Persistent Browser Cache Poisoning

Matthias Vallentin
mavam@cs.berkeley.edu

Yahel Ben-David
yahel@cs.berkeley.edu

Abstract

Caching web content increases the performance on both the client side (by avoiding unnecessary requests) and the server side (by reducing bandwidth and I/O load). To this end, the HTTP protocol uses expiration and validation mechanisms to ensure that stale content is updated when necessary. It performs the corresponding integrity checks, however, only on the server side. Worse, current browsers lack proper cross-domain cache authentication. We exploit this absence of security checks, and show that an attacker who accesses the network through the same shared medium as the victim can modify cacheable content in the victim's browser without attracting the notice of the victim. This attack is particularly effective against web applications that offload a large part of stable functionality to the client's cache. To demonstrate the feasibility of the attack, we implement `air-poison`*, a proof-of-concept tool designed to be used in wireless networks that answers HTTP requests for JavaScript documents by prepending a malicious payload of the attacker's choice. Each time the victim loads the document, regardless of the security origin context, the malicious script executes and establishes a bidirectional JavaScript communication channel to a host controlled by the attacker. The systematic study of the severity of this threat is an interesting future avenue of research.*

1 Introduction

The purpose of browser cache is to store web content for performance reasons in order to both reduce the server load and avoid unnecessary requests for an unchanged resource. The continuing trend towards faster, richer, and more sophisticated web applications makes caching an indispensable component of fulfilling the user's expectations for responsiveness. Unfortunately, the HTTP protocol providing the caching mechanisms was not designed with security considerations in mind, which has opened it up to a variety of trivial attack vectors in the presence of an adversary.

In this paper, we discuss the practical security implications of web caching, and show that current caching practices violate basic integrity assumptions. Worse, the lack of per-site cache isolation in current browser implementations enables particular powerful attacks against web applications used by multiple sites. We present an attack to persistently implant executable content into a victim's browser cache, that merely requires that the attacker and victim to access the network via the same shared medium. While the concept of this type of attack has been around for some time [12, 20, 13], we are the first to examine it in context with the HTTP cache primitives, demonstrate the ease at which it can be conducted, and implement a proof-of-concept attack tool.

The remainder of the paper is structured as follows. After recapitulating the basics of HTTP caching in §2, we illustrate how to persistently poison the browser cache in §3. We then turn to `air-poison`, our proof-of-concept implementation of the attack and point out a particular attractive target in §5. Thereafter in §6, we describe ways how the attacker can get the necessary position to conduct the attack. We summarize related work in §7 and suggest avenues for future research in §8. Lastly, we close the paper with concluding remarks in §9.

2 HTTP Caching

The fundamental goal of caching is to improve performance. Clients benefit from caching by avoiding unnecessary requests when the resource has not yet expired, while servers benefit via a reduction of bandwidth and I/O load. Caching in HTTP [11] is based on two principal mechanisms, *expiration* and *validation*.

In order to allow clients to verify the freshness of a resource, the server specifies an expiration time in the future, using either the `Expires` header or max-age di-

rective within the `Cache-Control` header. A fresh resource does not need to be refetched, thereby avoiding unnecessary requests entirely. The server should choose an expiration time that is adequate for the resource. Setting an expiration time in the past signals a stale response, which the cache should always validate. Determining whether a response is stale involves comparing the freshness lifetime of the cache entry to its age.[1] A stale response must be validated, which we describe below.

