowsupdate.com`), or were unreachable for other reasons, our analysis is restricted to 7,210 different sites in the following sections. We shed light on CDN usage, as well as the deployment of different types of Content-Security Policies and the use of anti-CSRF tokens. Finally, we present the share of sites that featured reoccurring or static security tokens. Section 5 provides an overview of our results and the connection with CDN usage, which are further discussed in the following sections.

5.1 CDN Usage

Table 3 shows the AS name distribution for all domains in our data set. Cloudflare and Amazon Web Services serve as CDN for the majority of sites. Akamai and Fastly follow in third and fourth place. While Google's and Microsoft's CDN services still appear in the top 10, they only have a fraction of the presence the two dominant CDN providers exhibit. This distribution closely matches the distribution of the customer count of CDN providers released by Intricately [13]. It is also important to note that CDN providers heavily use their own services. In Table 3, this inflates the numbers attributed to Amazon, Microsoft, and Google, as they each have multiple sites in the top 10k.

2,491 (about 35%) sites featured more than one AS name. Furthermore, about 62.5% of sites featured at least one of the aforementioned AS names attributed to CDN providers and thus most likely used a CDN. This shows that CDN usage is widespread. However, our numbers will likely be under-approximated since we do not consider additional CDN providers or custom caching solutions.

In the following sections, statistics about CDN usage refer to a site featuring at least one AS name in our data set's top 10 most prominent AS names (see Table 3).

5.2 CSP Nonces

Out of the 7,210 sites we visited in our initial crawl, 3,489 (48%) domains featured a CSP. Prior work has established three use cases: framing control (i.e., the *frame-ancestors*), TLS enforcement (i.e.,

block-all-mixed-content and *upgrade-insecure-requests*) and script content restriction. Here, the usage statistics of our analysis closely match the trends identified by previous works [30]. Only 1,855 (26%) sites deploy a script-restricting CSP. Most importantly, 446 domains (6%) use nonces. Note that nonces are used not only to restrict scripting but also to define valid stylesheets (i.e., *style-src*). In our data set, only one site used a nonce but did not restrict scripting. The number of sites deploying a script-restricting CSP may include sites that only perform TLS enforcement via a source directive (i.e., *script-src https:;*). This is, however, a rare occurrence in the top 10k sites (0.3%).

Figure 5 shows the nonce usage per rank group. In the plot, we count the number of sites featuring any CSP, a script-restricting CSP, and a CSP with nonce per rank group. A rank group combines 500 sites. In total, we thus aggregate these statistics for 20 different rank groups (e.g., 1-500 and 501-1000). Interestingly, CSP usage steadily declines with the rank of domains. Top-ranking domains are more likely to deploy script-restricting policies and are also more likely to employ nonces in those policies. Our observations match the trends identified by previous works [30, 31].

5.2.1 Nonce Re-Usage. In total, we found 446 different sites that used nonces in their CSP. Out of these, 74 (16.5%) sites feature *reoccurring* or even *static* nonces. According to our definitions in Section 4.2, 41 sites feature *static* nonces while 33 sites have

AS Name	Sites
CLOUDFLARENET, US	1,970
AMAZON-02, US	1,477
FASTLY, US	596
AMAZON-AES, US	590
AKAMAI-AS, US	570
AKAMAI-ASN1, NL	425
GOOGLE-CLOUD-PLATFORM, US	245
MICROSOFT-CORP-MSN-AS-BLOCK, US	224
CLOUDFLARESPECTRUM, US	209
GOOGLE, US	187

