Quantifying Persistent Browser Cache Poisoning

Matthias Vallentin
mavam@cs.berkeley.edu

Yahel Ben-David
yahel@cs.berkeley.edu

Abstract

Web browsers rely on caching for improving performance and for reducing bandwidth use. Cache poisoning poses alarming security concerns in light of HTTP's lack of an integrity guarantee in conjunction with the properties of its caching behavior.

In our previous study we demonstrated the simplicity of replacing objects in the browser cache with malicious code to enable a persistent attack. This paper expands on this topic with a quantitative analysis of the impact of this threat. Based on full-packet traces from two distinct environments – a large research lab in California and a network in rural northern India – we conduct an empirical study showing that *(i)* an attacker can with high probability achieve a long-lived attack vector when poisoning a web object picked at random and, *(ii)* that the high degree of object sharing, especially of executable code, enables an attacker to achieve high-coverage attack vectors by only poisoning a small set of intensively shared objects.

We believe that the increasing popularity of Web 2.0 mash-ups will increase the degree of sharing, making the discussed attack extremely wide in scope. In particular, we note the conceptual security shortcomings and risks of JavaScript content distribution networks (CDNs), techniques used by cellular carriers for compressing content on-the-fly, advertising networks, and popular services to track users' surfing behavior.

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 ongoing trend towards faster, richer, and more sophisticated web applications makes caching an indispensable component of fulfilling the user's expectations for responsiveness. Unfortunately, HTTP protocol was not designed with security considerations in mind, opening it up to a variety of trivial attack vectors in the presence of a network attacker.

In particular, HTTP does not provide integrity guarantees, allowing an attacker to modify the requested objects by the victim and to add malicious, executable code, mostly in the form of JavaScript. Although this is a well-known threat, little attention has been paid to the consequences when the attacker deliberately alters the object's caching properties in order to *persistently* plant malicious code in the victim's browser cache.

Consider the scenario where a victim surfs the web in a coffeeshop while being exposed to an attacker who not only injects malicious code in-flight, but also changes the caching properties of the objects requested by a victim; although the attack vector only exists while victim and attacker share the same network, the malicious code remains in the victim's cache, potentially for weeks or months, ready to execute each time the victim's browser makes use the poisoned object. We demonstrate that it is sufficient to poison only a few high-profile targets to render this attack a severe threat, even when the victim is only exposed for a short time period to the attacker.

The remainder of the paper is structured as follows: after familiarizing the reader with the relevant HTTP caching basics in §2, we present our evaluation in §3. We then discuss the impact of this attack in §4. We give a brief overview of mitigation approaches in §5 and summarize related work in §6. Finally, we give concluding remarks in §7.

2 HTTP Caching

Caching in HTTP [13] is based on two principal mechanisms, *expiration* and *validation*.

In order to allow clients to verify the freshness of an object, the server specifies its expiration, using either the `Expires` header or `max-age` directive in the `Cache-Control` header. A fresh object does not need

Environment	Start	End	Length	Connections	Size	HTTP
LBNL	4/21/2010 12:05pm	4/23/2010 12:34pm	48 hours	22,200,000	1,178.5 GB	95.9 %
AirJaldi	3/9/2010 4:27pm	3/11/2010 8:47am	40 hours	837,900	81.6 GB	99.5 %

Table 1: Summary of the two packet traces used in this paper.

Browser	Reload	Forced Reload
Firefox 3.6.3 (Win/Mac)	$\mathcal{C}$	Ctrl/⌘+ F5
Internet Explorer 8	$\mathcal{C}$	Ctrl + F5
Safari 4.0.5 (Mac)	$\mathcal{C}$	Shift + Reload
Chrome 5.0 beta (Mac)	$\mathcal{C}$	N/A[a]
Opera 10.10 (Mac)	$\mathcal{U}$ [b]	F5
Opera 10.53 (Mac)	$\mathcal{C}$	N/A

[a] On Windows, Ctrl/Shift + F5 supposedly bypasses the cache [3].
[b] Only of the cached URL and not the entire DOM.

Table 2: Browser validation behavior in different scenarios. We denote a conditional request by $\mathcal{C}$ and an unconditional request by $\mathcal{U}$.

to be refetched, thereby avoiding unnecessary requests entirely. An expired object must be validated by asking the origin server (i.e., not an intermediate cache) if the local copy can still be used.

For this purpose, HTTP enables validation by means of *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. If the values do not match, the server returns the full response. HTTP 1.1 distinguishes between *strong* and *weak* validators. A strong validator necessarily changes when the entity changes, whereas a weak validator only changes when significant semantic changes occur. The `Last-Modified` header is implicitly weak, as it offers only per-second granularity. Strong validators are implemented by means of an *entity tag* (ETag), which is a custom value chosen by the server and placed in the `ETag` header.

3 Evaluation

In this section, we quantify the threat posed by browser cache poisoning. After presenting our datasets in §3.1, we look at how and when browsers validate cache entries. Thereafter, we study the object cacheability in §3.3 from a general perspective. Not only does this enable us to gauge the relevant fraction of vulnerable objects, but it also allows us to quantify the persistence aspect. Then, in §3.4, we identify high-profile targets that are particularly attractive for cache poisoning.

