Request and Conquer: Exposing Cross-Origin Resource Size

Tom Van Goethem, Mathy Vanhoef, Frank Piessens, Wouter Joosen
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
`first.lastname@cs.kuleuven.be`

Abstract

Numerous initiatives are encouraging website owners to enable and enforce TLS encryption for the communication between the server and their users. Although this encryption, when configured properly, completely prevents adversaries from disclosing the content of the traffic, certain features are not concealed, most notably the size of messages. As modern-day web applications tend to provide users with a view that is tailored to the information they entrust these web services with, it is clear that knowing the size of specific resources, an adversary can easily uncover personal and sensitive information.

In this paper, we explore various techniques that can be employed to reveal the size of resources. As a result of this in-depth analysis, we discover several design flaws in the storage mechanisms of browsers, which allows an adversary to expose the exact size of any resource in mere seconds. Furthermore, we report on a novel size-exposing technique against Wi-Fi networks. We evaluate the severity of our attacks, and show their worrying consequences in multiple real-world attack scenarios. Furthermore, we propose an improved design for browser storage, and explore other viable solutions that can thwart size-exposing attacks.

1 Introduction

In 1996, Wagner and Schneier performed an analysis of the SSL 3.0 protocol [67]. In their research, the authors make the observation that although the content is encrypted, an observer can still obtain the size of the requested URL as well as the corresponding response size. The researchers further elaborate that because it is possible to make an inventory of all publicly available data on a website, knowing the size of requests and responses allows an attacker to determine which web page was visited. Although content is increasingly being served over secure SSL/TLS channels [55], the length of requests and responses remains visible to a man-in-the-middle attacker. Consequently, the attack that was described by Wagner and Schneier two decades ago remains universally applicable. However, due to the various transitions the web underwent, the consequences of uncovering the size of remote resources have shifted drastically.

With the advent of online social networks, the dynamic generation of web pages goes even further. When browsing, each user is now presented with a personalized version, tailored to their personal preferences and information they, or members of their online environment, (un)willingly shared with these online services. Consequently, the resource that is returned when a user requests a certain URL will often reflect the state of that user.

Two types of size-exposing attacks, namely traffic analysis and timing attacks, have been widely studied. In traffic analysis, an adversary passively observes the network traffic that is generated by the victim's browsing behavior. Based on the observed size, sequence, and timing of requests and responses, an attacker can learn which website was visited by the victim [13, 25, 68], or uncover which search queries the user entered [12, 40]. In contrast to traffic analysis, where the threat model is typically defined as a passive network observer, launching a web-based timing attack requires the adversary to trick the victim in making requests to certain endpoints, which is typically achieved by running JavaScript code in the victim's browser. The attacker then measures the time needed for the victim to download the specified resources, which, depending on the victim's network condition, allows him to approximate the resource size, and ultimately obtain information on the state of the user [19, 7, 14].

Motivated by the severe consequences on the online privacy of a vast amount of users, we present a systematic analysis of possible attack vectors that allow an adversary to uncover the size of a resource. As a result of this evaluation, we discover design flaws in various browser features that allow an adversary to uncover the exact size of any resource. Furthermore, we demonstrate that by intercepting and manipulating encrypted Wi-Fi traffic, an adversary can uncover the exact size of an HTTP response. By leveraging these techniques, we show that when an attacker can make the victim send requests to arbitrarily chosen endpoints, the potential consequences of traffic monitoring become significantly more severe. In contrast to prior attacks, where adversaries could typically only obtain a rough estimate of the resource size, or were unable to attribute network traffic to specific requests, our size-exposing attacks show that the capabilities of an adversary are worryingly extensive, as we exemplify by the means of several real-world attack scenarios. Finally, we explore the viability of several defense mechanisms, leading to an improved browser design and a variety of possibilities for websites to thwart size-exposing attacks.

