Web Cache Deception Escalates!

Seyed Ali Mirheidari
University of Trento & Splunk Inc.

Matteo Golinelli
University of Trento

Kaan Onarlioglu
Akamai Technologies

Engin Kirda
Northeastern University

Bruno Crispo
University of Trento

Abstract

Web Cache Deception (WCD) tricks a web cache into erroneously storing sensitive content, thereby making it widely accessible on the Internet. In a USENIX Security 2020 paper titled "Cached and Confused: Web Cache Deception in the Wild ", researchers presented the first systematic exploration of the attack over 340 websites. This state-of-the-art approach for WCD detection injects markers into websites and checks for leaks into caches. However, this scheme has two fundamental limitations: 1) It cannot probe websites that do not present avenues for marker injection or reflection. 2) Marker setup is a burdensome process, making large-scale measurements infeasible. More generally, all previous literature on WCD focuses solely on personal information leaks on websites protected behind authentication gates, leaving important gaps in our understanding of the full ramifications of WCD.

We expand our knowledge of WCD attacks, their spread, and implications. We propose a novel WCD detection methodology that forgoes testing prerequisites, and utilizes page identicality checks and cache header heuristics to test any website. We conduct a comparative experiment on 404 websites, and show that our scheme identifies over 100 vulnerabilities while "Cached and Confused" is capped at 18. Equipped with a technique unhindered by the limitations of the previous work, we conduct the largest WCD experiment to date on the Alexa Top 10K, and detect 1188 vulnerable websites. We present case studies showing that WCD has consequences well beyond personal information leaks, and that attacks targeting non-authenticated pages are highly damaging.

1 Introduction

A *web cache* refers to any technology that fronts a busy web infrastructure with the goal of temporarily storing and quickly serving frequently accessed objects. That translates to reduced load for servers, and better performance for clients.

The security community is no stranger to attacks targeting web caches. These often fall under one of two categories; poisoning caches with an exploit payload to be delivered to unsuspecting clients, or tricking the cache into storing confidential information which is then publicly exposed on the Internet. Attacks date back to the early 2000s, and the fundamental techniques have not significantly changed over the years – but the attack surface and damage potential *have*.

Content Delivery Networks (CDNs), which are globally distributed Internet overlay networks made up of caching reverse proxies, have become a ubiquitous component of many online systems that have stringent scalability, availability, and performance requirements. Official deployment figures published by three major CDN vendors Akamai, Cloudflare, and Fastly give us a glimpse of the vast amount of traffic proxied via these web caches [2, 9, 17]. A recent measurement by Guo et al. shows that 74% of the Alexa Top 1K websites utilize a CDN for delivery [22]. As of June 2021, BuiltWith estimates that of the top 10K, 100K, and 1M websites they observe, 71.79%, 62.70%, 46.59% are behind a CDN, respectively, with upward trends [5]. Combined with many other, stand-alone caching proxies (e.g., Squid, Varnish [42, 48]) and caching servers (e.g., Apache, NGINX [4, 37]) sprinkled along the Internet, it is evident that web caches are rapidly becoming critical infrastructure. That, in turn, considerably increases the likelihood and impact of a web cache attack.

As this evolution of caching technologies keeps raising the stakes, a surge of interest in novel exploitation techniques follow (e.g., [20, 29–31, 36, 38]). Notably, Omer Gil helped put the spotlight on this threat in 2017 with his work on *Web Cache Deception (WCD)*, an attack that tricks a publicly accessible proxy into caching and leaking sensitive content normally intended to be uncacheable [20, 21].

While Gil described proof-of-concept attacks on specific high-profile targets, Mirheidari et al. published *"Cached and Confused"* (or CC for short), the first work that explored the causes and consequences of WCD within a scientific framework in 2020 [36]. In particular, the authors proposed a detection methodology that involves manually creating accounts on websites to inject unique *markers* into user-editable fields, and then testing the websites with WCD exploits, checking

for the presence of markers in server responses. If the marker is present, that would indicate erroneous caching of a page containing user information, or in other words, a successful attack. The authors employed this methodology to conduct a large-scale measurement on 340 websites, found 37 to be impacted, and concluded that WCD is a widespread threat.

While the literature described above is functional and valuable as a starting point, we nevertheless observe two fundamental issues with the previous work, which limit the security community's understanding of WCD.

First, previous work solely investigates attacks on *user-provided* personal information protected behind authentication gates, and therefore, the aforementioned marker injection methodology is specifically crafted to detect erroneous caching of pages that contain such information. This approach falls short of testing pages that do not reflect user input, where there are no avenues for marker injection. Furthermore, there is a plethora of security-critical secrets (e.g., CSRF tokens, CSP nonces, OAuth state parameters) on publicly accessible pages that do not require authentication, or on websites that do not support creating user accounts at all. In such cases, marker injection is not possible or meaningful. Existing approaches have no way to test those websites, and consequently no visibility into the WCD vulnerabilities they may contain.

Second, a marker-based approach necessitates a costly process for creating and populating user accounts on every tested website, posing a roadblock to scaling up the experiments. As Mirheidari et al. also explained in their paper, this overhead limited their experiments to 295 websites using Google OAuth and 45 others where accounts had to be manually created, and therefore biased their results. In all cases, user inputs were identified and markers injected manually.

In this paper, we set out to propose a WCD detection methodology that is not hindered by the attack surface coverage and scalability limitations of the previous work. We subsequently aim to gain new insights into the severity and spread of WCD attacks.

We first present a novel methodology for detecting WCD vulnerabilities (Web Cache **D**eception **E**scalates, or DE for short). Our approach uses content identicality checks and HTTP response header heuristics in lieu of markers, and can identify vulnerabilities on *any* website. Eliminating markers also means that there is no manual setup phase involved.

We conduct an initial study on a dataset of 404 websites, and make a three-way comparison between CC and two variations of DE. Our results show that CC finds only 18 vulnerable websites, whereas our approach significantly outperforms the state-of-the-art by detecting over **100**.

Equipped with an effective methodology that is not bound by coverage or scalability limitations, we next perform the largest-scale WCD experiment to date on the Alexa Top **10K**. We detect **1188** vulnerable websites. We analyze and discuss the vulnerabilities in detail, presenting concrete evidence that WCD attacks that do not target personal information and do not exploit pages behind authentication gates are still highly damaging. Our findings reaffirm that WCD is a serious threat, but also show WCD impacts the Internet at a much greater scale than previously estimated.

To summarize, we make the following contributions:

- We present a novel methodology DE to detect WCD vulnerabilities. DE addresses the coverage and scalability limitations of the state-of-the-art approach for detecting WCD in the wild.
- We conduct a comparative experiment on 404 websites, evaluating the pros and cons of different WCD detection methodologies. We show that our approach DE significantly outperforms CC.
- We perform the largest-scale measurement experiment to date for detecting WCD in the wild, testing 10K websites. We identify 1188 vulnerable websites.
- We discuss case studies on real-life vulnerabilities impacting high-profile websites, presenting evidence for the first time that WCD attacks pose a serious threat beyond leaking personal information.

Availability. Our source code is publicly available on the authors' websites.

Disclosure. The authors of this work and "Cached and Confused" overlap. This is the follow-up to our previous WCD research.

2 Background & Research Goals

We first present an overview of web caches and how they can be exploited via WCD attacks. As our work extends the prior art on cache attacks, we also present an early discussion of related work and differentiate our research goals.

2.1 Web Caches

Even with troves of personal and sensitive data traversing the Internet, a disproportionately large slice of traffic is made up of content available for general consumption. These include static web pages, style sheets, JavaScript, documents, multimedia, software downloads, and streaming applications, which cover the whole gamut of possible sizes and access patterns. Repeated transfers of such objects can quickly get costly for both servers and clients, and even impact the overarching Internet infrastructure involved in traffic delivery. Web caches are designed to address this problem.