3.1 Data

Our dataset consists of two full packet traces from contrasting environments: a large research institute in California, USA, and a rural community in the Indian Himalayas. The high-order details of our data are summarized in Table 1.

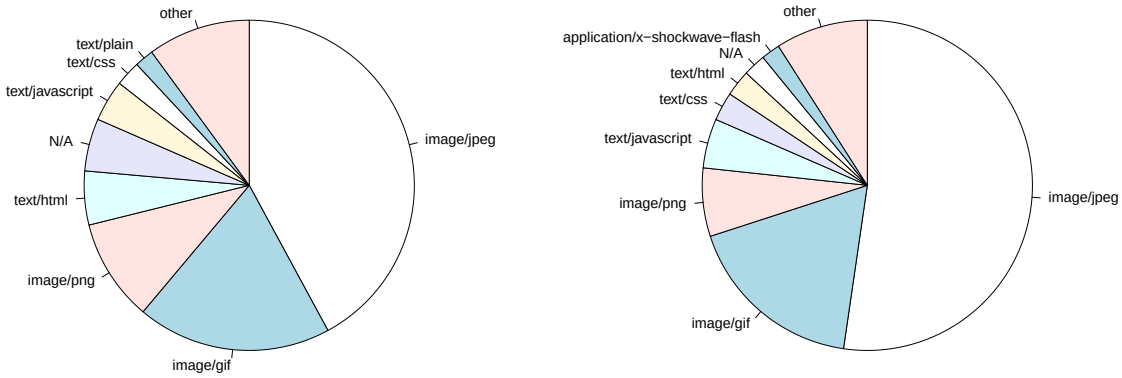
The first environment is the Lawrence Berkeley National Laboratory (LBNL, [16]), which is the oldest of the U.S. Department of Energy's national laboratories and managed by the University of California, Berkeley. Its approximately 4,000 users and 13,000 hosts are connected to the Internet via a 10 Gbps uplink. We captured a 48-hour full packet trace from April 21 to April 23 with 22.2 million connections (95.9 % HTTP).

The second environment is the AirJaldi [1] rural wireless network serving mostly the Tibetan community-in-exile. The network caters to about 10,000 users and has an uplink totaling 10 Mbps. The AirJaldi network currently uses several layers of NAT devices, limiting the visibility into the network at the granularity of a single host, since our vantage point is at the central gateway to the Internet. The trace we captured spans 40 hours from March 9 to March 11 with 837.9 thousand connections (99.5 % HTTP).

Since the attacker is only interested in manipulating executable object types that the browser interprets as code, we restrict our following discussion to HTML and JavaScript. Although Flash is another candidate carrying executable code, it takes more effort for the attacker to inconspicuously replace Flash content, which is why we address it only briefly.

In both environments, images account for the largest share of cacheable objects, as depicted by Figure 1; cacheable HTML and JavaScript only account for 2.57 / 5.27 % and 4.83 / 4.06 % respectively for LBNL/AirJaldi. We continue to use this notation throughout the paper where the first number is from LBNL data and the second from AirJaldi. The displayed shares are based on unique URL counts with stripped parameters (such as `?q=foo&k=v`) to avoid a sampling bias in favor of frequently visited sites. Unless otherwise noted, we apply this sanitization step in all remaining analyses.

To further winnow down the set of candidate targets, we examine the validator type. In principle, both strong and weak validators represent potential targets, but we restrict our analysis on weak validators, because they are



(a) LBNL.

(b) AirJaldi.

Figure 1: Breakdown of cacheable objects by MIME type.

easier to analyze with a short measurement time window. This is due to the fact that a response with a weak validator contains an exact last modification date, whereas a strong validator only includes an object hash. As shown in Figure 2, the majority of HTML traffic is not cacheable in both LBNL and AirJaldi. We hypothesize that this is due to the prevalence of dynamic web applications that generate most of the content on-the-fly. The use of weak validators dominate for JavaScript. For comparison, we also included Flash objects, which, not surprisingly, cache well due to their larger size.

3.2 Validation Behavior

Recall that the cache poisoning attack will only remain persistent when the browser *(i)* either does not validate the cache entry at all, or *(ii)* when a validation of a conditional request succeeds. Since the attacker sets an expiration time to ∞, only an unsuccessful validation or an unconditional request can overwrite a malicious cache entry with a clean one. Hence, we now investigate the client-side browser validation behavior.