Table 3: AS Name Distribution

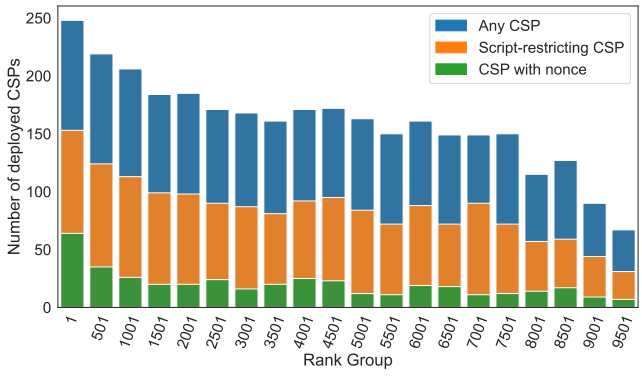

Figure 5: CSP Usage in Top 10k by 500 domain buckets

AS Name	Sites
AMAZON-02, US	15
CLOUDFLARENET, US	11
AKAMAI-AS, NL	7
AKAMAI-ASN1, NL	6
HURRICANE, US	5

Table 4: Top 5 AS names for CSP with reoccurring nonces

reoccurring nonces. For 11 (15%) of those, the HTTP response header indicated that all occurrences of the nonce resulted in cache hits. Thus, those static nonces might only seem to be static but are actually changing every few days depending on the time the cache is valid. Out of the 33 sites with *reoccurring* nonces, only five did not feature A records pointing to popular CDN providers. The top five AS names for the domains are listed in Table 4. Our analysis of the corresponding response headers showed that in at least 16 of those 33 cases, the CSP nonce only changes if and only if cache-related headers such as *last-modified*, *date*, or *age* also change. We thus observe a connection between reoccurring CSP nonces and caching for at least half of the aforementioned sites.

5.2.2 Case Studies: reoccurring CSP Nonces. In the following, we analyze notable exemplary cases representing issues we encountered during our analysis. Since incorrect implementations can vary drastically, we discuss three distinct cases that highlight a range of such incorrect implementations.

(1) **Cache Misconfiguration:** By analyzing the response headers of a Web service that checks data leaks, we were able to see that in this case, caches should be the root cause of the problem. We received in total two different nonces from this site, and the *last-modified* header is set to `Sat, 18 Mar 2023 17:26:53 GMT` for the first nonce and `Sat, 18 Mar 2023 18:06:47 GMT` for the second nonce. This indicates that the TTL of the cache is 30 minutes which allows a malicious actor to re-use the cached nonce for that timespan.

(2) **Cache Misconfiguration on a Shared Software Platform:** While the numbers for Amazon, Cloudflare, and Akamai are expected due to their high market share as seen in Table 3, the presence of `HURRICANE, US` is rather surprising. This autonomous system hosts the infrastructure and a CDN for a popular discussion platform. We found a high number of those forum instances to frequently serve reoccurring nonces. This hints at a misconfiguration of a shared CDN, which is used to serve their pages. This was indeed later confirmed during our disclosure campaign.

(3) **Incorrect Implementation:** A tournament management page of a gaming console vendor rotates or randomly selects a nonce from a bucket of about six nonces. However, the bucket seems to change every few days as observed by closer manual inspection. Thus, attackers can gather those nonces and opportunistically execute their attacks until it succeeds. Per the CSP3 standard [41], CSP nonces are supposed to be base64 values. The following two case studies were found during experiments with different encodings where we decoded

nonces and checked if they entirely consisted of printable characters. Finally, we manually inspected all decodable and printable nonces to check for predictable patterns.

A nonprofit organization that funds cardiovascular medical research uses many different but fixed nonces that are their name plus the use case of the individual script base64 encoded, e.g., `$ABRphonevalidationnonce` or `$ABRrecaptcha-verification`. Given that those nonces are not changing at all, and are all present in the CSP, an attacker can choose one of them to perform attacks.

A Web site of a technology company with focus on cloud and cybersecurity uses a base64 encoded fixed timestamp followed by a newline (`1616760559\n`). This timestamp refers to *Fri Mar 26 2021 12:09:19 GMT*, i.e, a timestamp well before any of our experiments took place. Thus, this might either be the time the nonce was created or it is created on application or server startup. In either case, an attacker can just look up the nonce and use it for attacks. On some pages of their application, they instead use the base64 encoded string `nuxt-site7.9.0`. In a GitHub issue[9] of the `nuxt-js` repository users describe their files being rebuilt upon deployment to serverless Cloud functions. This changes their hash values which, however, have to already be included in the CSP configuration during the deployment. Thus, the application cannot be deployed with a valid CSP configuration. As a workaround, users propose the use of a static nonce which allows deploying a valid CSP, notably citing warnings about the usage of unsafe-inline being discouraged. Obviously, the use of a static nonce is semantically equivalent to relying on unsafe-inline, yet CSP evaluation tools would likely not flag this unsafe behavior.