After having determined that a cache entry is stale, the cache will validate the resource by asking the server whether or not the resource is still usable. For this purpose, HTTP 1.1 uses *conditional requests*, in which the client includes a *validator* that has been stored with the original cache entry. The server compares the received validator against the current validator, and, if these values match, responds with a 304 (Not Modified) status code, and returns the full response otherwise. HTTP 1.1 distinguishes between *strong* and *weak* validators. The strong validator changes by necessity when the entity changes, whereas the weak validator does not always change, e.g., only when significant semantic changes occur. Conditional requests differ from regular requests only in that they carry the validator in an extra header, which is often the `Last-Modified` header. The `Last-Modified` header is implicitly weak, as it offers only per-second granularity. As an alternative, the server can include an *entity tag*, which is a custom validator specified in the `ETag` header.[2] This enables strong validators, e.g., if the entity tag is calculated as a checksum of the resource. Entity tags therefore allow for a more generic and reliable validation. The current HTTP standard advocates for the use of a strong validator together with a `Last-Modified` header.[3]

3 Persistent Cache Poisoning

The goal of the attacker is to replace a piece of JavaScript in the victim's browser cache with a malicious one. In particular, the attacker seeks to *(i)* maximize the time until the victim validates the malicious cache entry, and *(ii)* avoid validation failures of conditional requests. In order to do so, the attacker must be able to forge the response from the web server, which is particularly easy in a Man-in-the-Middle (MITM) scenario. Multiple ways to get in such a position exist, which we discuss in §6. In order to convey the basic idea of our attack, we must first make the assumption that the attacker has managed to become a MITM and can observe, block, and manipulate traffic. We relax this assumption later and demonstrate a variation of the attack when the attacker is merely an eavesdropper in a wireless network – without the ability to manipulate ongoing communication, yet with the ability to inject packets.

3.1 MITM Cache Poisoning

Consider the scenario where a client requests a page from site A, which in turn includes a link to a JavaScript file hosted by server S; in other words, after receiving the page from A, the client opens a new connection to S to download the linked script, as illustrated by step ① in Figure 1. This script could be part of a widely used JavaScript library embedded by various different sites. Because such libraries change less frequently than the primary page content, S returns the script with the necessary `Cache-Control` headers in the HTTP response (②). When observing the response, the attacker prepends a malicious payload to the script and relays it back to the client (③), keeping the original cache control headers intact. Although the attacker could have ignored the client's response directly and returned merely the payload after observing the request, it is stealthier to prepend the payload to the original script, because the page in the client's browser can still make use of the provided functionality.

The combination of the attacker's payload and the original script now resides in the client's cache. As long as the cache entry is fresh, the client will not issue new requests for the script. Each time the client visits page A or any other page B that includes the same script, the malicious payload will be executed. For example, the payload could implement a JavaScript keystroke logger, or report detailed information about the client's plugins and browser version to deliver a 0-day exploit when the client is vulnerable (see §4). Because the attacker can modify the lifetime of the cache entry when returning the forged response, the script may reside in the cache for a long period of time – days, weeks, or even months.

However, there are several conditions that trigger a validation of the cache entry by issuing a conditional request, as represented by step ④ in Figure 1. Successful validation elicits a 304 response (⑤), which indicates that the client can keep using the resource. For example, Firefox validates the cache hitting reload [8], and our own experiments show that Safari validates cache entries upon startup. In future work, we plan to more closely examine the conditions that trigger validation.

[1] If both a `Cache-Control` header with `max-age` directive and an `Expires` header are present, the former overrides the latter; if neither header is present, the cache may use a heuristic to compute the freshness lifetime.

[2] When an `ETag` header appears in a response, the client must use it for subsequent conditional requests in the `If-Match` or `If-None-Match` header.

[3] To enable backwards compatibility with HTTP 1.0, which does not feature entity tags, clients should include entity tag and `Last-Modified` if both are present.