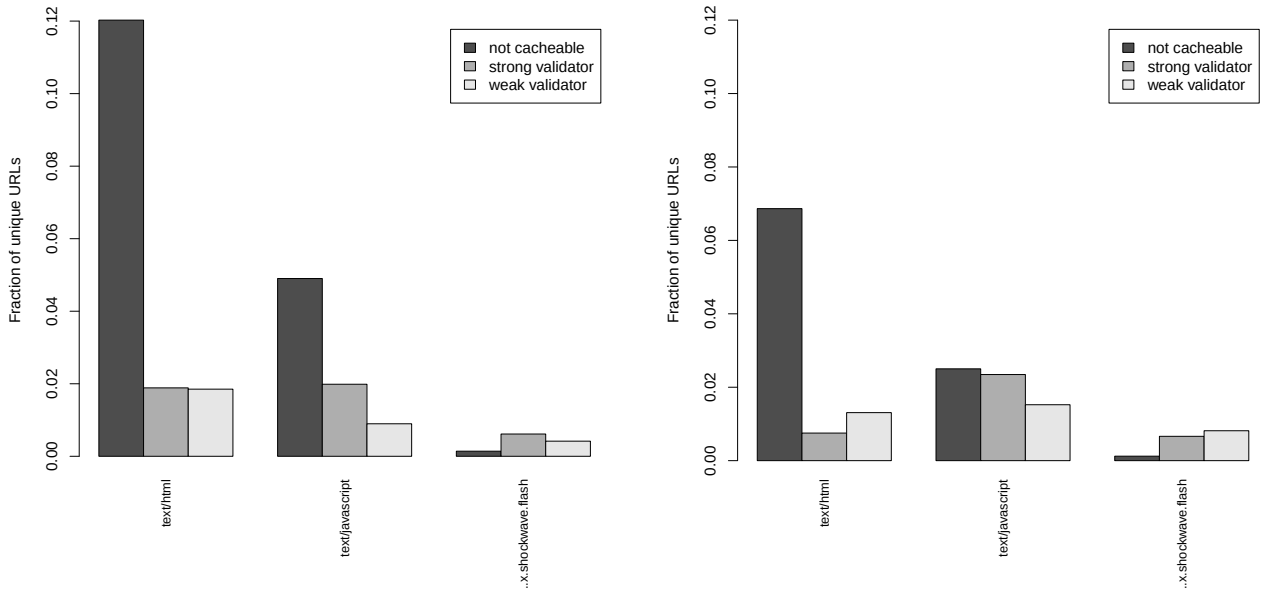
The browsers and conditions that we tested are summarized in Table 2. As expected, we also find all browsers suppress requests for a fresh cache entry and issue a conditional request when an entry becomes stale. All tested browsers issue a conditional request on reload, except for Opera 10.10 which adds a `Cache-Control: no-cache` to the request, rendering it unconditional. Opera's reload behavior also differs when hitting the reload button compared to pressing F5; the former only triggers an unconditional request for the cached URL, whereas the latter validates the entire DOM tree. Even different browser versions differ. Opera 10.53 on the Mac does not support forced reload by either F5 or Shift + click Reload.

Overall, we find that current browsers do a conditional request on reload, browser restart, and wake-up from standby, and do an unconditional request only if the user explicitly requests it via a forced reload, assuming that the browser supports it in the first place.

3.3 Cacheability

Since a network attacker has control over the object's expiration, we assume it to be ∞. That is, the cache entry never expires and will always stay fresh in victim's cache. We further assume, based on the previous section, that the victim does not initiate a forced reload, but will only send conditional request for validation. Now, only an unsuccessful validation overwrites the malicious cache entry, which turns the problem of quantifying the attack persistence into a function of the object's *lifetime*, which we define as the mean difference between two consecutive modification times. Note that unlike the expiration, the lifetime is a random quantity.

Our measurement window is very small compared to the values the lifetime can take, and we can only observe



(a) LBNL.

(b) AirJaldi.

Figure 2: Validator breakdown measured as the fraction of unique URLs (with stripped parameters) for a given MIME type. The bars for each type sum up to the total fraction of unique URLs of that type.

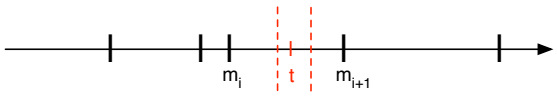


Figure 3: Lifetime estimation for one object. Let m_i be the modification time obtained from the `Last-Modified` header and t be t the sampling time in our measurement window, then the observed *truncated lifetime* sample of the object is $l_i^- = t - m_i$, whereas the actual lifetime sample is $l_i = m_{i+1} - m_i > l_i^-$.