A web cache conceptually sits between a user issuing a web request and the destination the requested object originates from – hence often called the *origin* server. Web caches act as man-in-the-middle proxy devices, intercept the traffic, and temporarily store objects so that subsequent requests for the

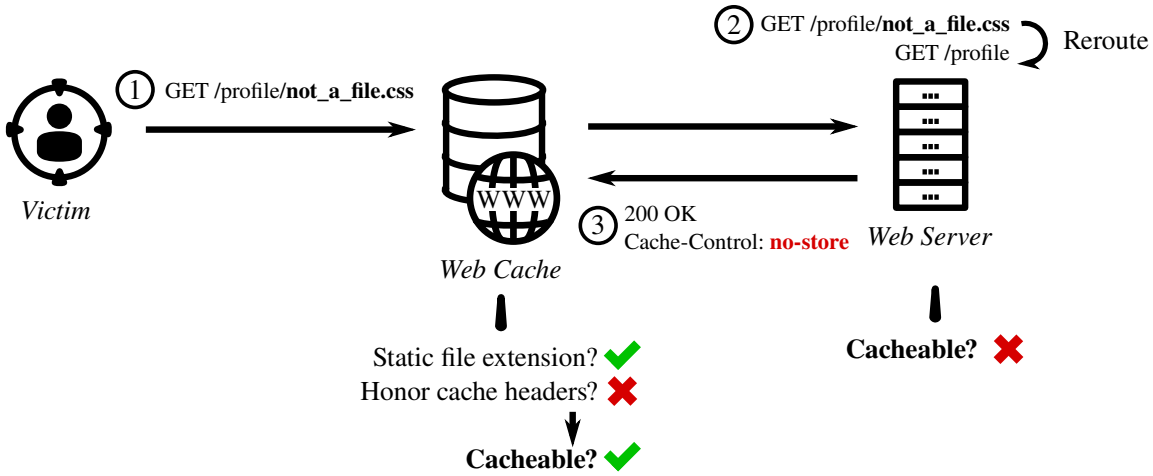

Figure 1: WCD in action. A social engineering victim clicks on a malicious URL, which in turn tricks a web cache into storing sensitive profile information, publicly exposing it on the Internet.

same can be quickly served from the cache. This reduces the round-trip time for the requester, load for the server, and the overall traffic volume for the Internet infrastructure.

Web caches are implemented at multiple stages on the traffic delivery path, starting from the private caches inside browsers, ending at the application caches deployed together with the origin server, and any caching proxies that may lie in-between. Foremost, Content Delivery Networks (CDNs) with their global networks of caching proxies (i.e., *edge servers*) have become pervasive [5, 22].

Web caches are designed for storing static objects that do not have confidentiality requirements, whereas dynamically generated content that includes personal or sensitive information for each different client must be fetched from the origin afresh with each request. It is important to point out that one should not conflate *static* content with *public* content. For instance, public web pages may still contain unique, sensitive parameters dynamically generated for each visitor.

CDNs offer numerous options for website administrators to configure the caching behavior according to their needs. For example, caching decisions can be made based on the request endpoint, file extension, query string parameters, presence of a cookie, request headers, response content type, or a complex combination of many similar parameters [8, 12, 13]. More recently, major CDNs have also started to offer *edge computation* capabilities, enabling website operators to make these decisions programmatically [1, 11, 15].

Finally, the HTTP/1.1 specification defines the *Cache-Control* response headers, allowing an origin to indicate to all the downstream caches how a response body should be handled [18]. However, note that all major CDN providers allow for disregarding these cache control headers, and as Mirheidari et al. showed previously, some indeed have default configurations that do [36].

2.2 Web Cache Deception

Web Cache Deception (WCD) is an attack that exploits the request processing discrepancies between a web cache and an origin server, and subsequently tricks the cache into erroneously storing sensitive content. WCD was introduced by Omer Gil in 2017 [20, 21]. Below, we demonstrate the attack through a hypothetical case inspired by Gil's original proof-of-concept.

Figure 1 represents a typical deployment model where the origin application server is fronted by a cache. The cache server is configured to store frequently accessed static objects as determined by checking their file extensions. The attack begins when a miscreant crafts a malicious link containing the URL to a page with sensitive user profile details, but also appends to it an invalid path component that *appears* to be a static file. In this case, *"example.com/profile/"* is the legitimate page being targeted, and ***"not_a_file.css"*** is a reference to a non-existent style sheet. The attacker then distributes this link (i.e., the attack URL containing a WCD payload) via social engineering channels, and the attack plays out as follows.

1. The victim clicks on the link and their browser issues the HTTP request for the resource. The web cache receives and promptly forwards the request to the origin server.

2. The origin receives the request for the made-up resource and sees that the referenced style sheet does not exist. Therefore, it strips away the invalid path component, and reroutes the request to the *"/profile"* endpoint instead. The server indicates that the profile details should not be cached by setting the appropriate cache control headers in the response.

3. The web cache receives back the response and consults its caching rules. Oblivious to the request rewriting tak-

ing place at the origin, the cache finds a match indicating that *.css* extensions are cacheable. While there may be cache control headers present in the response, the cache is not configured to honor upstream headers. The web cache concludes that the response is safe to store. At this point, the sensitive content is publicly accessible under the URL *"example.com/profile/not_a_file.css"*.

This attack is possible due to the complex interactions between web caches, origins, and their administrators, which collectively lead to myriad potential HTTP processing discrepancies. For example, the request rerouting in Step 2 is a common behavior implemented by web frameworks that follow *clean URL* principles, as opposed to treating URLs as filesystem paths [50]. However, this backend logic is invisible from the caching proxy's vantage point. Similarly, ignoring upstream cache control headers is common practice and sometimes the default web cache configuration [36], for instance, in a large enterprise environment, where centralized management of caching rules is preferable to individually configuring web servers to return the correct headers. All in all, detecting and mitigating WCD is a non-trivial task, and neither application owners nor cache vendors are to individually blame; this is a complex system interaction problem.

2.3 Cached and Confused

In their USENIX Security 2020 paper titled *"Cached and Confused: Web Cache Deception in the Wild"*, Mirheidari et al. presented the first study exploring WCD within a scientific framework [36]. In particular, they proposed a methodology for detecting WCD in the wild and conducted a large-scale study on 340 websites drawn from the Alexa Top 1K, finding 37 of them vulnerable. The authors also proposed novel WCD payloads, or *path confusion* techniques, and surveyed the top CDN vendors with their default caching configurations, highlighting the factors contributing to the issue. This WCD detection methodology is highly relevant to our work, and we use the abbreviation CC to refer to it in the text.

At a high level, CC works as follows.

1. The tester creates an account on the website and populates user-editable fields that would normally hold personal or sensitive information with unique markers.

2. A crawler with valid authentication cookies tests the pages of the website with WCD exploits. This crawler simulates a logged in victim clicking on URLs containing WCD payloads.

3. A second crawler, this time without authenticating to the site, requests the same pages targeted in the previous step. This crawler simulates an attacker probing for successful exploits. If the response contains a marker, one of the exploits in the previous step was successful in tricking a cache into storing the page, exposing the information to an unauthenticated request.

One advantage of this approach is its robustness against false positives; the presence of a marker is strong evidence that an information leak is taking place. In fact, Mirheidari et al. cite this property as one of the reasons they chose not to employ fuzzier detection techniques. On the downside, marker injection is a manual process. The authors also acknowledge this limitation, which forces them to cap their experiments at 340 websites, 295 of which are chosen specifically due to their support for Google OAuth, easing the account creation burden through automation support.

A more fundamental limitation of CC is that it is calibrated for WCD scenarios that involve leakage of personal information protected behind authentication gates. That comes at a cost: CC has no visibility into the caching behavior of a website when the page under test does not reflect user input (i.e., markers). In fact, some websites may not even have viable avenues for marker injection. Hence, CC forfeits the opportunity to detect vulnerabilities on such pages in order to achieve robust results on pages that do reflect user input. This is significant, because erroneous caching has implications beyond personal information leaks. Dynamic pages, be they publicly accessible or protected behind authentication gates, may include secrets such as CSRF tokens, CSP nonces, and OAuth state parameters, with dire consequences if stolen. Mirheidari et al. do allude to this possibility, but they are not equipped to explore that direction using CC.

2.4 Our Motivation & Goals

Our research is directly motivated by the limitations of prior work on WCD, and important gaps those may have left in the security community's understanding of WCD's spread and impact. We propose a new methodology DE, which challenges the core design decisions made for the state-of-the-art approach CC, and in doing so allows us to explore WCD in the wild at a depth and scale previously not possible. In doing so, we aim to equip website owners and researchers with better awareness, techniques, and tools to mitigate vulnerabilities, but also to estimate how easily miscreants can identify the same vulnerabilities.

In particular, we tackle the following limitations of CC.

(P1) **Coverage Problem.** CC cannot test web pages that do not reflect markers.

(P2) **Scalability Problem.** CC has the costly prerequisites of account creation, user input identification, and marker injection – all performed manually.

By addressing these limitations, our goal is to answer the below research questions.

(Q1) How does our fuzzier WCD detection methodology DE perform compared to marker injection?

(Q2) How does expanding the scope of an Internet-wide measurement to 10K websites change our established understanding of WCD?

(Q3) What is the impact of WCD on security beyond personal information leaks? Is erroneous caching of other types of sensitive data, and in particular, those found on public pages not protected behind authentication gates, practicable? If so, what are the consequences?