Our main contributions are:

- We perform an in-depth analysis at the level of the browser, network and operating system, and explore techniques that can expose the size of resources either directly or through a side-channel attack.
- We introduce several new attack vectors that can be leveraged to uncover the exact response size of arbitrarily chosen endpoints.
- By the means of several attack scenarios on high-profile websites, we demonstrate that an adversary can reveal the unique identity of an unwitting visitor within mere seconds, and extract sensitive information that the user shared with a trusted website.
- We propose an improvement to the specification of the Storage API, and explore various existing solutions that can be used to mitigate all variations of size-exposing attacks.

The remainder of the paper is structured as follows: in Section 2 we provide a high-level overview of the technical aspects related to recently introduced browser features. In Section 3 we present an in-depth analysis on potential size-exposing techniques, and elaborate on how these can be used in various attack scenarios. In Section 4, we discuss how adversaries can leverage these techniques against a number of real-world services. Furthermore, in Section 5 we propose and explore methods that can thwart size-exposing attacks. Section 6 covers related work, and Section 7 concludes this paper.

2 Background

One of the most important security concepts of modern browsers, is the notion of Same-Origin Policy [64, 73]. Despite what its name may suggest, it is not strictly defined as a policy, but rather represented as certain principles that ensure websites are restricted in the way they can interact with resources from a different origin. Although it is possible to initiate a cross-origin request, the Same-Origin Policy prevents reading out the content of the associated response, which is obviously imperative in order to provide online security. Naturally, the content of resources is not the only part that should be shielded off from other origins; the size of a resource should also be considered sensitive, as evidenced by the several case studies presented in Section 4 and prior work [7, 20, 60]. As such, it comes as no surprise that the browser APIs that are responsible for making HTTP requests will only report the length of a response when the associated request was to the same origin.

The Fetch API, which is currently implemented by Google Chrome, Firefox and Opera, and is under development by other browser vendors [39, 69], introduces a set of new semantics that aim to unify the fetching process in browsers. In short, the `fetch()` method is given a `Request` object, and a second, optional parameter that specifies additional options for the request. For instance, when the `credentials` option is set to `"include"`, the user's cookies will be sent along with the request, even when it is cross-origin. The `fetch()` operation will return a `Promise` that yields a `Response` object as soon as the response has been fetched. In case the request was authenticated, cross-origin, and did not use the CORS mechanism, i.e., the `mode` was set to its default value `"no-cors"`, the `Response` will be marked as `"opaque"`, which will mask all information (status code, response headers, cache state, body and length) of the response to prevent cross-origin information leakage. In the following sections we will show how certain browser mechanisms can be abused to uncover the length of cross-origin responses.

Although the content of an opaque `Response` can not be accessed, it is possible to force the browser to cache that resource. The Cache API, which is part of the Service Worker API [66], can be used to place `Response` objects in the browser's cache. For security purposes, the cache that is accessed by the Cache API is completely isolated from the browser's HTTP cache, and is not shared across different origins. The cache is accessed by opening a `Cache` object, which can then be used to store `Response` objects with their associated `Requests`. Note that *any* response can be stored, regardless of the `Cache-Control` headers sent out by the web server. To prevent a malicious entity from completely filling up the user's hard disk, certain quota rules apply. The details of these rules will be explained in more detail in Section 3.4.

3 Size-exposing Techniques

As demonstrated by prior research, the size of a website's resources is often related to the state of the user at that website [7, 38, 60]. Consequently, knowing the size of these resources allows an adversary to (partially) uncover the state of the user, which often yields sensitive information. In order to detect the presence of size-exposing attack vectors, we performed an in-depth analysis on all operations in which resources are involved. In this section, we present the results of this analysis and discuss the various techniques that can be used to infer the size of cross-origin resources. Next to the size-exposing methods that were discovered in prior research, we also introduce various novel techniques and re-evaluate methods in the light of recent protocol evolutions.