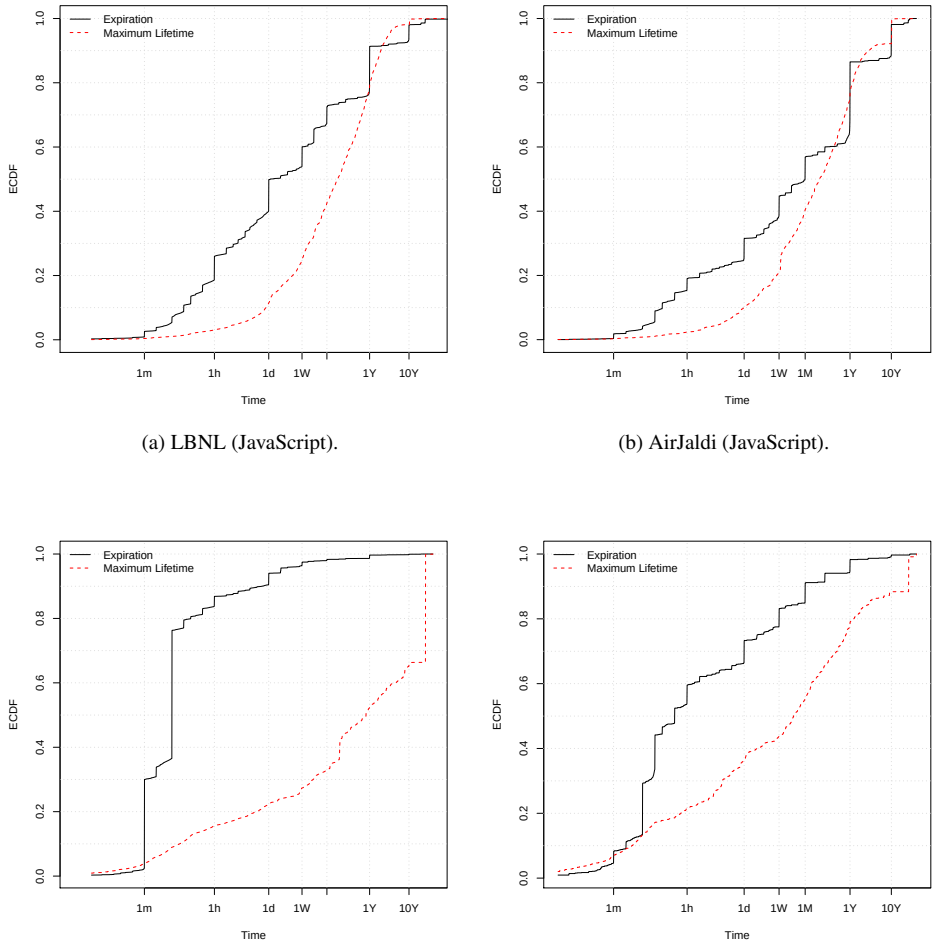
a single sample of the *truncated lifetime*, as illustrated in Figure 3. Let m_i be the time in the `Last-Modified` header and t be the measurement time, then the truncated lifetime sample i is $l_i^- = t - m_i$, whereas the actual lifetime sample is $l_i = m_{i+1} - m_i$, with $l_i^- < l_i$. If we observe one or more modifications of an object in our measurement window, we take the maximum over all lifetime samples including the truncated lifetime. We use l_i^- as an estimate for l_i. We plan to give a more rigorous definition and estimation of lifetime based on homogeneous Poisson processes in the future, as sketched in Appendix B.

Unlike the lifetime, the expiration of an object is constant, which makes it independent of the measurement window size and hence an attractive candidate as a lower bound on the lifetime. The relationship between expiration and lifetime is illustrated in Figure 4, which shows the empirical cumulative distribution functions (ECDFs) for HTML and JavaScript. The median expiration of JavaScript is 1 / 30 days compared to a median lifetime of 57 / 67 days (Figure 4a and 4a). We observe a larger discrepancy for HTML: the median expiration is 5 / 30 minutes, whereas the median lifetime is 295 / 16 days (Figure 4c and 4d).

Consider an attacker who randomly picks one object from the victim's surfing session, and wants to perform a cache poisoning attack that survives at least one week. That is,

$$\mathbb{P}\left[X_T > 1W\right] = 1 - \mathbb{P}\left[X_T \leq 1W\right] = 1 - \widehat{F}_n^T(1W)$$

where $X_T \sim \widehat{F}_n^T$ is a random variable denoting the lifetime of an object with MIME type T, and $\widehat{F}_n^T$ denoting the corresponding ECDF calculated from the l_i^- samples of n distinct objects. This probability equals to 0.75 / 0.79 for JavaScript and 0.73 / 0.56 for HTML. Therefore, we conclude that long-lived cache poisoning attacks are easy to conduct.



(a) LBNL (JavaScript).

(b) AirJaldi (JavaScript).

(c) LBNL (HTML).

(d) AirJaldi (HTML).

Figure 4: Relationship between expiration and lifetime. The x-axis shows time in seconds on a logarithmic scale.