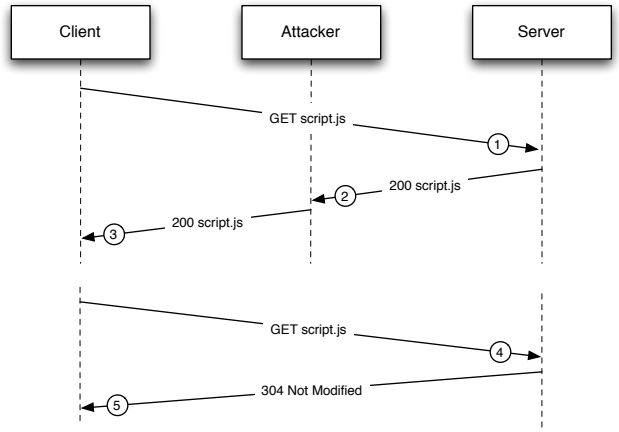

Figure 1: Persistent cache poisoning when the attacker has a Man-in-the-Middle (MITM) position. The server returns a cacheable response to which the attacker prepends a malicious payload. Later successful validation signals that the client can still use its existing (modified) response, even thought the attacker is not present anymore.

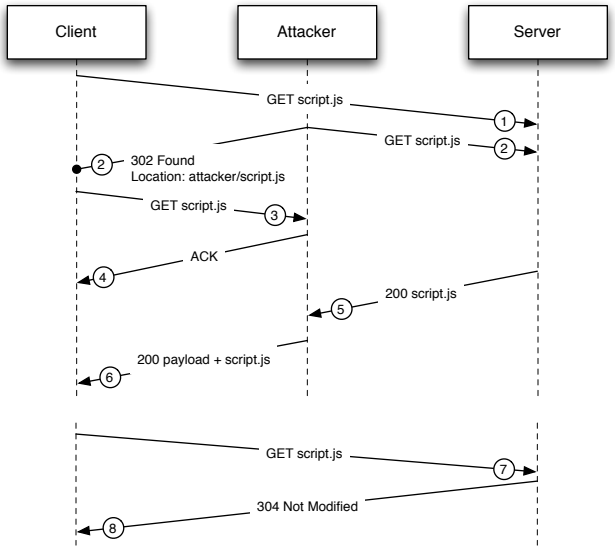

Figure 2: Poisoning the client's browser cache without a MITM position. The attacker employs a 302 redirect to delay the client until the real response arrives from the server and then returns the malicious payload together with the response.

3.2 Shared Medium Cache Poisoning

We now relax our assumptions about the MITM position of the attacker and present a generic attack that merely requires that a client and attacker access the network via the same shared medium, as commonly occurs within a wireless network. For unencrypted and WEP encrypted networks, the medium is truly shared, in the sense that each station can see the traffic of all other stations if in range. For WPA/2 encrypted networks, however, each station additionally negotiates an session key with the access point (AP) after joining the network with the pre-shared secret key known to all stations. Any station that observes the session key negotiation also sees the session key. Consequently, when the attacker joins the network, only ongoing sessions are protected from eavesdropping until the next re-keying procedure. From this point on, we assume the attacker can observe all frames from the client, but not block, delay, or manipulate the communication of the client with other stations.

The scenario we consider now is almost identical to the one in §3.1, with the exception that the attacker has to do more work to poison the client's cache due to the lack of the MITM position. Again, a client visits a site that includes a cacheable piece of JavaScript from another site, and the attacker's objective is to add a malicious payload to the script. This time, the attacker can neither manipulate the request of the client nor the response of the server; thus the only way to interfere is via traffic injection.[4] As shown in Figure 2, the client requests a file `script.js` from the server (①). After observing the request of the client, the attacker simultaneously fetches the same script and sends a terminating 302 redirect back to the client (②), indicating that the script is located at the attacker's machine. Because the attacker closes the connection after the redirect, the client does not consider the original response from the server anymore. The client then asks for the script from the attacker (③), who only sends back an ACK (④) in order to keep the client waiting until the response arrives from the server (⑤). Now equipped with the response, the attacker forwards it together with the reply to the client (⑥).

4 Implementation

We implemented the above attack in 400 lines of Ruby code. Our tool, called `air-poison`, uses the LORCON [14] library[5] for driver-independent 802.11 frame injection via the Linux mac802.11 stack [15]. In principle, `air-poison` tries to parse each frame seen in the air and check whether it is a HTTP GET request. If the

[4]The client's connection to the server is only protected by the TCP sequence numbers which the attacker can observe and use to generate valid traffic.