5.2.3 Global vs. Local Caches. Many CDN providers also offer globally distributed services that may share a cache. This section investigates the prevalence of such globalized caches, as they provide fewer restrictions for potential attackers than localized caches. Since we are using Python Requests instead of a real browser and TOR proxies, some of our connections did not succeed due to bot detection mechanisms or TOR connections being blocked. Thus, we could only validate nonce re-usage on 57 sites. Out of these sites, 42 (74%) sites had nonces that reoccurred on more than one vantage point. This means that about a quarter of sites use localized caches, so no nonces reoccurred on other vantage points. Such localization is often based on IP geolocation [6]. In practice, this is, however, not a hard restriction for an attacker since they can use a VPN or proxy service with endpoints hitting the localized cache to fetch the reoccurring nonce for an attack. For a successful attack, the attacker now has to use the correct nonce that applies to the victim. In many cases, the geolocation of a victim may be estimated (e.g., localized attack distribution) or known before the attack (e.g., targeted attack). Otherwise, an attacker can also use an opportunistic approach with a lower success rate.

[9]https://github.com/nuxt/nuxt.js/issues/8646

AS Name	Sites
AMAZON-02, US	66
CLOUDFLARENET, US	53
AMAZON-AES, US	29
FASTLY, US	26
AKAMAI-AS, US	22
MICROSOFT-CORP-MSN-AS-BLOCK, US	19

Table 5: Top six AS names for reoccurring anti-CSRF tokens

5.3 Anti-CSRF Tokens

In total, we found 1,058 distinct sites that used anti-CSRF token names matching our constraints, from which 152 (14%) showed *reoccurring* token values, and one domain where the value was a *static* token. It is not unexpected that the majority of anti-CSRF implementations distribute random tokens from the origin server as most anti-CSRF tokens appear to be added by popular Web frameworks. In addition, CSRF mitigations are common, as opposed to the use of a CSP, such that most popular frameworks offer mitigations.

Out of the 152 sites with reoccurring values, only 17 sites did not feature A records pointing to popular CDN providers. The top five AS names for the domains are listed in Table 5. We thus observe a connection between reoccurring anti-CSRF tokens and CDN caching. Again, the AS name distribution matches the market share discussed earlier.

Table 6 shows the distribution of token names for sites showing reoccurring anti-CSRF tokens. While it roughly matches the distribution outline in Table 1, the tokens used by ASP.NET Core (`__RequestVerificationToken`) are far less prominent than other token names relative to their initial distribution, as they do not even appear in the top 5. We did not further investigate this phenomenon but have observed differences in cache headers that may indicate that the usage of caching middlewares[10] is responsible for this. ASP.NET Core features two middlewares that dynamically inject cache headers to customize the caching behavior of HTTP proxies and clients.

5.3.1 Case Studies: Anti-CSRF Token Reoccurence. Similar to Section 5.2.2, we analyze notable exemplary cases representing issues we encountered during our analysis in the following section.