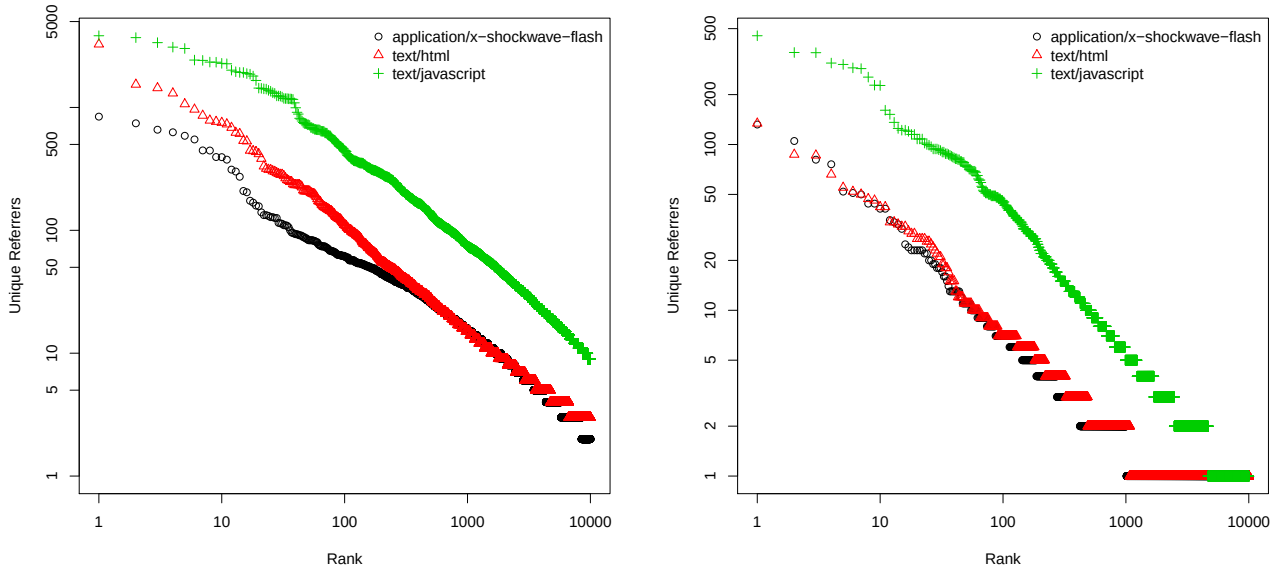
3.4 Shared Cache Access

The attacker's code can only run while the victim stays on a page which includes the code. When the victim navigates away, the attacker has lost the opportunity to interact with the victim. Therefore, poisoning the cache is particularly effective for objects shared across multiple sites because it increases the frequency that the malicious code executes.

Let a browsing session be a set of URLs that the victim surfs to in a given time window. Then, define the attack *coverage* to be the fraction of pages that execute the maliciously cached code divided by the total number of pages visited for a given session. A coverage of 1 means the attacker can execute code on every page, whereas 0 coverage means the attacker's code does not execute at all.

To identify the most attractive targets that will maximize the attack coverage, we conduct the following experiment. Our key insight is that largely shared objects exhibit a high number of unique referrers. In Figure 5, we plot the number of unique referrers of an object against its rank, which is its index when sorting all objects by their number of referrers in decreasing order.

The object distribution is Zipf-like, which is aligned with our intuition that only a few popular URLs are highly shared. Amongst the MIME types that can carry executable code, JavaScript is shared the most. We list the top 10 JavaScript objects in Table 3. The majority of these objects stem from advertisement networks and tracking services (see §4) and 30 / 70 % of the objects belong to Google. Due to the increasing popularity of Web 2.0 mash-ups, we believe that the amount of sharing will increase in the future, which facilitates achieving a high



(a) LBNL.

(b) AirJaldi.

Figure 5: The number of unique referrers of per unique cacheable object is proportional to its rank. Note the logarithmic scale on both axes. The Zipf-like object distribution demonstrates that it is quite effective to poison a few high-profile targets in order to substantially increase the coverage of the attack.

attack coverage. Note that the referrer numbers serve only as a lower bound, because they have been calculated from cacheable objects only. Our next experiment seeks to increase the precision of this bound.

To this end, we pick one specific object that is easier to analyze: `ga.js` from Google Analytics, whose number of unique referrers is 2266 / 358. It is easier to analyze because each visit on a page with Google Analytics triggers a request for `www.google-analytics.com/__utm.gif`, a 1x1 pixel image which includes detailed user statistics in its request parameters. Since the number of unique referrers for `__utm.gif` represents an exact estimate, we can use it as a proxy for `ga.js` which allows us to understand the impact of caching on absorbing visible requests.[1]. The number of unique referrers in `__utm.gif` requests amounts to 64,995 / 3,789. This is a factor 28.7 / 10.6 difference. In fact, the referrers in `__utm.gif` requests represents 1.0 / 0.9 % of *all* requested URLs, or 14.0 / 7.1 % of all hosts.

Since it is possible to achieve a coverage of 14 % just by poisoning Google Analytics, we conclude that it is enough to poison a few heavily shared objects in order to achieve a substantial coverage.

[1]Note that we can only use this example to compare cacheable objects with similar expiration times.

4 Impact

In the previous section, we have shown that *(i)* object lifetimes are sufficiently long for persistent cache poisoning attacks and that *(ii)* it is enough to target a few intensively shared objects to gain a high attack coverage. In this section, we highlight the practical implications of these findings with examples involving tracking and advertisement services (§4.1), JavaScript CDNs (§4.2), and cellular carriers (§4.3).