[5]LORCON currently only supports injection with unencrypted networks, although there is no conceptual obstacle to support injection in WEP and WPA/2 encrypted networks.

GET request matches a regular expression specified on the command line, `air-poison` forges an cacheable HTTP response with the following headers:

- `Cache-Control`: `max-age` set to one year in the future
- `Expires`: set to one year in the future
- `Last-Modified`: set to the current time

In addition, `air-poison` includes a `Connection` header set to `close` in order to signal the victim to terminate the TCP connection after processing the response. This avoids a situation where the real response from the web server would potentially override the attacker's response. After all, the HTTP standard recommends the use of the most recent answer in the case of the arrival of multiple responses, even if the previous answer is still fresh. At this point, `air-poison` simply injects the payload with the above headers without waiting for the original response to arrive. We plan to integrate response inclusion and 302 redirects when releasing `air-poison` to the public.

The user can use any payload with `air-poison`, which reads input from `STDIN` or via `-r` from file. Moreover, `air-poison` ships with a payload that registers the victim as a "zombie" with the BeEF framework [4]. When the payload executes, it establishes a bidirectional communication channel over JavaScript to the zombie, who can then be controlled by the attacker over a web front-end. For example, BeEF offers a variety of modules to fingerprint the browser and gather available plug-ins versions, log keystrokes, detect whether the zombie uses Tor [6], capture cookies, perform port scans in the local network, or exploit the browser through metasploit XMLRPC [16]. The communication channel exists until the victim navigates away from the site that included the payload.

To facilitate the exploitation process, we also implemented a DNS cache poisoning attack. The user can specify a regular expression that matches DNS requests, and answers them with a custom IP serving the payload. This is particularly effective because long-lived DNS cache entries are much harder to evict than JavaScript in the browser cache. It is also possible to tell `air-poison` to spawn a fake web server on the attackers machine that replies to any request with the payload. This is a powerful attack vector when first poisoning the DNS cache, and then redirecting the victim to the attacker's fake web server.

5 Case Study: Google Analytics

A particularly attractive target for the persistent cache poisoning attack is Google Analytics [2], a popular service for comprehensive tracking, reporting, and understanding of user's surfing behavior. The reports include information such as the amount of time a user stayed a page, the depth of the navigation, and the site to which the user went afterwards. Additionally, the decomposition of traffic sources further breaks the visits down into visits from search engines, referring sites, and direct traffic. Google Analytics is used by approximately 32.2% of the Alexa's list of the 10,000 most popular web sites [10].

To enable Google Analytics on a web site, the content provider includes a small piece of JavaScript that contains the site identifier and a link to an external script hosted on `google-analytics.com` with the bulk logic. The external script, named `ga.js`, has an expiration time of 7 days. However, when the browser validates the cache entry, Google returns a 304 status code indicating that the script has not changed since the end of November 2009. In other words, a malicious cache entry could reside for roughly 4 month in the browser while remaining undetected.

In general, all widespread web applications that offload a cacheable API to the client are equally susceptible. We intend to conduct a detailed evaluation about the prevalence of such cacheable APIs in the second project of this class.

6 Owning the Medium

In this section we summarize several techniques to bring the attacker into a MITM position, which is required by the attack described in in §3.1.

The vast majority of efforts in securing wireless networks have been traditionally focused on protecting the AP, without giving equal consideration to client-side security. The lack of mutual authentication exposes multiple difficult-to-defend attacks against clients. One fundamental issue shared by all major operating systems is probing for previously used APs in the list of preferred wireless networks. Due to the fact that 802.11 does not authenticate link-layer frames, *anyone* – not only the correct AP – can respond to these probes. Off-the-shelf tools [1] allow an attacker to reply to these probes and transparently impersonate the requested network to realize a full MITM scenario,[6] without the client even noticing its occurrence.