2.5 Other Related Work

The works we extensively discussed above remain the only literature directly investigating WCD. Below we briefly list other attacks on web caches and CDNs.

Web cache *poisoning* is a class of attacks that involves tricking a web cache into storing a malicious payload. This essentially escalates any reflected web application attack into a stored one, widely distributed to every client accessing the cache. For example, James Kettle presented a set of such attacks on popular caching proxies [29], and more recently introduced more advanced attacks exploiting the cache key construction mechanisms used by these technologies [31]. In academic literature, Chen et al. exploited the inconsistent processing of the host header values in requests to the same effect [6]. Nguyen et al. proposed a different take on cache poisoning, employing erroneous negative caching (i.e., caching of error responses) as a means to block access to websites, resulting in a denial-of-service attack [38].

A closely related attack is *HTTP request smuggling (HRS)*. HRS targets the discrepancies in how proxies and origins determine HTTP message boundaries, which can be exploited to poison caches among other nefarious tasks. The first documented instance of practical HRS dates back to a white paper by Linhart et al. published in 2005 [35]. HRS has seen a resurgence in popularity like cache attacks, and researchers proposed new variations (e.g., [30, 32, 33]). Jabiyev et al. presented the first systematic exploration of HRS across popular server and CDN technologies via differential fuzzing [27].

The security community has made available numerous open-source projects to simplify the detection of cache attacks (e.g., [14, 26, 39, 41]). These tools primarily aim to assist penetration testers with their manual processes, targeting a specific, controlled environment. On the defense front, Amazon Web Services released a tool that inspects and categorizes requests according to their RFC compliance [3]; however, the effectiveness of this tool is yet to be quantified. All in all, there is no generally applicable detection or defense tool for cache attacks at this time.

Besides the caching issues under focus here, researchers have long studied CDNs in other security contexts, including insufficient origin validation [22], insecure mapping of clients to edge servers [24], request forwarding problems that may facilitate denial-of-service attacks [7, 23, 47], and use as a censorship evasion vector [19, 25, 51]. Other works investigated methods to reveal the origin addresses fronted by edge servers, effectively bypassing the protections afforded by a CDN [28, 49]. These works are orthogonal to our research.

3 Methodology

Our new methodology DE uses a combination of content identicality checks and header inspection heuristics to overcome the limitations of CC. While the high-level approach is the same (i.e., launch a WCD attack, verify its success), DE may not be as intuitive as injecting and retrieving markers at a first glance. Therefore we adopt a top-down presentation; we describe the high-level scheme first, and later dive into details.

Algorithm 1: DE testing an input URL for WCD.

```
   input : URL
1  result1 ← get(URL);
2  result2 ← get(URL);
3  if result1 ≠ result2 then
4      attackURL1 = generateAttackURL(URL);
5      attackURL2 = generateAttackURL(URL);
6      result1 ← get(attackURL1);
7      result2 ← get(attackURL2);
8      if result1 ≠ result2 and result1.cache = MISS then
9          result2 ← get(attackURL1);
10         if result1 = result2 and result2.cache = HIT then
11             return WCD detected;
12         end
13     end
```

3.1 Overview

Algorithm 1 presents the complete pseudo-code for our approach. Given a URL to test for the presence of a WCD vulnerability, we perform checks in three steps. If all three checks pass, we conclude that the URL contains an exploitable WCD vulnerability. We explain these steps below.

Step 1 – Does the URL return dynamic content? The premise of WCD is tricking a cache into storing dynamically generated content, as static pages are unlikely to contain sensitive data. Therefore, as a first step, we request the input URL two times, each with a fresh client state, and compare the responses (lines 1-3). If the results are identical, we conclude that this is a static page, and we abort the test. Otherwise, the URL contains dynamic content, and we proceed.

Step 2 – When we launch a WCD attack, does the server still respond with dynamic content? The next step is launching a WCD attack by modifying the input URL with a WCD payload to craft an attack URL, and requesting it. The modification process is similar to the example we presented in

Figure 1; we append a path component to the URL, which points to a non-existent style sheet. We randomize the file name to prevent Internet users from inadvertently accessing the same URL and getting poisoned cache contents. We use the `.css` extension in our payloads following the guidance from prior WCD literature; while the attack could work with other static file extensions, style sheets exist on virtually all websites, making them the optimal candidate for WCD tests.

We then make our WCD attempt by requesting this attack URL, simulating a victim visiting the link. One consideration here is to ensure that the server still responds with dynamic content to the request. That may not always be the case, for example, if the attack fails and the server responds with a generic error page. To tackle this problem, we generate *two* unique attack URLs with randomized payloads as described above (lines 4-5), launch two attacks by requesting both (lines 6-7), and compare the results (line 8, the first condition). If the results are identical, the attack has failed, and we abort the test. Otherwise, if the results differ, we proceed to the final step where we verify whether the attack was successful.

The avid reader may wonder why the dynamic content check in Step 1 is necessary if we perform a similar check again in Step 2. In a real-life test scenario, a website would be probed with multiple path confusion techniques, each resulting in a different attack URL and exposing new WCD vulnerabilities – we use the 5 techniques presented in previous work, and propose 7 new ones later in our experiments. In other words, Step 2 would be repeated many times over, slowing down the tests and putting a heavy traffic load on websites. The check in Step 1 gives us an early opportunity to filter out static pages that are not of interest, using only one request pair – a significant optimization. We need to perform a second check in Step 2 for each WCD payload to ensure that the server still responds to the modified URL.

Step 3 – Is the origin response to the attack URL cacheable? Recall that for WCD to succeed, the origin server must serve a dynamic response that erroneously gets cached. Further breaking that down, on a vulnerable site, the attack URL we requested in Step 2 (i.e., simulating a victim interaction) must elicit a response from the origin server, but further requests for the same attack URL must be served from the cache (i.e., simulating how an attacker would retrieve the sensitive content).

In this final step, we precisely perform this check by inspecting the HTTP response returned when we first visited the attack URL (line 6), and the response for a repeat request for the same URL (line 9)[1]. Specifically, we perform two sets of checks. First, we utilize HTTP response header heuristics to verify that the initial request was a cache miss (i.e., it was served by the origin), but the latter request was a cache hit (lines 8 and 10, both second conditions). Next, we compare the response bodies to verify that they are indeed identical (line 10, the first condition), which provides added assurance for the correctness of our header heuristics. If both checks pass, we conclude that the attack was successful, and that the URL has an exploitable WCD vulnerability.

[1] We could have used either of the two attack URLs we generated in Step 2 to verify the attack's success. We chose to use the first one.

3.2 Cache Header Heuristics

DE inspects HTTP response headers to heuristically determine whether a request is served from the origin server or a web cache in Step 3 above.

Web caches often transform responses by including a header that indicates to the client the result of the cache lookup. However, this mechanism is not standardized, and cache technologies implement their own proprietary headers (e.g., [10, 16, 40]). Therefore, we performed an exploratory crawl of the Internet prior to this work, supplemented that with vendor documentation, and compiled a list of header fields and values returned by popular web caches. We present these results in Table 1.

Note that the headers and their values show strong similarities between different caches. Namely, all headers we identified contain the term `cache`, and most values either `hit` or `miss`. Therefore, instead of doing strict equality checks, DE normalizes the received headers and then performs keyword searches in them. In our exploratory study, we determined this method to work as well as enforcing strict checks, with two added advantages. First, this approach makes our detection more robust against minor format or structure differences in headers often observed in the wild, for example, due to man-in-the-middle devices that incorrectly transform requests, or version differences between caches. Second, it opens up the possibility for DE to work correctly with sparsely used or private cache technologies that may be observed in large-scale experiments, provided that they follow the same conventions with their headers.

3.3 Interpreting the Results

DE addresses both limitations of CC. We do not rely on the presence of a marker or any other particular reflected input on the page, and therefore DE can test any website for WCD (i.e., we resolve the coverage problem (P1)). Similarly, because there is no initial setup necessary, DE can run large-scale experiments on the Internet or complex private enterprise deployments (i.e., we resolve the scalability problem (P2)).

We achieve these properties by utilizing fuzzier detection techniques and heuristics. Heuristics can and do fail, presenting interesting trade-offs between DE and CC. Before we experimentally investigate these, we explain what our scheme is designed to detect, and the ways it can fail.

True Positives. DE is designed to detect dynamic content that is *not cacheable* when requested through its normal URL, but is *erroneously cached* when requested with a maliciously crafted URL – the very definition of WCD. This definition

Table 1: Cache lookup status headers used by popular web caches.