4.1 Tracking and Advertisement

Tracking services enable a web site operator to extract a plethora of statistics from a user in order to understanding of user's surfing behavior and optimize the site according to it. The gathered information includes the time a user stayed on a page, the depth of the navigation, and the site to which the user went afterwards. With the availability of numerous tracking provides, operators do not have to implement their own service, it suffices to include a third-party script on the pages to analyze. Since the amount of sharing is directly proportional to the popularity of a tracking service, an attacker merely needs to pick the most popular tracker to achieve high attack coverage.

# Unique Referrers	URL
3833	b.scorecardresearch.com/beacon.js
3700	rmd.atdmt.com/tl/DocumentDotWrite.js
3368	edge.quantserve.com/quant.js
3096	pagead2.googlesyndication.com/pagead/show_ads.js
3019	s0.2mdn.net/879366/flashwrite_1_2.js
2430	googleads.g.doubleclick.net/pagead/test_domain.js
2415	static.ak.connect.facebook.com/connect.php/en_US/js/Api/CanvasUtil/Connect/XFBML
2339	pagead2.googlesyndication.com/pagead/js/graphics.js
2334	static.ak.fbcdn.net/rsrc.php/z49PH/hash/9p47jvzp.js
2284	upload.wikimedia.org/centralnotice/wikipedia/en/centralnotice.js?270z54
2266	www.google-analytics.com/ga.js
453	pagead2.googlesyndication.com/pagead/show_ads.js
359	pagead2.googlesyndication.com/pagead/sma8.js
358	www.google-analytics.com/ga.js
310	pagead2.googlesyndication.com/pagead/js/graphics.js
304	pagead2.googlesyndication.com/pagead/expansion_embed.js
290	googleads.g.doubleclick.net/pagead/test_domain.js
287	pagead2.googlesyndication.com/pagead/js/abg.js
255	edge.quantserve.com/quant.js
228	rmd.atdmt.com/tl/DocumentDotWrite.js
227	s0.2mdn.net/879366/flashwrite_1_2.js
42	adserver.itsfogo.com/default.aspx?t=f&v=1&zoneid=40761

Table 3: The top 10 cacheable JavaScript objects sorted by their unique referrer count. The upper half of the table corresponds to LBNL and the lower half to AirJaldi.

According to BuiltWith [2], Google Analytics has the largest market share in the space of tracking services (56.41 %), followed by Omniture SiteCatalyst [4] (14.08 %), and Quantcast Tracking [5] (12.21 %), as of May 2010. Others report Google Analytics to be used by approximately 32.2% of the Alexa's list of the 10,000 most popular web sites [12], and we find that Google Analytics is used on 14 % of all visited sites (see §3.4).

Ad networks and market research firms use similar techniques to tailor their advertisements to the user. We find that the majority of shared JavaScript stems from ad networks, as shown in Table 3.

4.2 JavaScript CDNs

JavaScript content distribution networks (CDNs) provide a resilient hosting infrastructure for popular, open source JavaScript libraries used by a large number of sites. This service is attractive to content providers who make use of JavaScript to yield improved performance and availability, while reducing their own bandwidth at the same time.

For example, the Google AJAX libraries [7] use a single shared loader script that enables access to 10 such libraries. An attack that poisons the loader automatically gains cross-site scripting (XSS) capabilities on each site that uses *any* of the provided libraries by the Google CDN. [2] Alternatively, the attacker could inject links to all 10 JavaScript libraries hosted on the CDN to trigger a request from the victim and then reply with a modified version. This is a very lucrative vector since all libraries have an expiration of one year.

[2] Although the loader itself not cacheable, the attacker will add the necessary headers to keep it in the cache until the next validation occurs.

4.3 Cellular carriers

With the argument to save bandwidth and client-side resources, some carriers transparently compress requested content or reduce the quality of images when accessing the Internet via their cellular data networks. For example, we tested the 3G network of T-Mobile and find that image and video links are rewritten to be fetched through a proxy. Some carriers go even further and inject a piece of JavaScript on *every* page that a user visits [18, 6, 8, 21]. AT&T for instance injects the script `http://2.2.3.4/bmi-int-js/bmi.js`, which installs custom keyboard shortcuts to allow for image quality adjustment and changes page element titles. Not only does this break pages that use these keyboard shortcuts, but this particular script presents a perfect target for a persistent cache poisoning attack. An attacker who injects a malicious version of the script can execute code on every accessed page while the victim uses the cellular data network.

Note that the attacker does not have to be on the cellular 3G network to attack the user. In the coffeshop scenario, it suffices to inject the link to `bmi.js` on an arbitrary website and then respond to the provoked request. If the victim switches to a 3G network in some point in the future, the infection already took place and

the attacker's code executes on each page.

5 Countermeasures

The presented attack exploits the lack of integrity in the HTTP protocol coupled with the relaxed cache validation practices. Although the use of HTTPS solves the problem by eradicating the man-in-the-middle (MITM) attack vector, the site provider has to offer this option in the first place. Nonetheless, there exist mitigations for this attack which we outline below.