[10]https://learn.microsoft.com/en-us/aspnet/core/performance/caching/overview

Token Name	Sites
authenticity_token	84
csrfmiddlewaretoken	16
csrfKey	8
form_token	7
csrf_token	6

Table 6: Number of sites with reoccurring tokens that used a certain anti-CSRF token name

(1) **Cache Misconfiguration on a Shared Software Platform:** While there is no significant outlier in the top AS names (Table 5), the appearance of `MICROSOFT-CORP-MSN-AS-BLOCK, US` in the top 6 is a bit surprising. We noticed the share of `.edu` sites to be significantly higher than the respective for other CDN providers (100%). In particular, all of these `.edu` sites have reoccurring tokens on subdomains such as `events.example.edu` or `calendar.example.edu`. They are using the same calendar system that is using the `authenticity_token`. Their deployment appears to consistently be using the Microsoft Azure CDN while featuring some mechanism to cache the front pages regardless of anti-CSRF tokens.
(2) **Incorrect Implementation:** We found two cases of predictable anti-CSRF tokens. First, an error-monitoring tool uses the current base64-encoded UNIX timestamp as a token at least on their account creation page. Thus, an attacker can easily forge a token before performing the attack. The second case is the Web presence of a US university, which uses the current URL concatenated with the current UNIX timestamp, prepended with a number between 7,200 and 7,800, and base64 encoded the resulting string. For example, their "report a problem" page contains the base64-encoded string equivalent of `7501#web-update/report-a-problem/index.php#1651624778` as a token. Here an attacker could perform the attack multiple times with all possible tokens and one of them will succeed as we only have a small number of possible tokens. Further, a flight-tracking Web application seems to deploy a token that only changes once every day. The token is a nine-digit long number that is not only used in form submissions but also in URLs that are in the href attribute of an HTML anchor element. Thus, an attacker can just get the current token (as it is not bound to a session) and perform a CSRF attack.

5.3.2 Server-Side Behavior. For the server-side behavior in the case of reused Anti-CSRF tokens, we distinguish three possible cases:

(1) **Correct validation:** Server-side caching will interfere with a correct Anti-CSRF implementation as stale, reused or mismatching tokens will be rejected. Such a correct implementation may either be stateful (i.e., keeping server-side state) or stateless (i.e., Double Submit Cookies). As a result, the user experience will degrade as the first submission of a form with a cached Anti-CSRF token will lead to the state-changing action not being performed and the user being displayed an error. This is however only the case when the Synchronizer Token Pattern is used, as cached sites using Double Submit Cookies will pass the stateless check unless the tokens have an expiration date set. Only when the user is served a fresh token from the origin server, the action will be successfully performed in the Synchronizer Token Pattern. The investigation of the effects of such error messages on the user experience and the interaction with Web applications is beyond the scope of this study, but may merit its own study.
(2) **Lax validation:** In this category, we consider validation mechanisms that are not secure by default. This may be the case when tokens are not associated with a user session. An example of such an implementation would be anti-CSRF tokens that may be used only once independent of the user. Such a token leaves a time window for an attack. Similarly, tokens may be valid for a restricted period of time. When not associated with a concrete user session, such tokens again open a time window for an attack. Additionally, this category encapsulates mechanisms that only match a token format.
(3) **Missing validation:** Developers may add Anti-CSRF tokens to all forms especially since this does not come with any overhead when using a framework that provides Anti-CSRF measures. Such measures are however only required if the form is behind authentication. Thus, removing token validation for forms that do not behave differently for authenticated users does not lead to any security drawbacks.

We investigated a random sample of 20 sites serving reoccurring Anti-CSRF tokens for their server-side behavior. Since our crawl is unauthenticated, we mostly found forms for newsletter signup, account creation, search bar, and login. Out of these, only login forms are relevant to our threat model. Thus, our random sample only focuses on sites that case-insensitively feature the keyword `login` or `signin` in the form action. In our random sample, we observed three different behaviors.

Sites falling into the first behavior category displayed an error. This is the expected behavior of sites correctly validating the Synchronizer Token Pattern. In the case of login forms, this behavior did not occur in our random sample. The investigation of security irrelevant forms (e.g., newsletter signup), however, showed that validation is a frequent on these forms. This may result from these forms being of lesser interest than login forms such that cache misconfigurations go unnoticed.