CDN / Cache	Header Name(s)	*Hit* value(s)	*Miss* value(s)
Akamai	`server-timing, X-Cache, X-Cache-Remote`	`desc=HIT, TCP_HIT`	`desc=MISS, TCP_MISS`
CDN77	`X-Cache`	`HIT`	`MISS`
Cloudflare	`cf-cache-status`	`HIT`	`MISS`
CloudFront	`x-cache`	`Hit from cloudfront`	`Miss from cloudfront`
Fastly	`X-Cache`	`HIT`	`MISS`
Google Cloud	`cdn_cache_status`	`hit`	`miss`
KeyCDN	`X-Cache`	`HIT`	`MISS`
Azure	`X-cache`	`TCP_HIT, TCP_REMOTE_HIT`	`TCP_MISS`
Apache, ATS	`X-Cache`	`HIT`	`MISS`
NGINX	`X-Proxy-Cache`	`HIT`	`MISS`
Rack Cache	`X-Rack-Cache`	`hit`	`miss`
Squid	`X-Cache`	`HIT from *`	`MISS from *`
Varnish	`X-Cache`	`HIT`	`MISS`
Unknown	`x-cache-info`	`cached`	`caching`

does not make any assumptions about the *impact* of the attack; the erroneously cached content may or may not be valuable for an attacker. As long as caching happens contrary to the informed instructions of the website owner, an *exploitable* WCD vulnerability exists.

For example, some pages with non-sensitive content may include dynamic parts containing dates, server response time metrics, or email obfuscation strings. If these pages are normally not cacheable, but with a WCD attack they are cached, this is a true positive for our purposes, regardless of the value of the leaked content. The server & cache combination interacts in a hazardous manner, and a future update to the page with sensitive information would have a security impact.

False Positives. Our definition of false positives directly follows from the above. Any finding that does not involve accidental caching of non-cacheable content is a false positive.

While this definition remains a constant, the particular *reasons* for false positive findings are closely tied to the WCD detection mechanism used. In CC, false positives are due to markers that a web application *intentionally* reflects in its responses. Even when there is no successful WCD attack taking place, the marker presence incorrectly signals to the crawler that sensitive information has leaked. Identifying such false positives requires a manual analysis of every finding and assessing whether the markers are returned due to WCD.

DE probes a page with a WCD payload, and checks whether the page is dynamic and whether it is cached. If both are true, it flags this as a finding. However, this detection mechanism cannot distinguish between *explicitly* and *erroneously* cached dynamic content.

Dynamic pages may still be explicitly configured to be cacheable by the website owner. In other words, the page would be cached even when requested normally, without a WCD attack. This may be due to aggressive server performance optimizations; for example, some non-sensitive dynamic objects could be allowed to be served from a cache, perhaps with a short TTL, even if they go stale. Alternatively, there could be human error; the website owner may have accidentally configured a dynamic page for caching – even though this is not an informed decision, it is still an explicit instruction. Regardless of the circumstances, DE would incorrectly flag the situation as a successful WCD attack.

One advantage of DE over CC is that its false positives can be identified and removed automatically, without human analysis. This is a trivial check shown in Algorithm 2. Specifically, we take each URL DE flags as vulnerable, request it twice normally, *without using a WCD payload*, and use the same header heuristics to test whether the second response was served from the cache. A cache hit means that the URL is still cached when there is no attack, hence a false positive. This check can also be integrated into our methodology (Algorithm 1, lines 1-3) with no added traffic load.

Algorithm 2: Test if a DE finding is a false positive.

input : URL
1 $result \leftarrow get(URL)$;
2 $result \leftarrow get(URL)$;
3 **if** $result.cache = HIT$ **then**
4 **return** False positive;
5 **return** True positive;

False Negatives. DE relies on cache status headers to determine whether our WCD attempts indeed result in the prerequisite cache miss followed by a hit. Because cache status reporting mechanisms are not standardized, servers may return headers unknown to DE, or no headers at all. Furthermore, by design, DE does not authenticate to websites, and hence cannot test pages behind authentication gates. As a result, DE is bound to miss WCD vulnerabilities in the wild. The impact of false negatives is not trivial to quantify; there exists no ground truth. Thus, our results should be interpreted as a lower bound on vulnerabilities.

4 Comparative Evaluation

We now present the results of our first experiment, where we run both DE and CC on a dataset of 404 websites for a comparative evaluation.

4.1 DE with Authentication

In doing this exercise, we are primarily interested in understanding how our scheme compares to the marker injection approach; however, there is a confounding factor in this experiment: DE cannot access pages behind authentication gates, whereas CC was specifically designed to test those pages *only*. Therefore, in order to investigate both the impact of the protocol change and authentication state on WCD detection efficacy, we introduce a third methodology, called $\mathsf{DE}_{\mathsf{auth}}$.

$\mathsf{DE}_{\mathsf{auth}}$ is a hybrid approach between DE and CC. It uses our novel detection scheme at its core, but like CC, requires an account to be manually created on the website so that the attack URL is requested (Algorithm 1, lines 6-7) with valid authentication cookies, simulating a logged in victim clicking on the malicious link. There are no other changes; $\mathsf{DE}_{\mathsf{auth}}$ probes the cache contents with an unauthenticated request like before, simulating an attacker (Algorithm 1, line 9).

4.2 The Experiment

We implement CC as described by Mirheidari et al. [36] and our two new schemes inside HTTP crawlers, and perform one crawl with each for a total of three runs. We set up our crawler to visit pages on any subdomain we may discover on the target website, and test at most 500 URLs on each FQDN.

We test each page with 12 attack URLs utilizing distinct WCD payloads. These include the original invalid path extension technique we illustrated in Figure 1, 4 path confusion techniques Mirheidari et al. proposed that exploit URL encoding discrepancies, and a further 7 novel encoding tricks we devise. We do not aim to position these new techniques as a scientific contribution; however, they are valuable for practical bug hunting situations. Readers can refer to Appendix A for examples and a breakdown of our findings for each.

We draw our crawl seed pool of 404 websites from the Alexa Top 100K. We choose these targets due to the marker injection requirements/limitations of CC, by following the general protocol described in "Cached and Confused". Specifically, we first crawl the front pages of Alexa Top 100K, and identify websites that support standard Single Sign-On schemes by searching for links containing keywords (e.g., login, register) and OAuth & OpenID Connect parameters. We then manually filter out websites that require sensitive credentials such as social security numbers or bank accounts for account creation. We end up with 404 websites, create accounts on them, inject markers into user-editable fields, and collect session cookies for each to be used by CC and $\mathsf{DE}_{\mathsf{auth}}$. This process necessarily yields a data set that carries the same biases as the one used in "Cached and Confused"; this is another limitation of CC, and it has no material impact on our comparative analysis.

We configure the DE and $\mathsf{DE}_{\mathsf{auth}}$ crawlers to record the page differences during dynamic content checks for websites flagged as vulnerable, so that we can scan these with regular expressions to detect common categories of sensitive data that may be leaked by the attack.

In all of our experiments, we flag a tested site as vulnerable if it contains at least one URL impacted by WCD. We believe this is the most relevant metric for our purposes that also supports our research goals. In practice, our crawler often finds multiple vulnerable URLs on each target website. However, without an in-depth manual analysis of each finding, we cannot accurately determine whether these vulnerabilities truly stem from distinct caching configuration issues, or whether the different URLs in fact correspond to unique pages. This analysis is not feasible or essential for our research.

4.3 Results

Table 2 shows the results of our experiments with each methodology, where we detected a combined total of 123 websites vulnerable to WCD. Table 3 presents a breakdown of the leaked data we found on these sites.

True Positives. The true positive findings confirm our hypothesis: Markers are severely limiting as a WCD detection approach. Even though our dataset is specifically biased toward websites that *must* support marker injection, many otherwise vulnerable pages did not reflect those markers. In fact, CC could only test 244 (60.40%) of the websites, but the remaining did not have any pages with a marker present. As a result, CC identified only 18 vulnerable websites in our experiments, whereas $\mathsf{DE}_{\mathsf{auth}}$ and DE performed considerably better at 115 and 104 hits respectively.

$\mathsf{DE}_{\mathsf{auth}}$ had a slight edge over DE. As one might expect, the difference was due to the vulnerable pages behind authentication gates, which DE cannot access. For example, we manually confirmed that a vulnerable billing settings page on a target website was detected by $\mathsf{DE}_{\mathsf{auth}}$, but DE was redirected to a secure login page when testing the same URL.