Throughout this section, we consider different attacker models based on the evaluated resource operation. As a rule of thumb, for each resource operation we considered all the attacker models in which the adversary is able to make observations about the operation. For instance, when analysing the transfer of a resource over the network, multiple attacker models were taken into account: an eavesdropper might inspect the encrypted network traffic directly, or Wi-Fi packets could be examined when the adversary is in physical proximity of the victim, or the attacker might simply use JavaScript to measure the time it took to complete the request.

Our evaluation mainly focuses on attacks in which the adversary infers sensitive information from the size of the resources that are returned to the victim when requesting specific endpoints. As such, we evaluate potential attack techniques under the assumption that an adversary can trigger the victim's browser to send authenticated requests to arbitrarily chosen endpoints. This can be easily achieved by a moderately motivated attacker due to the plethora of methods that can be used to execute arbitarary JavaScript code in a cross-origin context (with regard to the target endpoint). For instance, an attacker can trick the user in visiting his website using phishing via e-mail or social networks [29], register a typosquatting domain [41], launch an advertising campaign where JavaScript code or an iframe containing the attacker's web page is included [54], register a stale domain from which a JavaScript file is included [43], redirect insecure HTTP requests [37], ... Note that recent attacks on TLS also assume an attacker can execute JavaScript code in the victim's browser [16, 1, 62].

3.1 Operations Involving Resources

By looking at their typical "lifetime", we identified six different operations that involve resources, as shown in Figure 1. In the first step, a resource is generated at the

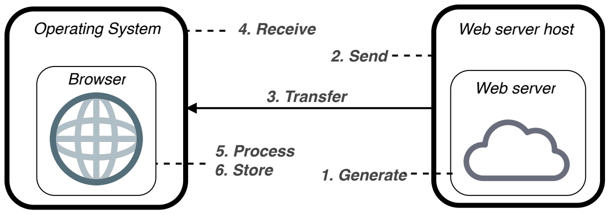

Figure 1: Overview of operations that involve resources.

side of the web server, as a result of the request initiated by the browser. Here the web server will associate the user's state with the included cookie, produce the requested content, and pour it into an HTML structure. Although several attacks have been presented that can extract sensitive information from this generation process, e.g., direct timing attacks [7], our research focuses on methods that can expose the size of the generated content. Since the length of the response is only known *after* it has been generated, attacks against the resource generation process are excluded from our evaluation.

Once a resource has been dynamically generated, the machine where the web server is hosted on will send it back to the user that requested it. This means that if an adversary is able to observe the amount of traffic generated by the web server, he could use this information to infer the size of the response. We discuss size-exposing techniques in this context in more detail in Section 3.2.

When the resource leaves the web server, it is sent over several networks before it reaches the client. In any of these networks, an adversary capable of intercepting or passively observing the network traffic could be present. Because size-exposing attacks can be considered to be superfluous when an adversary can inspect the contents of a resource, we only consider encrypted traffic in our evaluation. Prior work has shown that popular encryption schemes such as SSL and TLS do not conceal the length of the original HTTP request and response, leading to various attacks [67, 12]. In our analysis, we extend this existing work by re-evaluating the feasibility of size-exposing attack methods when the new HTTP version (HTTP/2) is used. Furthermore, we explore possible size-exposing attack techniques in the context of Wi-Fi networks, where another layer of encryption is added, and elaborate on our findings in Section 3.3.

As soon as the response reaches the client's machine, it is first received by the network interface, and then sent to the browser, where it is processed and possibly cached. Similar to the server-side, an adversary with a foothold in the operating system, can leverage traffic statistics to uncover the resource's length. In Section 3.2, we investigate these types of attack techniques under various threat models, both for mobile devices as well as desktops.

Table 1: An overview of size-exposing attack techniques with their associated resource operations (as per Figure 1) and whether the techniques can be used to obtain the exact size of a resource.