Sites of the second behavior category accepted any value for the anti-CSRF token. This shows that the token was not validated at all. We observed this behavior on seven sites. Some sites did not even send the token along in their form submission, indicating the detection of a false positive by our crawl. This was the case for one site. In a likely scenario, a developer noticed that, for some reason, most form submissions contained incorrect tokens. As a temporary measure, the token validation might have been turned off. This is an interesting scenario, as the misjudgment of a developer may lead to the validation being disabled on a critical form requiring anti-CSRF mitigations. As a prominent example, the calendar system mentioned earlier does not validate the `authenticity_token` on their login form. In addition, they appear not to be using any additional mitigations such as checking the origin header. This allows an attacker to perform a *Login CSRF Attack* [2].

Additionally, we encountered three forms that featured reoccurring anti-CSRF tokens but were not vulnerable due to additional mitigations being applied. In particular, we often encountered the use of CAPTCHAs and were redirected to the next step of a multi-part form where the initial form did not perform a state-changing action. Here, the following parts of the form did not feature reoccurring anti-CSRF tokens. Lastly, one site featured reoccurring tokens used in the double submit cookie technique. This opens an attack window as discussed in Section 3.2 should such a token reoccur. We were not able to check the behavior for eight sites, as they

either changed their caching behavior (6) or no longer employed anti-CSRF mitigations (2).

6 DISCUSSION

In this section, we give recommendations for the different stakeholders on how to mitigate the issue of token re-usage. Moreover, we describe our disclosure process and shed light on the limitations and ethical considerations of our methodology.

6.1 Recommendations

The following section contains our list of recommendations for different parties that may be involved in the deployment process of a Web application.

6.1.1 For Web Operators. This work, as well as previous work on Web Cache Deception [21, 22], highlight the danger of caching dynamic content. Web operators should be aware of the security implications of caching dynamic content. In addition, there should be no disconnect between developers implementing security features and Web operators managing the deployment of the features. Some security features usually implemented on the server side can also be added by Web operators. Cloudflare Workers for example are a serverless execution environment that can be used to implement a secure CSP without generating random values on the origin server[11].

6.1.2 For Web Developers. A first possible approach to mitigating over-eager caching of non-static sites would be to add caching directives via the *Cache-Control* header to responses. In particular, there even exist middlewares that dynamically inject cache headers to customize the caching behavior of HTTP proxies and clients (e.g, ASP.NET Core Caching Middlewares[12]). This effectively shifts some of the responsibility resting on the shoulders of Web operators to the developers. As a CDN may however ignore caching directives from the origin server, usually after explicit configuration, this approach does not completely mitigate the issue.

To mitigate the issues that arise from cached anti-CSRF tokens, one could employ additional mitigation techniques on top of, or instead of the token-based mechanisms. When using *Double Submit Cookies* we recommend the usage of custom headers to transmit the second value, as the protective effects of the Same-origin Policy are now added to the protective effects of randomness.

6.1.3 For CDN Vendors. For both misconfigurations investigated in our work, the CDN provider distribution of misconfigured sites closely matches the actual market share of CDN providers. This indicates that such misconfigurations are widespread and affect all CDN providers equally.

To investigate the possibility of a misconfiguration, we conducted a small case study using Cloudflare as an example. Here, we used a node.js web application that replied with a random token in the HTML of each HTTP response. Additionally, the application also implements a CSP that features a randomly generated nonce in each response. This application was deployed on a virtual private server. Setting up Cloudflare as a CDN only required some modifications of our DNS setup. After configuring the Cloudflare nameservers as the authoritative nameservers for our domain, the CDN was already proxying our traffic under the base configuration. Afterwards, we are greeted with a short quickstart guide featuring a few options to increase security and optimize performance. These options however only address the use of TLS and compression. By default, the Cloudflare CDN respects the cache directives of the origin server added via the *Cache-Control* and the *Expires* header. Additionally, it caches some file types depending on the file extension. Note that JS and CSS files are cached by default, while HTML files are not. Furthermore, responses are not cached if they contain the *Set-Cookie* header. The default behavior can be overwritten using individual page rules. The documentation also states: "To cache additional content [...] create a rule to cache everything." [7] Creating such a rule will now also cache our CSP nonces and the random tokens in the HTML. Furthermore, we can also instruct our rules to ignore cache directives from the origin server. During the configuration of such a rule, the operator is not facing security warnings for either lacking randomness like we discuss or the possibility to introduce Web Cache Deception flaws. This is interesting since other parts of the interface are very beginner friendly and contain hints.