5.1 Disabling the Cache

The simplest solution is to disable the browser cache entirely. But since intermediate proxies may still have a poisoned entry the browser, the user must additionally make sure to bypass them and always request the object from the origin server. Each request must then contain a `Cache-Control` or `Pragma` header set to `no-cache`.

Although disabling the browser cache is the safest solution, it prevents from accessing any of the benefits of caching. The penalty is particularly severe when *(i)* performance and responsiveness are important, *(ii)* accessing large objects such as videos, *(iii)* bandwidth is a scarce or expensive resource, or *(iv)* only intermittent connectivity is available.

5.2 Frequent Cache Clearing

To minimize the time that malicious content stays in the browser cache, another strategy is to frequently clear the cache. One practical model that balances the benefits of caching with the potential for maliciously cached objects would be to automatically purge the cache when the external IP address of the machine changes. The rationale behind this strategy is that a present network attacker can always inject content, so the client is hosed anyway for the time sharing the network with the attacker. However, when the client later joins a different network, clearing the cache would get rid off all the accumulated badness.

One minor problem is that, due to the prevalence of NATs and private address spaces, a client which moves networks might not necessarily change IP addresses. Thus a client would benefit from using an external service which reported its externally visible IP to determine when its address changes.

5.3 Improving Validation

Although the attacker controls the object expiration, scenarios still exist that trigger premature validation, offering an opportunity for disinfection by retrieving the current object from the server (see §3.2). But since the validator is opaque (i.e., the client does not recompute it locally), the attack persistence is a function of the object lifetime. Instead, it already would be an advantage if we could ensure that the validation fails for corrupted objects. HTTP has a potential mechanism which would enable client-side validation, but it requires server-side changes. The HTTP header `ETag` is defined as a unique opaque identifier which the server can use to validate whether the client has up-to-date content. Unfortunately, server implementers have historically chosen very poor validators. For example, Apache uses file size, MIME type, and inode number to create the ETag. Neither the client nor the servers in a cluster can recreate the ETag.

The proper solution is for validators to be a cryptographic hash of the file contents, as validators must be cryptographically secure and calculateble on both the client and server. A simple solution would be to make the ETag non-opaque if it begins with the string "SHA-256:," which would allow the client to properly determine that this ETag represents a cryptographic hash, and that it should validate that the hash in the ETag matches the received contents before placing it in the cache.

Although there is no cryptographic protection for the downloaded object, the hash does protect the cache on subsequent validations. If the attacker does not create a proper hash ETag, the object will not be cached. If the attacker forges an ETag, the content will be fixed when the victim's browser validates the contents.

An additional advantage of shifting to hash-based validators is that it makes CDN and cluster deployments work properly.

5.4 Cache Partitioning

Another solution is to implement the browser cache on a per-site granularity. Jackson et al. [14] frame the existence of a shared browser cache as lack of per-site cache isolation, with the argument that the browser does not apply the same-origin policy consistently when writing to and reading from the cache. For example, 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.[3]

Although cache partitioning reduces the attack coverage, it does not solve the underlying problem, and suffers from the same problems arising from disabling the cache (see §5.1).

6 Related Work

We are not the first to find that the lack of integrity in the HTTP protocol in combination with its caching behavior opens a dangerous attack vector. Similar ideas have been summarized mostly by the penetration testing community and independent activists [20, 15]. However, we are the first to frame this threat in an academic context and quantify its impact.

In previous work on HTTP caching, Mogul et al. developed a scheme to detect duplicate message bodies [17]. Barford et al. studied the effect of request distribution patterns and their caching implications [9], and Doyle et al. explored the effect of ubiquitous caching on request patterns [11]. Our research differs in that we particularly focus on the security implications, rather than studying the caching phenomenon itself.

The occurrence of Zipf-like distributions in web traffic is well-known [10, 22, 11]. Our research is novel in its use of the number of unique referrers as a proxy for the popularity and degree of sharing of an object.

7 Conclusion

The browser's object 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. In the current HTTP protocol, the server is the only entity performing integrity checks, opening the opportunity for an attacker to change the object's content and expiration without causing later validation checks to fail.

While we demonstrated the ease at which this can be conducted in our previous project, this project focuses on the evaluation of the magnitude of this threat.

With data from two very distinct environments, we estimate object lifetimes and find that the median lifetime varies between two months and one year, enabling long-lived persistent attacks. This is exacerbated by the high degree of sharing that objects exhibit, which invites to poison a few high-profile target to achieve high attack coverage.

Current mitigation techniques are insufficient. The best advice we can give is to use an encrypted HTTPS session when possible. We hope this analysis raises awareness for this latent threat and stimulates future research in this area.

[3] 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 of Firefox.

Acknowledgements

We would like to thank Devdatta Akhawe for the inspiring discussions about potential attack targets and Robin Sommer for his invaluable feedback on the evaluation. Our gratitude extends to Nicholas Weaver for his comments on the countermeasures and Rohan Gupta for his thoughts on the browser validation behavior.

References

[1] AirJaldi. `http://www.airjaldi.org`.