This transparent form of MITM is particularly devastating when an attacker gains control over random clients, such as on the street, in a coffee shop, at the airport, or even in the plane where a user does not expect to

[6]This attack can even be used to crack the WEP key of a network solely by interacting with the client [18] and capture WPA/2 handshakes to launch effective brute-force attacks [17]. Furthermore, such attacks could be carried out over distances using high-gain directional antennas.

```
var url = "http://HOST/beef/hook/beefmagic.js.php";
var script = "<script language='Javascript' src=" + url + "></script>";
document.write(script);
```

Figure 3: The default payload that ships with `air-poison`. The `HOST` running BeEF can be specified on the command line. The loaded file `beefmagic.js.php` installs bidirectional JavaScript communication channel to the victim, allowing the attacker to query the browser version, capture keystrokes, sniff cookies, or launch browser exploits. The channel persists until the victim navigates away from the site loading the payload.

be connected. Mobile devices are especially vulnerable to this attack, as the infection occurs unnoticed without the device even leaving the user's pocket. In general, any location with a high turnaround of mobile devices represents an attractive gateway for infecting a large number of victims quickly, and allowing the malicious code to remain persistent even long after the victims have left the scene.

Other known techniques to establish a MITM position are ARP spoofing [5] and DHCP spoofing [7]. Additionally, DNS cache poisoning [9, 3] represents a powerful off-path attack since the attacker does not to share the same local network.

7 Related Work

Jackson et al. [12] identifies the problem of insufficiently isolated per-site caches, arguing that the browser does not apply the same-origin policy consistently when writing to and reading from the cache. In other words, when cacheable third-party content hosted by a server S is included in site A, the browser does not restrict the cached content to be used only by A, but instead allows another site B, that includes the same third-party content via S, to use the cached copy that has been previously cached in the context of A.

To prevent cache content content from being used across domains, the authors propose partitioning the cache based on the embedding context. That is, the third-party content in the above example hosted by S should be cached for A and B separately such that the pairs (A, S) and (B, S) represent two disjoint caches. The authors implement this policy as an extension for the Firefox browser called *SafeCache* [19]. Unfortunately, our attempts to test the extension failed due to incompatibility with the most recent version 3.6 of Firefox.

Cache partitioning would solve the problem of cross-domain cache access, yet no browser that we are aware of currently implements it out of the box. We strongly advocate that browser vendors implement cache partitioning on a per-site basis in order to limit the impact of the attacks presented in this paper.

8 Future Work

Our attack tool presented in §4 currently does not spoof strong validators in the `Etag` header, but instead always returns a `Cache-Control`, `Expires`, and `Last-Modified` header. Strong validator spoofing requires the attacker to copy the validator from the original into the forged response. We will implement this feature in future work.

In the second part of this project, we also plan to perform an extensive trace-based study to understand the severity of this threat by analyzing the cache expiration and validation behavior observable in practice. In particular, we plan to study the conditions that trigger cache validation in the browser, which is an important aspect because the attacker can forge the cache control headers with extremely long expiration times in order to suppress validation through conditional requests. We already identified that validation occurs at the startup of Safari, and when hitting the reload button. In order to assess the severity of this threat, it is crucial to have a clear understanding of the validation model.

Another future avenue worth investigating is the impact of this attack on mobile devices. An intriguing possibility is to target smartphones, since these devices put significant effort into joining wireless networks to achieve higher bandwidth. The question remains as to how aggressively the web browsers in these devices should cache data. If smartphones operate like the disk cache of regular web browsers, then it is straightforward to exploit this behavior, e.g., with the probe stealing attack sketched in §6. A unique aspect of an infected population of mobile devices is the ability to bridge the gap to attack the cellular network. For instance, a trivially distributed denial-of-service could be to used to instruct the victim to download a large file, which would be difficult for the provider to filter.