We therefore emphasize CDN providers should place more emphasis on the implications of caching dynamic content. The Cloudflare documentation contains a guide on best practices, which specifically discusses Web Cache Poisoning [8]. It also mentions ways to ensure dynamic content is not inadvertently cached. However, as many may not read these parts of the documentation [38], we believe the warnings should be placed next to options susceptible to misconfiguration.

To further assist Web operators we would recommend a simple check that is performed upon deployment or upon explicit request of the Web operator. While it is difficult for CDN vendors to detect anti-CSRF token caching due to the non-standardized implementation and naming conventions, this is trivial in the case of CSP nonces. These checks would not have to cover the entire cache but rather different subdomains or paths as CSPs are in most cases (90%) deployed per origin [5].

6.1.4 For Browser Vendors. CHIPS (Cookies Having Independent Partitioned State)[13] is a proposal by the Privacy Community Group of the World Wide Web Consortium (W3C) for a new cookie attribute. This attribute, `Partitioned`, indicates that a cross-site cookie should only be available in the same top-level context that the cookie was first set in. Cookies used in the double submit cookie technique could set this new attribute to prevent attackers from performing state-changing requests using cookies set from attacker-issued requests. This feature was shipped with Google Chrome version 110.[14] Other major browsers such as Mozilla's Firefox have also indicated a positive position regarding the standard [15].

Another thing that browser vendors could do to at least notify a developer about the re-usage of security tokens such as nonces is to remember the last `N` nonces that the browser has received from an origin. This way they can easily detect re-usage of a nonce, and as soon as they encounter this they can display a warning message in

[11] https://github.com/moveyourdigital/cloudflare-worker-csp-nonce
[12] https://learn.microsoft.com/en-us/aspnet/core/performance/caching/overview
[13] https://github.com/privacycg/CHIPS
[14] https://chromestatus.com/feature/5179189105786880
[15] https://github.com/mozilla/standards-positions/issues/678

Figure 6: Error page shown by `lego.com` when accessed from the Onion Network (TOR)

the developer console of the browser or even use the report feature of CSP (if specified) to issue a report to the CSP endpoint in order to notify the operator about the problem.

6.2 Ethical Considerations & Disclosure

Our crawling procedure is designed in a way that we only put the minimum necessary load on the server, in order to not interfere with the availability of the Web application.

To not send notifications to no longer affected parties, we performed an additional revalidation step before notifying all affected parties about the corresponding issue(s). In the revalidation step before our disclosure, we could again confirm the reoccurrence of security tokens on 71 Web sites. We sent plaintext emails describing the issue, implications, and methodology. As contacts, we chose the `security@` and `webmaster@` addresses of the respective site since these generic aliases performed adequately in research on large-scale vulnerability notifications [37]. In case both mails could not be delivered, we manually searched for a contact email/form. We received 25 answers, while only nine of those were non-automated responses. Out of these, four confirmed that the reported issues are indeed related to CDN caches, while the remaining did not comment on the underlying issue. In addition, one affected party confirmed that the issue is with their shared hosting provider.

6.3 Limitations

We are limited by a variety of factors during our initial crawl and the following initial analysis to find reoccurring tokens. Since our crawl was automated, there is an unknown amount of sites that we could not visit due to bot protection. Here, we could for example have been presented with CAPTCHAs our automated approach was unable to circumvent. We face a similar issue while validating from vantage points proxied by TOR. Some sites, in our case nine, block requests originating from the TOR network since regular users are indistinguishable from automated traffic routed through the network [28]. For example, `lego.com` return an "Access Denied" error page with the status code set to 403, as depicted in Figure 6.

This is even worsened by the fact that some sites do not send the appropriate status codes when blocking requests and instead just show a block page without an error code. Thus this part of the discussion is not representative for the top 10k sites.