Likewise, CC found 7 vulnerabilities that DE missed thanks to its access to authenticated pages; but, in addition, it caught 2 unique vulnerabilities that even $\mathsf{DE}_{\mathsf{auth}}$ missed. We verified that in one case this was due to the target website returning no cache status headers, defeating our new scheme. The other case appears to be a vulnerability that was fixed between our two experiment runs.

Finally, DE found 5 unique vulnerabilities that neither authenticated approach identified. We verified that these cases were due to the websites either explicitly sending cache control headers that prevent caching, or quietly ignoring all cache directives, when we attached a cookie to the request. As we

Table 2: WCD detection performance, i.e., the number of websites flagged as vulnerable, for each methodology. Percentages are calculated over the entire crawl set of 404 sites.

	CC		DE_{auth}		DE		Combined	
Total Detections	21	(5.20%)	134	(33.17%)	129	(31.93%)	160	(39.60%)
True Positives	18	(4.46%)	115	(28.47%)	104	(25.74%)	123	(30.45%)
False Positives	3	(0.74%)	19	(4.70%)	25	(6.19%)	37	(9.16%)
Unique True Positives	2	(0.50%)	13	(3.22%)	5	(1.24%)	—	

Table 3: The number of vulnerable websites found to leak common categories of sensitive data by each methodology. There may be multiple leaks on a given website; columns do not add up to totals. Percentages are calculated over the total number of true positives for each methodology.

	CC		DE_{auth}		DE	
CSRF Token	4	(22.22%)	35	(30.43%)	39	(37.50%)
CSP Nonce	0	(0.00%)	1	(0.87%)	1	(0.96%)
OAuth State	0	(0.00%)	3	(2.61%)	2	(1.92%)
Session ID	2	(11.11%)	3	(2.61%)	3	(2.88%)
Personal Information	18	(100.00%)	16	(13.91%)	0	(0.00%)
Total Leaks						
Sensitive	18	(100.00%)	36	(31.30%)	39	(37.50%)
Potential		—	56	(48.70%)	50	(48.08%)
Harmless		—	23	(20.00%)	15	(14.42%)

discussed in Section 2, bypassing caching rules based on the presence of authentication cookies is a common option web caches provide to prevent hazardous caching. The unauthenticated DE scheme successfully defeated that protection.

False Positives. Recall that the false positives of DE and DE_{auth} can be eliminated automatically. However, we choose to present a clear breakdown of all false positives here to highlight the differences between CC and our new schemes. We apply our automated check to identify the false positives for DE and DE_{auth}, and perform a manual inspection of the context around the reflected markers for CC.

DE and DE_{auth} both had higher false positives compared to CC. As discussed, this was due to their inability to distinguish between explicitly and erroneously cached dynamic content. While CC was more reliable in this department, some markers were indeed intentionally reflected in all responses from the web application as we previously explained, and their presence did not imply WCD. For example, one website publicly listed its recent visitors, one of which was our marked username. CC falsely flagged this as a vulnerability.

Leaks. To correctly interpret the data in Table 3, recall that a WCD vulnerability can only result in a damaging data leak if there is sensitive data on the page to begin with. In our analysis, we found that some vulnerable websites did not contain such data, and the dynamic content leaked in the cache was *harmless* (e.g., timestamps, email obfuscation strings). Other websites did contain seemingly-randomized values that may *potentially* be sensitive, but these did not match any patterns of common sensitive tokens. Unfortunately, we are not in a position to reason about this potentially-sensitive category without a white-box understanding of the impacted websites' backend logic. We reiterate that all cases still stem from exploitable, true positive WCD findings, albeit some without immediate consequences. We present a breakdown of these totals at the bottom section of Table 3. Also note that, for CC, detections are due to markers known to populate sensitive fields, and therefore all findings are sensitive by definition.

The top slice of Table 3 presents a breakdown of the leaks in the sensitive category, once again highlighting the differences between each approach. CC primarily detected personal information leaks, but a small number of other security tokens were present on the same vulnerable pages by happenstance. DE_{auth} also detected 16 out of these 18 leaks without relying on markers, and myriad other sensitive leaks. DE performed similarly well for security tokens, but could not find personal information leaks without access to authenticated pages.

4.4 Summary

This experiment answers our first research question (Q1), showing that the marker injection approach is limited by both its attack surface coverage and the variety of leaks it can detect. Overall, identicality and header heuristics enable considerably better WCD detection. We also partially answer (Q3), demonstrating that leaks of non-personal sensitive data with WCD are practicable. We still need to investigate the implications of this finding in the upcoming sections.

That being said, the idea of using an authenticated crawling approach still holds merit. Both CC and DE_{auth} perform well with detecting personal information leaks, whereas DE is inherently unsuitable for the task. Where the setup overhead is manageable (e.g., when penetration testing one's own environment), DE_{auth} or perhaps a combination of all three approaches would expose the most vulnerabilities.

Nevertheless, DE remains the only viable option for a large-scale measurement, with its good detection performance and zero setup overhead. Equipped with this knowledge, we proceed with our experiment on the Alexa Top 10K. The findings in this section are already alarming, with 30.45% of our data set containing WCD vulnerabilities – well above the estimations in "Cached and Confused".

Table 4: The number of websites containing at least one WCD vulnerability, and websites that leak common categories of sensitive data. Percentages are calculated over the entire crawl set of 10K sites.

Vulnerable Sites	1188	(11,88%)
CSRF Token	436	(36.70%)
CSP Nonce	13	(1.09%)
OAuth State	34	(2.86%)
Session ID	63	(5.30%)

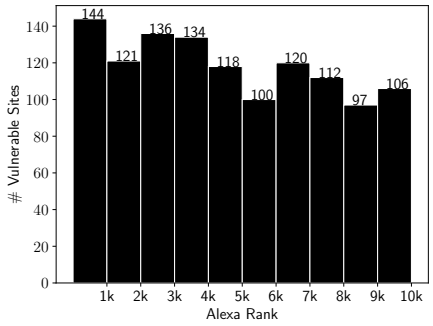

Figure 2: The distribution of vulnerable websites with respect to their Alexa ranking in 1K bins.

5 Large-Scale Experiment with DE

We now present our final experiment, where we run DE on the entire Alexa Top 10K, and describe concrete exploitation scenarios demonstrating real-life impact.

5.1 The Experiment

This experiment generally follows the previously established protocol, except for two important changes.

First, we enable the automated false positive filtering outlined in Algorithm 2, therefore eliminating all false positives in our results. *All numbers we report in this section represent true, exploitable WCD vulnerabilities.*

Second, we relax our definition of true positives by choosing not to test pages containing known harmless dynamic components. It is true that these pages may still be vulnerable to WCD, and while that may not be an immediate threat today, it may lead to a real-life exploit if the page is updated with sensitive content in the future. However, we opt to forgo testing these as a performance trade-off due to the limitations of our crawler resources and to minimize the traffic we generate. Specifically, during Step 1 of DE, we apply pattern matches on the dynamic components we find during identicality checks. If we detect a known email obfuscation mechanism, web analytics script, Edge Side Includes tag, timestamp, or error page that reflects our WCD payload, we conclude that the content is non-sensitive, and abort the test.

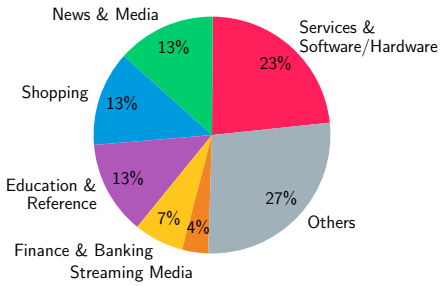

Figure 3: Content categories for the vulnerable websites. A website may be labeled with multiple categories.

5.2 Results

Table 4 shows our findings. As a result of the aforementioned changes to the experiment protocol, we no longer need to report false positives or harmless data leaks – all flagged websites have true positive findings, and leak known or potentially sensitive values. We also do not have personal information leaks as DE cannot automatically detect them; however, we will demonstrate later that these findings assist us in finding personal information leaks upon further analysis.

1188 websites among the Alexa Top 10K contain WCD vulnerabilities. This 11.88% incidence is significantly lower than the 30.45% we observed in the previous experiment; but we emphasize that the two results are not comparable. The previous dataset is non-uniformly drawn from the Alexa Top 100K based on the viability of marker injection; it is heavily biased. This larger dataset and the experiment have fundamentally different characteristics. Here, we study the most popular 10K websites likely to attract more attention from bounty hunters and attackers, and therefore discover and mitigate their vulnerabilities quickly. We also filter out the harmless leaks and report a looser lower-bound on vulnerabilities.