Our initial experiments show that such mobile devices do not actually maintain a persistent cache. In particular, the current version of Safari on the iPhone and iPod touch (iPhone OS version 3) does not maintain a cache between sessions, so if another application is selected, the web cache is cleared (for any not-currently active

pages).[7] We have not yet investigated other mobile devices, however, we do not expect this to be the case for long: HTTP caches offer huge performance increases and the lack of caching on these devices seems an unusual oversight.

9 Conclusion

The browser's document cache exists primarily for performance reasons; clients can avoid unnecessary requests and increase their responsiveness, while servers can reduce their used bandwidth and I/O load. HTTP supports caching through expiration and validation mechanisms. Unfortunately, the server is the only entity that performs integrity checks of the resource, opening the opportunity for an attacker to replace cache content without causing later validation checks to fail. This is exacerbated by the lack of cross-domain cache isolation, which allows the browser to use the same cached resource across different origin contexts.

We exploit these security issues by constructing an attack that poisons the client's browser cache with a malicious version of the originally requested resource. In particular, we implement `air-poison`, a proof-of-concept tool that sniffs the air for HTTP JavaScript requests and prepends a cacheable malicious payload to the victim. We will release `air-poison` to the public after having completely implemented the attacks described. Google Analytics presents a particularly vulnerable target for this attack, due to its widespread use and aggressive caching behavior.

To quantify the severity of this threat, we plan to conduct a trace-based evaluation of the observable caching behavior in practice. We believe that the presented attack poses a clear threat for mobile devices, and are particularly interested in developing a better understanding of the caching behavior of such devices.

References

[1] Airbase-ng. `http://www.aircrack-ng.org/doku.php?id=airbase-ng`.

[2] Google Analytics. `http://www.google.com/analytics`.

[3] Michael D. Bauer. Securing DNS and BIND. *Linux Journal*, page 2, 2000.

[4] BeEF – Browser Exploitation Framework. `http://bindshell.net/tools/beef`.

[5] Thomas Demuth and Achim Leitner. Arp spoofing and poisoning: traffic tricks. *Linux Magazine*, (56):26–31, July 2005.

[6] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the Second-Generation Onion Router. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[7] R. Droms and W. Arbaugh. Authentication for DHCP Messages. RFC3118, 2001.

[8] Darin Fisher. HTTP Caching in Mozilla. `http://www.mozilla.org/projects/netlib/http/http-caching-faq.html`, October 2002.

[9] Steve Friedl. An Illustrated Guide to the Kaminsky DNS Vulnerability. `http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html`, August 2008.

[10] The Biggest Google Analytics Sites. `http://www.backendbattles.com/backend/Google_Analytics`.

[11] HTTP/1.1: Caching in HTTP. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html`.

[12] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting Browser State from Web Privacy Attacks. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 737–744, 2006.

[13] Mike Kershaw. Wifi Security –or– Descending Into Depression and Drink. BlackHat DC, Arlington, VA, USA, 2010.

[14] LORCON (Loss Of Radio CONnectivity). `http://802.11ninja.net/lorcon`.

[15] mac80211. `http://linuxwireless.org/en/developers/Documentation/mac80211`.

[16] Metasploit penetration testing framework. `http://www.metasploit.com`.

[17] Pyrit – WPA/WPA2-PSK and a world of affordable many-core platforms. `http://code.google.com/p/pyrit`.

[7] The only caching that Safari seems to support is HTML5 application caching, which might still be attacked in this manner.

[18] Vivek Ramachandran and MD Sohail Ahmad. Caff Latte with a Free Topping of Cracked WEP – Retrieving WEP Keys From Road-Warriors. TOOR-CON9, San Diego, USA, October 2007.

[19] Stanford SafeCache. `http://www.safecache.com`.

[20] Roi Saltzman and Adi Sharabani. Active Man in the Middle Attacks. Technical report, IBM Rational Application Security Group, February 2009.