Another limitation of this work is the limited insight into actual server behavior. While we attribute reoccurring security tokens to a cache misconfiguration, the server-side implementation may be the origin of the reoccurrance. Furthermore, there is no clear statistical correlation between CDN usage and token reoccurrance. This is due to the widespread usage of CDNs and also due to the large number of sites where we did not observe a misconfiguration.

In addition, the vast majority of anti-CSRF tokens may well be in parts of Web applications that is not accessible to unauthenticated visitors, e.g., in fields related to account updates. Since there is no reliable way of authenticating automated crawlers, this interesting area remains unexplored.

Lastly, due to the non-standardized naming of anti-CSRF tokens, our estimate of the issue is likely to be an under-approximation since we filtered out ambiguous token names. Future work could address this issue by using other indicators that hint at a certain Web framework being used. Not only CDNs cache responses, but also some Web frameworks and Web servers or proxies do so. Thus, if a site for example used a custom caching via e.g., their nginx[16], we would not have detected this case as caching related although it actually is caused by a server-side cache.

Roth et al. [30] have shown that CSP configurations change frequently. Thus, some sites changed their CSP during our initial crawl over between our crawling and validation phase. Thus, we might have missed cases, because the usage of nonces in the CSP changed throughout our experiment. As mentioned in Section 4.1 we dropped all anti-CSRF tokens that have empty value attributes, because collecting them is error-prone. In either of the cases mentioned above, we might have missed multiple static tokens which are dynamically added. However, as we aim to report a lower bound of the problem, this does not interfere with the validity of our results.

7 CONCLUSION

In this work, we investigated the re-usage of security tokens in real-world Web sites. Through our analysis of the 10k most popular Web sites, we found that out of the 7,210 sites we reached, 446 sites use CSPs that include nonces. However, 74 (16.5%) of them use nonces that are reoccurring or even static. Also, 1,058 sites use Anti-CSRF tokens in their Web applications. However, also here, we detected reoccurring tokens on 152 (14%) sites. Moreover, we detected a connection between reoccurring security tokens and the usage of CDNs for caching. In fact, the majority (88% for nonces, 89% for anti-CSRF) of reoccurring or static security tokens that we found were present on Web sites that were likely served via a CDN. This is also supported by an analysis of caching-related headers, which indicated a cache-related issue in at least half of the investigated cases. In addition, at least four operators of the affected Web sites confirmed during our disclosure process that caching is the root cause of this problem. To mitigate this issue, we outline several mitigation techniques in order to fix this issue from the perspective of the different stakeholders. Web site operators need to be aware of the issues that caches can introduce and solutions like Cloudflare Workers should be advertised more prominently for this use case. Developers should carefully design their Cache-Control headers to control the caching behavior of their CDN. But also CDN vendors should work on easing the configuration of the caching behavior to reduce misconfigurations, and browser vendors should try to notify affected sites about the issue, such that the security header can protect what they ought to protect.

[16] https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/

ACKNOWLEDGMENTS

We would like to thank all reviewers for their comments and advices on how we can further improve the presentation and readability of our paper. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 491039149. It was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