Figure 2 presents the distribution of vulnerable websites with respect to their Alexa ranks, exhibiting a fairly uniform, rectangular shape with a slight right skew. This suggests that WCD is pervasive among the websites in our dataset with no strong connection to their popularity ranking.

Figure 3 shows a breakdown of the vulnerable website content categories, as determined by multiple domain classification services and aggregated by us. These services perform a fuzzy classification, and we only report percentages to avoid giving the impression that the categories are definitive. Approximately a quarter of impacted websites involve financial data and transactions, suggesting WCD may cause direct monetary loss. Another quarter includes cloud service providers and software vendors, showing that attacks could have far-reaching consequences via supply chain attacks. News outlets, wikis, blogs, and document stores appear to be disproportionately impacted; this might be a consequence of their hosting large static objects, and hence heavy cache use.

6 Security Impact & Case Studies

Our findings already imply that the leaked sensitive tokens may be abused by an attacker to break the security mechanisms each support. For instance, leaked CSRF tokens enable confused deputy attacks, CSP nonces break defenses against inline JavaScript inclusions, and OAuth state parameters & session IDs enable hijacking victim accounts or stealthily logging victims into attacker-controlled accounts.

However, the implications of our findings extend beyond these basic attacks. In this section, we present real-life case studies drawn from our experiment, and provide insights into the less obvious damage potential of WCD. These discussion points also enable us to affirmatively answer our final research question (Q3), demonstrating that WCD has ramifications distinct from personal information leaks.

Due to the excessive number of vulnerabilities we identified, it is not feasible to investigate all findings systematically. The below scenarios represent an arbitrary list of real-world attacks that nevertheless demonstrate the severity of WCD. We chose these particular targets for manual exploration motivated by the website owners' presence on vulnerability management platforms, so that we could rapidly communicate and help mitigate any issues. All attacks described below were carried out with a test user, no actual Internet users were targeted or harmed.

Leaked Tokens Lead to Standard Attacks. We first describe two representative attacks made possible by stealing the sensitive tokens listed in Table 4 via WCD to give readers assurance that the impact is practical.

We found a popular travel & lodging reservation platform to leak session IDs. We were successfully able to use this stolen token to hijack customer service chat sessions of an unauthenticated user. The same attack translated to authenticated users as well; when a logged-in user visited the WCD exploit link, we were able to hijack their entire session and access complete booking details.

In another instance, we identified that the error pages on Mozilla Thunderbird's add-ons portal were vulnerable, and they contained registration and login links with OAuth state parameters. By stealing this value we launched a *Login CSRF* attack [46], which allowed us to trick a victim into unknowingly logging into an account we controlled, hence enabling us to view their activity and the information they enter. Mozilla fixed the issue within 24 hours of our notification.

These attacks demonstrate that sensitive token leaks on publicly accessible pages pose a real threat to unauthenticated visitors of a website as well as logged in users. As an additional empirical observation, a plethora of other traditional CSRF and session hijacking attacks were possible via WCD, but we noticed that damage was sometimes contained thanks to layered defenses such as referrer checks and captchas. This once again highlights the importance of a defense-in-depth strategy for practical web security.

WCD Leads to Cache Poisoning. WCD is a specialized subcategory of cache poisoning attacks, where a cache is tricked into storing and leaking sensitive data. That being said, the underlying mechanism for exploitation remains the same for all such cache attacks: content is erroneously cached. This implies that the vulnerable websites we detected may be exposed to other varieties of cache attacks, regardless of whether they immediately leak any sensitive data.

We found one such instance to impact a major American payment processor. Many pages on this website were impacted by a *reflected* cross-site scripting (XSS) vulnerability, where the value of the `X-Forwarded-Host` header included in requests was printed on the page without output sanitization. This enabled arbitrary script injection attacks.

As with many reflected XSS attacks, the avenues for exploitation would normally be limited. However, this website was also vulnerable to WCD. An attacker could combine the two vulnerabilities, and consequently cause the fronting cache to store the response together with the reflected XSS payload. This escalates the attack to a *stored* XSS, where the injected malicious payload is now automatically served from the cache to unsuspecting clients visiting the website.

This attack illustrates that WCD has dire consequences even when the website has no sensitive data to leak. Identifying such caching hazards is key to preventing complex, non-obvious system issues that may be lying dormant.

Token Leaks Correlate to Personal Information Leaks. DE is not designed to catch personal information leaks. However, our manual analysis shows that the presence of a WCD vulnerability on a public page is often indicative of more WCD issues that impact pages protected behind authentication gates, and therefore endanger personal information, too.

While we cannot scientifically quantify the incidence or reasons without a dedicated study, one intuitive explanation is that there is no fundamental difference between caching misconfigurations that lead to WCD vulnerabilities affecting authenticated and unauthenticated victims. Thus, a caching rule that leads to erroneous content storage on a public page may enable the same attack on a protected page in the absence of a session or cookie-based cache bypass mechanism.

We selected 55 websites flagged by DE that support user accounts, implying that they contain personal information. We created test accounts on these websites, and attempted WCD attacks on pages that require authentication for access. In 10 out of 55 cases, we were successfully able to cause personal information fields to get cached. To provide insights into the type of information that could be leaked, these were well-known websites including a domain registrar, a travel reservation platform, a job application & company review portal, an online course provider, a security product vendor, and a cryptocurrency exchange.

While this is not conclusive evidence, 18% is a non-negligible success rate. This suggests that our approach of detecting WCD vulnerabilities by performing checks on pub-

licly accessible pages do not completely forfeit the opportunity to detect personal information leaks. Website owners should carefully examine vulnerabilities lest they remain exploitable in different authentication contexts.

WCD Poses a Supply Chain Issue. Recently, highly-publicized cybercrime campaigns such as the Magecart attacks [45] and the SolarWinds incident [45] have put a spotlight on supply chain attacks, alerting the security community to the widespread damage one vulnerable supplier or vendor may cause to the Internet ecosystem. In our experiment, we found that supply chain attacks are not limited to the traditional malicious code inclusion vectors, and that a single vulnerable online service provider with a caching hazard can expose many websites to WCD.

We identified a multitude of vulnerable URLs in our results that share an identical subdomain and similar path components (i.e., *support*.example.com/*common-pattern*). Upon manual inspection, we determined these pages to be integration points with a popular customer service and support management platform. Due to the WCD vulnerabilities present on this vendor's platform, many (or, potentially all) of their customers were also impacted under their respective domains. To demonstrate the weight of the issue, 399 out of the 1188 websites we flagged were expressly due to this vulnerability, and 57 websites were impacted by it in addition to other WCD vectors, bringing the total to an astounding 456.

We found similar cases, involving three vendors providing customer community management, social media integration, and discussion board services. These were less prevalent in our findings, each impacting less than 10 websites. Nonetheless, this illustrates that WCD exhibiting itself as a supply chain vulnerability is not an isolated incident. As evidenced by the alarming numbers, the security community would benefit from investigating supply chain attacks in a broader scope in the face of novel web cache attacks.

7 Bounty Hunting with WCD

All of the WCD vulnerabilities we have reported in this work are *exploitable*, causing unintended content leaks into a public cache. However, a working exploit does not always equate to real-life *damage*; for instance, the vulnerable website may not process any sensitive data. Beyond the case studies we discussed above, we do not aim to measure such damage at scale in this work – that requires a manual analysis of each application and its data. However, we present a final empirical study to provide insights into the incidence of damaging exploits, and how vulnerable websites mitigate damage.

We perform this study on a separate dataset of 48 random vulnerable websites identified by running DE on domains listed on the bug bounty platforms Hackerone, BugCrowd, Intigriti, and YesWeHack. This is not an arbitrary choice; obtaining the evidence we seek requires active exploitation of websites which provide a safe harbor for such testing in their infrastructure and reward bounties for damages that they acknowledge as real. We limit the scope by allowing DE to crawl a maximum of 50 pages on each website, and all manual analysis is performed by one researcher capped at a few hours of work. Therefore, readers should interpret our findings as the result of a best-effort attempt, but not a comprehensive penetration test.

Out of the 48 vulnerable websites, we were able to launch damaging attacks on **9**. These are similar to the case studies described above, and we omit their detailed discussion. 4 vendors paid out bounties, 2 acknowledged the issues but informed that another researcher reported it earlier, and the remaining 3 are still under evaluation.

Below is a breakdown of the reasons why we could not escalate the remaining WCD exploits to a damaging attack.

We were able to fully analyze the context around 24 websites, but there was no data valuable for an attacker. Another 10 websites did not allow us to explore the entire application, either disallowing public account creation, or requiring private information (e.g., a social security number) to proceed. We only analyzed these partially, and found no valuable data.