Size-exposing technique	Resource operation	Exact size	References
Cache timing attacks	2, 4		[48, 76, 72, 44]
Traffic statistics pseudo-files	2, 4	✗	[77], Section 3.2
SSL/TLS traffic analysis	3	✗	[67], Section 3.3
Wi-Fi traffic analysis	3	✗	Section 3.3
Cross-site timing attacks	3		[7, 20]
Browser-based timing attacks	5		[60]
Storage side-channel leaks	6	✗	Section 3.4

After receiving the response, the browser will first signal the completion of the request by firing an `Event`. In the threat model we consider, the request is initiated by the malicious JavaScript code, and thus, its completion is signaled to the attacker. It is known that the time it takes for a request to complete is correlated with its size, giving rise to so-called timing attacks. However, these attacks have several limitations, and can only be used to obtain a rough estimate of a resource's size. While a rough estimate is sufficient to perform certain attacks [7, 20], most of the real-world attacks we present in Section 4 require knowing the exact size of resources.

In a recent study, Van Goethem et al. found that the next step of a resource's lifetime, i.e., parsing by the browser, is susceptible to timing attacks as well [60]. In contrast to classic timing attacks, these browser-based attacks do not suffer from network irregularities, and thus provide attackers with a more accurate and reliable estimate. Nevertheless, the maximum accuracy that can be achieved with these methods is still in the range of a few kilobytes, which is insufficient for some of the novel attacks presented in Section 4.

Finally, browsers may store resources in the cache, allowing them to be retrieved much faster in future visits. Motivated by the potentially nefarious consequences of caching resources chosen by an adversary, we analyzed the specification of the various APIs that are involved in this process. Surprisingly, we found multiple design flaws that allow an adversary to uncover the *exact* size of any resource. In Section 3.4, we elaborate in detail on these newly discovered vulnerabilities, and their presence in modern browsers.
An overview of all size-exposing techniques we discovered during our evaluation is provided in Table 1.

3.2 OS-based Techniques

In this section, we elaborate on size-exposing techniques that occur at the level of the operating system, on the side of the web server and client. In our analysis, we considered four types of hosting environments for the web server, namely dedicated hosting, shared hosting, and cloud-based solutions (VMs and PaaS). To be able to observe the length of resources in the case of a dedicated hosting environment, an attacker would need to have either physical access, or infect the machine with a malicious binary. In both cases, we argue that the capabilities of the attacker far surpass what is required for a size-exposing attack, thereby making other attack vectors more appealing to the attacker.

The same argument applies to cloud-based hosting. It has been shown that cache-based side-channels attacks can extract sensitive information, including traffic information, in a cross-tenant or cross-VM environment [48, 76, 72]. However, if an attacker would have the capabilities to leverage a cache-based attack to accurately determine the size of a requested resource, this would mean that the attacker could also leverage the cache-based attack to determine (part of) the execution trace, which can be considered as significantly more severe in most scenarios. Given the lack of incentive for an attacker to uncover the resource size by launching a cross-tenant or cross-VM attack, we do not consider this in more detail.

In a shared hosting environment, web requests for several customers are served by the same system. Next to cross-process cache-based side-channel attacks, which can be considered similar to the above-mentioned cross-VM attacks, adversaries can typically also access the system-wide network statistics. These network statistics can be obtained by either running the `ifconfig` command, or by reading it directly from system pseudo-files such as `/proc/net/dev`. As these network statistics report the exact amount of bytes sent and received by a network interface, an adversary could leverage this information to uncover the size of a response. The attacker's accuracy will of course depend on the amount of background traffic, but the ability to coordinate with the victim's browser gives the adversary a strong advantage. Because shared hosting environments are typically used by less popular websites, we consider this type of attack scenario to be unlikely, and thus do not explore this issue further.

On the side of the client, we explored various size-exposing techniques, but found that most techniques either require too many privileges, e.g., infecting the system with a malicious binary, or yield inaccurate results [44]. An interesting exception is the Android operating system, which also keeps track of network statistics. In addition to the global network statistics, Android also exposes network statistics *per user*, which, surprisingly, can be read out by any application without requir-

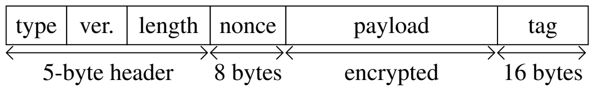

Figure 2: TLS record layout when using AES-GCM.