REFERENCES

[1] Adam Barth. 2011. RFC 6265: HTTP State Management Mechanism - Overview. https://www.rfc-editor.org/rfc/rfc6265#section-3
[2] Adam Barth, Collin Jackson, and John C Mitchell. 2008. Robust defenses for cross-site request forgery. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[3] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. 2019. Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.
[4] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content Security Problems?: Evaluating the effectiveness of Content Security Policy in the Wild. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[5] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. 2021. Reining in the web's inconsistencies with site policy. In *Network and Distributed Systems Security Symposium (NDSS)*.
[6] Cloudflare. 2022. Cloudflare about the Cloudflare Network. *Online @ cloudflare.com* (2022).
[7] Cloudflare. 2022. Cloudflare Cache Documentation. *Online @ developers.cloudflare.com* (2022).
[8] Cloudflare. 2022. Cloudflare Documentation on Avoiding Web Cache Poisoning. *Online @ developers.cloudflare.com* (2022).
[9] Cloudflare, Inc. 2022. What is a CDN? | How do CDNs work? *Article. Online @ cloudflare.com* (2022).
[10] Adam Doupé, Weidong Cui, Mariusz H Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. 2013. deDacota: Toward Preventing server-side XSS via automatic Code and Data Separation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[11] Ben Hayak. 2015. Same Origin Method Execution (SOME). *Online @ benhayak.com* (2015).
[12] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. 2013. mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations. In *ACM Conference on Computer and Communications Security (CCS)*.
[13] Intricately. 2020. CDN Industry: Trends, Size, And Market Share. *Blog Post. Online @ blog.intricately.com* (2020).
[29]]jakobssonjavascript Markus Jakobsson, Zulfikar Ramzan, and Sid Stamm. [n. d.]. JavaScript Breaks Free. *Online @ citeseerx.ist.psu.edu* ([n. d.]).
[15] Ashar Javed and Jörg Schwenk. 2013. Towards Elimination of Cross-Site Scripting on Mobile Versions of Web Applications. In *International Workshop on Information Security Applications (WISA)*.
[16] Amit Klein. 2005. DOM Based Cross Site Scripting or XSS of the Third Kind. *Online @ webappsec.org* (2005).
[17] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A Vela Nava, and Martin Johns. 2017. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[18] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[19] Xhelal Likaj, Soheil Khodayari, and Giancarlo Pellegrino. 2021. Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[20] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *Network and Distributed System Security Symposium (NDSS)*.
[21] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. 2020. Cached and Confused: Web Cache Deception in the Wild. In *USENIX Security Symposium (USENIX Security)*.
[22] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. 2022. Web Cache Deception Escalates. In *USENIX Security Symposium (USENIX Security)*.
[23] Open Web Application Security Project (OWASP). 2021. Top 10 Web Application Security Risks. *Online @ owasp.org* (2021).
[24] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. CSPAutoGen: Black-box Enforcement of Content Security Policy upon real-world Websites. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[25] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. DexterJS: robust testing platform for DOM-based XSS vulnerabilities. In *Joint Meeting on Foundations of Software Engineering (FSE)*.
[26] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[27] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Network and Distributed Systems Security Symposium (NDSS)*.
[28] The Tor Project. 2022. FAQ on Websites Blocking Tor. *Online @ support.torproject.org* (2022).
[29]]content-sniff-xss Phil Ringnalda. [n. d.]. Getting around IE's MIME type mangling. http://weblog.philringnalda.com/2004/04/06/getting-around-ies-mime-type-mangling
[30] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In *Network and Distributed Systems Security Symposium (NDSS)*.
[31] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. 2022. The Security Lottery: Measuring Client-Side Web Security Inconsistencies. In *USENIX Security Symposium (USENIX Security)*.
[32] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 2021. 12 Angry Developers – A Qualitative Study on Developers' Struggles with CSP. In *Conference on Computer and Communications Security (CCS)*.
[33] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Network and Distributed Systems Symposium (NDSS)*.
[34] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the Web with Content Security Policy. In *International Conference on World Wide Web (WWW)*.
[35] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. 2021. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In *Network and Distributed Systems Security Symposium (NDSS)*.
[36] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.. In *Network and Distributed Systems Symposium (NDSS)*.
[37] Ben Stock, Giancarlo Pellegrino, Frank Li, Christian Rossow, and Michael Backes. 2018. Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications. In *NDSS*.
[38] Brigit van Loggem. 2014. 'Nobody reads the documentation': true or not?. In *Proceedings of ISIC: the information behaviour conference*.
[39] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. 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)*.
[40] Michael Weissbacher, Tobias Lauinger, and William Robertson. 2014. Why is CSP failing? Trends and challenges in CSP adoption. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*.
[41] Mike West. 2021. CSP Level 3. *W3C Standard. Online at w3.org* (2021).