3 websites leaked sensitive tokens, but this was not sufficient on its own. For example, a CSRF attack was stopped thanks to layered defenses of referrer checks and captchas; a CSP nonce leak was useless as there was no XSS vulnerability to abuse it. 2 websites pulled sensitive data over an API at the browser side, therefore nothing damaging was cached.

This is decidedly a limited view into how WCD exploits escalate into end-to-end attacks. In an adversarial scenario, attacks may also be impeded by short cache eviction times, and cache locality in the case of distributed caches, as previously measured in "Cached and Confused". Regardless, we hope these added insights help qualify the core findings in our large-scale experiment. *Not every instance of WCD is an immediate threat; however, they are still exploitable vulnerabilities exposing applications to unpredictable risks.*

8 Ethical Considerations

No Harm to Users or the Internet. We carefully designed the methodologies and experiments in this paper to prevent a negative security impact on the tested websites or their users.

In particular, we never poison caches with malicious content, and never target Internet users with WCD. The personal information leaks explored in the paper are our own markers, and other sensitive tokens are the secrets that websites generate for our own test clients. In all case studies we play the role of the victim and attacker; we never target other users or launch exploits that persistently impact the target websites.

Furthermore, our path confusion techniques utilize randomized file names, meaning that cache keys corresponding to the erroneously cached content cannot feasibly be predicted or accidentally accessed by others. This is an added safeguard against confusing the websites' users. Even if the caches were

accessible, there would be no danger to users; we never inject malicious payloads into the caches in the first place.

Coordinated Disclosure. We are committed to following coordinated disclosure procedures that exceed the established best practices. Unfortunately, with thousands of findings, especially those involving systematic issues that cannot be solved by deploying a common patch and therefore are out of scope for CERT assistance, this is not a straightforward process. The infeasibility of common approaches to large-scale vulnerability disclosures were documented in literature [34, 43, 44].

We adopted the guidance in the above literature to reach out to as many impacted parties as possible. We collected security contacts that were 1) disclosed on vulnerability management and bug bounty platforms, 2) compiled into open-source security lists, 3) found in WHOIS records, 4) published on the homepages of vulnerable websites. For the remaining 529 websites we could not identify a security contact for, we emailed the generic inboxes *security@* and *privacy@*.

These exhaust the viable options available to us. The cases that may not be covered by the above require deep exploration of the website or filling out non-automatable forms, which we could only do on a best-effort basis.

We began notifications promptly after finalizing the experiments, and gave website owners over 3 months to implement mitigations before a public disclosure. Our notification emails included our affiliation, a summary of WCD and our experiments, and a report of the findings pertinent to each party.

9 Discussion & Conclusion

We directly tackled the limitations of the state-of-the-art approach in WCD vulnerability detection, subsequently conducting the largest-scale WCD measurement over 10K websites.

Let's revisit our research questions and summarize findings.

- (Q1) We demonstrated through our comparative experiment that our new methodology DE addresses both the coverage (P1) and scalability problem (P2), and it can indeed significantly outperform CC. However, we also showed that CC and the authenticated variation of our scheme, $\mathsf{DE}_{\mathsf{auth}}$, open up opportunities to identify additional vulnerabilities. Where scalability is not a concern, a combination approach is ideal.

- (Q2) We showed with our large-scale experiment that over 4 years after the conception of the attack, and 2 years after the experiments in "Cached and Confused," WCD is still distressingly pervasive. This aligns with the popularity of the attack on bug bounty platforms – and likely miscreant activity that goes unnoticed.

- (Q3) Our experiments and case studies illustrated that there is an abundance of sensitive security tokens present on publicly accessible pages, which can be stolen via WCD to bypass standard defenses and facilitate real-life attacks. Many websites that leak such tokens are evidently impacted by WCD in more than one way, exposing flaws that lead to further attacks and leaks. These observations, combined with the significant performance advantage of DE over CC, suggest that focusing on personal information sources and sinks for WCD detection is not the most effective detection strategy, even when testing individual websites in a controlled setting.

Our findings sufficiently address the research questions we set out to explore, and we contribute novel insights into the scale and impact of the problem. The methodology we present will help website owners test their own systems for vulnerabilities, and researchers to run experiments with ambitious scopes. However, another implication of this work is that attackers, too, can quickly identify vulnerabilities en masse. WCD, and web cache attacks in general, require immediate attention from the security community for a robust solution.

Before we conclude, we reiterate that WCD is a system problem. Individual components such as the clients, web servers, proxy services, or CDN providers are not necessarily faulty in isolation; their complex interactions give rise to unexpected and dangerous caching decisions. One corollary of these circumstances is that our findings do not implicate the developers and operators of these individual components. But, perhaps the more critical take away is that website owners cannot rely on traditional vulnerability management and software testing processes to eradicate these vulnerabilities – there is often no unit test to run, no signature to check, no CVE to track, and no patch to deploy. It is not yet clear whether mapping complex traffic flows and analyzing them holistically for cache attacks is feasible, or even possible. That remains an open challenge for the security research community, and in light of the resurging popularity of web cache attacks, we believe it has already become a pressing line of investigation.

In the meantime, our work presents one key takeaway for website owners who are inevitably getting more familiar with the escalating web cache attacks: CDNs and caching proxies are powerful technologies in an already complex ecosystem. Simple caching rules can have far-reaching effects, and making assumptions about the cacheability of objects based on their public exposure to the Internet alone is, evidently, *unsafe*. Website owners should carefully consider (and test) the security implications of changes to their caching infrastructure, and exercise caution when using blanket rules such as those that cache all objects served from a given endpoint or all files with a given extension.

Acknowledgments. We thank our fellow researcher Bahruz Jabiyev for his valuable input, and our shepherd Stefano Calzavara for championing our paper. This work was supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe, the National Science Foundation grant CNS- 1703454, and by Secure Business Austria.

References

[1] Akamai Developer. EdgeWorkers. https://developer.akamai.com/akamai-edgeworkers-overview.

[2] Akamai Technologies. Facts & Figures. https://www.akamai.com/us/en/about/facts-figures.jsp.

[3] Amazon Web Services (AWS). HTTP Desync Guardian, 2020. https://github.com/aws/http-desync-guardian.

[4] Apache HTTP Server Project. Caching Guide. https://httpd.apache.org/docs/2.4/caching.html.

[5] BuiltWith. BuiltWith Technology Lookup. https://trends.builtwith.com/CDN/Content-Delivery-Network.

[6] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In ACM Conference on Computer and Communications Security, 2016.

[7] Jianjun Chen, Jian Jiang, Xiaofeng Zheng, Haixin Duan, Jinjin Liang, Kang Li, Tao Wan, and Vern Paxson. Forwarding-Loop Attacks in Content Delivery Networks. In The Network and Distributed System Security Symposium, 2016.

[8] Cloudflare. Creating Cache Keys. https://support.cloudflare.com/hc/en-us/articles/115003206852s.

[9] Cloudflare. The Cloudflare Global Anycast Network. https://www.cloudflare.com/network/.

[10] Cloudflare. Understanding Cloudflare's CDN, 2021. https://support.cloudflare.com/hc/en-us/articles/200172516-Understanding-Cloudflare-s-CDN.

[11] Cloudflare Docs. Cloudflare Workers Documentation, 2021. https://developers.cloudflare.com/workers/.

[12] Akamai Documentation. Caching, 2021. https://learn.akamai.com/en-us/webhelp/api-gateway/api-gateway-user-guide/GUID-B717E657-4C07-4B76-934A-36F1C40F91AE.html.

[13] Fastly Documentation. Configuring Caching, 2020. https://docs.fastly.com/en/guides/configuring-caching.

[14] Evan Custodio. Smuggler, 2020. https://github.com/defparam/smuggler.

[15] Fastly. Compute@Edge. https://www.fastly.com/products/edge-compute/use-cases.

[16] Fastly. Fastly Developer Hub – X-Cache. https://developer.fastly.com/reference/http-headers/X-Cache/.

[17] Fastly. Fastly Network Map. https://www.fastly.com/network-map.

[18] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. IETF – RFC 7234, 2014. https://www.rfc-editor.org/info/rfc7234.

[19] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-Resistant Communication Through Domain Fronting. In Privacy Enhancing Technologies, 2015.

[20] Omer Gil. Web Cache Deception Attack. Black Hat USA, 2017. https://www.blackhat.com/us-17/briefings.html#web-cache-deception-attack.

[21] Omer Gil. Web Cache Deception Attack, 2017. https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html.

[22] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation. In IEEE International Symposium on Reliable Distributed Systems, 2018.