[2] BuiltWith Technology Usage Statistics. `http://trends.builtwith.com`.

[3] Getting started: Keyboard and mouse shortcuts. `http://www.google.com/support/chrome/bin/answer.py?answer=95743`.

[4] Omniture SiteCatalyst. `http://www.omniture.com/en/products/online_analytics/sitecatalyst`.

[5] quωntcast. `http://www.quantcast.com`.

[6] Bmi.js. `http://blog.zoiah.net/?p=14`, January 2009.

[7] Google ajax libraries api. `http://code.google.com/apis/ajaxlibs/`, May 2010.

[8] Jon Atkinson. http://1.2.3.8/bmi-int-js/bmi.js. `http://jonatkinson.co.uk/http1238bmi-int-jsbmijs`, 12 2007.

[9] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1-2):15–28, 1999.

[10] Lee Breslau, Pei Cue, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *In IEEE INFOCOM*, pages 126–134, 1999.

[11] Ronald P. Doyle, Jeffrey S. Chase, Syam Gadde, and Amin Vahdat. The Trickle-Down Effect: Web Caching and Server Request Distribution. *Computer Communications*, 25(4):345–356, 2002.

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

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

[14] 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.

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

[16] Lawrence Berkeley National Laboratory. http://www.lbl.gov.

[17] Jeffery C. Mogul, Yee Man Chan, and Terence Kelly. Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[18] RSnake. AT&T UTMS JS Injection. http://ha.ckers.org/blog/20100412/att-evdo-js-injection, April 2010.

[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.

[21] Howie Weiner. Tethering the iPhone : My Experience plus o2's secret little bmi.js file. http://www.badlydrawntoy.com/2009/08/07/tethering-the-iphone-my-experience-plus-o2s-secret-little-bmi-js-file, August 2009.

[22] Alec Wolman, M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative Web proxy caching. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 16–31, New York, NY, USA, 1999. ACM.

A Determining Cacheability

Determining whether an object is cacheable and extracting the expiration is not a straight-forward undertaking, because various headers have an effect on cacheability. The algorithm we use to gauge the expiration of an object is shown in Figure 6.

B Lifetime Estimation

Using the observed truncated lifetime as an estimate for the object lifetime is simple, but could be improved by a more rigorous mathematical model. In future work, we plan to recast this estimation problem in the framework of stochastic processes.

For a given object, consider a homogeneous Poisson process with rate λ and define W_i to be a random variables representing the i^{th} modification time and set $W_0 = 0$. Further, define $L_i = W_{i+1} - W_i$ to be the time between two consecutive modifications. Then, the lifetime of the object is given by the set

$$\left\{L_i : i \in \mathbb{N}_0^+\right\}$$

where

$$L_i \sim \text{Exp}\left(\frac{1}{\lambda}\right)$$

and $\widehat{\lambda} = 1/\bar{X}_n$ being the MLE for λ. There are some complications that make the application of this framework a little more complicated. First, with our short measurement window, we cannot observe more than one L_i sample per object. Second, we can only observe a truncated version of this lifetime sample, because our measurement window is placed randomly in the Poisson process.

```
# Determine whether a HTTP response is cacheable. The parameter date and expires
# must be in epoch seconds. Return the expiration time or -1 if the document is
# not cacheable.
function cacheable(cache_control, pragma, date, expires, last_modified, etag)
{
   if (cache_control ~ /no-cache/ || pragma ~ /no-cache/)
      return -1

   if (cache_control ~ /max-age/)
   {
      m = split(cache_control, directives, ",")
      for (i = 1; i <= m; i++)
      {
         if (directives[i] ~ /max-age/)
         {
            n = split(directives[i], kv, "=")
            maxage = int(kv[2])

            # The last character in the max-age value could be alphabetic.
            # Because this extension is not treated in the standard, we
            # avoid the ambiguous 'm' character entirely (and underestimate
            # the expiration).
            last = substr(kv[2], length(kv[2]), 1)
            if (last ~ /^[hdwy]$/)
            {
               if (last == "h")
                  maxage *= 3600
               else if (last == "d")
                  maxage *= 86400
               else if (last == "w")
                  maxage *= 604800
               else if (last == "y")
                  maxage *= 31536000
            }

            if (maxage >= 0)
               return maxage
            else
               break
         }
      }
   }

   # Gauge expiration time based on Date and Expires header.
   if (date > 0 && expires > 0)
   {
      delta = expires - date
      if (delta > 0)
         return delta
   }

   # At this point, a strong validator implies 0 expiration because the
   # document is checked each time. Unless it changes, the server returns a
   # 304. Further, according to 13.3.4 in the HTTP spec, if we have only a
   # Last-Modified header, the origin server "SHOULD use that value in
   # non-subrange cache-conditional requests (using If-Modified-Since)". This
   # means the document has zero expiration, but is cacheable in principle.
   if (etag != "" || last_modified > 0)
      return 0

   # Not cacheable.
   return -1
}
```

Figure 6: Determining the expiration time of an object implemented in Awk.