[23] Run Guo, Weizhong Li, Baojun Liu, Shuang Hao, Jia Zhang, Haixin Duan, Kaiwen Sheng, Jianjun Chen, and Ying Liu. CDN Judo: Breaking the CDN DoS Protection with Itself. In The Network and Distributed System Security Symposium, 2021.

[24] Shuai Hao, Yubao Zhang, Haining Wang, and Angelos Stavrou. End-Users Get Maneuvered: Empirical Analysis of Redirection Hijacking in Content Delivery Networks. In USENIX Security Symposium, 2018.

[25] John Holowczak and Amir Houmansadr. CacheBrowser: Bypassing Chinese Censorship Without Proxies Using Cached Content. In ACM Conference on Computer and Communications Security, 2015.

[26] Arbaz Hussain. Auto Web Cache Deception Tool, 2017. https://medium.com/@arbazhussain/auto-web-cache-deception-tool-2b995c1d1ab2.

[27] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In ACM Conference on Computer and Communications Security, 2021.

[28] Lin Jin, Shuai Hao, Haining Wang, and Chase Cotton. Your Remnant Tells Secret: Residual Resolution in DDoS Protection Services. In IEEE/IFIP International Conference on Dependable Systems and Networks, 2018.

[29] James Kettle. Practical Web Cache Poisoning. PortSwigger Web Security Blog, 2018. https://portswigger.net/blog/practical-web-cache-poisoning.

[30] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog, 2019. https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.

[31] James Kettle. Web Cache Entanglement: Novel Pathways to Poisoning. PortSwigger Research, 2020. https://portswigger.net/research/web-cache-entanglement.

[32] James Kettle. HTTP/2: The Sequel is Always Worse. Black Hat USA, 2021. https://www.blackhat.com/us-21/briefings/schedule/#http2-the-sequel-is-always-worse-22668.

[33] Amit Klein. HTTP Request Smuggling in 2020 – New Variants, New Defenses and New Challenge. Black Hat USA, 2020. https://www.blackhat.com/us-20/briefings/schedule/#http-request-smuggling-in---new-variants-new-defenses-and-new-challenges-20019.

[34] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In USENIX Security Symposium, 2016.

[35] Chaim Linhart, Amit Klein, Ronen Heled, and Steve Orrin. HTTP Request Smuggling. Watchfire, 2005. https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf.

[36] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and Confused: Web Cache Deception in the Wild. In USENIX Security Symposium, 2020.

[37] NGINX. NGINX Content Caching. https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/.

[38] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In ACM Conference on Computer and Communications Security, 2019.

[39] PortSwigger. HTTP Request Smuggler, 2019. https://github.com/PortSwigger/http-request-smuggler.

[40] Apache HTTP Server Project. Apache Module mod_cache – CacheHeader Directive. https://httpd.apache.org/docs/2.4/mod/mod_cache.html#cacheheader.

[41] Johan Snyman. Airachnid: Web Cache Deception Burp Extender. Trustwave – SpiderLabs Blog, 2017. https://www.trustwave.com/Resources/SpiderLabs-Blog/Airachnid--Web-Cache-Deception-Burp-Extender/.

[42] Squid. Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[43] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. Didn't You Hear Me? — Towards More Successful Web Vulnerability Notifications. In The Network and Distributed System Security Symposium, 2018.

[44] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In USENIX Security Symposium, 2016.

[45] David Strom. What is Magecart? How this hacker group steals payment card data. CSO Online, 2019. https://www.csoonline.com/article/3400381/what-is-magecart-how-this-hacker-group-steals-payment-card-data.html.

[46] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. Large-Scale Analysis & Detection of Authentication Cross-Site Request Forgeries. In IEEE European Symposium on Security and Privacy, 2017.

[47] Sipat Triukose, Zakaria Al-Qudah, and Michael Rabinovich. Content Delivery Networks: Protection or Threat? In European Symposium on Research in Computer Security, 2009.

[48] Varnish. Varnish HTTP Cache. https://varnish-cache.org/.

[49] Thomas Vissers, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. Maneuvering Around Clouds: Bypassing Cloud-based Security Providers. In ACM Conference on Computer and Communications Security, 2015.

[50] World Wide Web Consortium (W3C). Cool URIs don't change, 1998. https://www.w3.org/Provider/Style/URI.html.

[51] Hadi Zolfaghari and Amir Houmansadr. Practical Censorship Evasion Leveraging Content Delivery Networks. In ACM Conference on Computer and Communications Security, 2016.

A Path Confusion Techniques

Table 5 presents examples for each path confusion technique we use when crafting the attack URLs in our comparative evaluation, and a breakdown of the findings for each. Table 6 shows a similar summary for the large-scale experiment.

Path Parameter refers to the original WCD technique proposed by Omer Gil, and the remaining 4 encoding techniques listed in the first group of rows were presented by Mirheidari et al. in their paper "Cached and Confused". The second group contains 7 additional path confusion techniques we propose here. While there are overlaps between the websites each technique can exploit, combining all 12 greatly increases the chances of exposing WCD vulnerabilities.

Disclaimer

The authors Seyed Ali Mirheidari and Kaan Onarlioglu are affiliated with Splunk Inc. and Akamai Technologies Inc., respectively, at the time of this publication. However, this research is not sponsored or carried out by either company. The work and results we present in this paper do not use any internal or proprietary company information, or any such information pertaining to the companies' customers.

Table 5: The number of vulnerable websites detected via each path confusion variation over 404 targets in our comparative experiment. The middle rule separates the previously known variations above from the new ones we introduce in this research below. Percentages are calculated over the total number of true positives for each methodology.

Path Confusion Technique	Example	CC	DE_{auth}	DE
Path Parameter	example.com/profile/not_a_file.css	13 (72.22%)	63 (54.78%)	62 (59.62%)
Encoded Newline	example.com/profile%0Anot_a_file.css	7 (38.89%)	90 (78.26%)	90 (86.54%)
Encoded Question Mark	example.com/profile%3Fname=valnot_a_file.css	8 (44.44%)	89 (77.39%)	87 (83.65%)
Encoded Semicolon	example.com/profile%3Bnot_a_file.css	9 (50.00%)	90 (78.26%)	90 (86.54%)
Encoded Sharp	example.com/profile%23not_a_file.css	9 (50.00%)	89 (77.39%)	88 (84.62%)
Encoded Slash	example.com/profile%2Fnot_a_file.css	8 (44.44%)	94 (81.74%)	96 (92.31%)
Double Encoded Newline	example.com/profile%25%30%41not_a_file.css	7 (38.89%)	90 (78.26%)	87 (83.65%)
Double Encoded Null	example.com/profile%25%30%30not_a_file.css	6 (33.33%)	87 (75.65%)	85 (81.73%)
Double Encoded Question Mark	example.com/profile%25%33%46not_a_file.css	8 (44.44%)	90 (78.26%)	86 (82.69%)
Double Encoded Semicolon	example.com/profile%25%33%42not_a_file.css	9 (50.00%)	89 (77.39%)	84 (80.77%)
Double Encoded Sharp	example.com/profile%25%32%33not_a_file.css	8 (44.44%)	89 (77.39%)	86 (82.69%)
Double Encoded Slash	example.com/profile%25%32%46not_a_file.css	7 (38.89%)	84 (73.04%)	88 (84.62%)

Table 6: The number of vulnerable websites detected via each path confusion variation in the large-scale measurement over the Alexa Top 10K. The middle rule separates the previously known variations above from the new ones we introduce in this research below. Percentages are calculated over the total number of findings.

Path Confusion Technique	Example	DE
Path Parameter	example.com/profile/not_a_file.css	618 (52.02%)
Encoded Newline	example.com/profile%0Anot_a_file.css	528 (44.44%)
Encoded Question Mark	example.com/profile%3Fname=valnot_a_file.css	801 (67.42%)
Encoded Semicolon	example.com/profile%3Bnot_a_file.css	863 (72.64%)
Encoded Sharp	example.com/profile%23not_a_file.css	526 (44.28%)
Encoded Slash	example.com/profile%2Fnot_a_file.css	559 (47.05%)
Double Encoded Newline	example.com/profile%25%30%41not_a_file.css	383 (32.24%)
Double Encoded Null	example.com/profile%25%30%30not_a_file.css	349 (29.38%)
Double Encoded Question Mark	example.com/profile%25%33%46not_a_file.css	387 (32.58%)
Double Encoded Semicolon	example.com/profile%25%33%42not_a_file.css	402 (33.84%)
Double Encoded Sharp	example.com/profile%25%32%33not_a_file.css	386 (32.49%)
Double Encoded Slash	example.com/profile%25%32%46not_a_file.css	365 (30.72%)