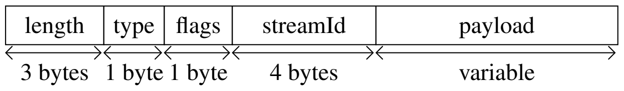

Figure 3: Simplified HTTP/2 frame layout.

ing permissions[1]. In their work, Zhou et al. showed that by passively monitoring network statistics on Android, an adversary can infer sensitive information from the requests made by other applications. We make the observation that these attacks can be extended when considering an attacker model in which the adversary can actively trigger specific requests in the victim's mobile browser. As a proof-of-concept application, we created an HTTP service, which reports the number of bytes received by the user associated with the `com.android.chrome` application. Finally, our applications triggers the mobile browser to open a web page, which first contacts the local service, next downloads an external resource, and then obtains the network statistics again, allowing us to determine the exact size of the external resource.

3.3 Network-based Techniques

We now show the size of a resource can be uncovered by monitoring its transmission over a secure connection. First we do this for TLS, and then we evaluate the case where Wi-Fi encryption is used on top of TLS. Although Wi-Fi hides individual connections, effectively offering a secure channel similar to that of VPNs or SSH tunnels, we show attacks remain possible. We also study the impact of the new HTTP/2 protocol.

3.3.1 Transport Layer Security (SSL / TLS)

Web traffic can be protected by HTTPS, i.e., by sending HTTP messages over TLS [47, 15]. Once the TLS handshake is completed, TLS records of type application data are used to send HTTP messages. The type and length of a record is not encrypted, and padding may be added if block ciphers are used. Since nowadays more than half of all TLS connections use AES in Galois Counter Mode (GCM) [27], we will assume this cipher is used unless mentioned otherwise. The layout of a TLS record using AES-GCM is shown in Fig. 2. Note that for this cipher no padding is used. An HTTP message can be spread out over multiple TLS records, and in turn a TLS record can be spread out over several TCP packets. An endpoint can freely decide in how many records to divide the data being transmitted.

[1]These statistics can be read out from the pseudo-files `/proc/uid_stat/[uid]/tcp_rcv`, or, since Android 4.3, can be obtained from the `getUidRxBytes()` interface.

To determine the length of a resource sent over TLS, we first need to know when it is being transmitted. We accomplish this by using JavaScript to make the victim's browser fetch a page on our server, signaling that the next request will be to the targeted resource. We then monitor any TLS connections to the server hosting this resource, which is possible because the TCP/IP headers of a TLS connection are not encrypted. Once the resource has been received, we again signal this to our server. This enables us to identify the (single) TLS connection that was used to transmit the resource. Finally we subtract the overhead of the TLS records (see Figure 2) to determine the length of the HTTP response. If the connection uses a cipher that does not require padding, this reveals the precise length of the HTTP response. Otherwise only a close estimate of the response length can be made. By subtracting the length of the headers from this HTTP response, whose value can be easily predicted, we learn the length of the requested resource.

We tested this attack against two popular web servers: Apache and nginx. Even when the victim was actively browsing YouTube and downloading torrents, our attack correctly determined the length of the resource. Interestingly, we noticed that Apache puts the header of an HTTP response in a single, separate, TLS record. This makes it trivial to determine the length of the HTTP response header sent by Apache: it corresponds exactly to the first TLS record sent by the server.

We also studied the impact of the HTTP/2 protocol [4] on our attacks. HTTP/2 does not change the semantics of HTTP messages, but optimizes their transport. In HTTP/2, each HTTP request and response pair is sent in a unique stream, and multiple parallel streams can be initiated in a single TCP connection. The basic transmission unit of a stream is a *frame* (see Figure 3). Each frame has a streamId field that identifies the stream it belongs to. Several types of frames exist, with the two most common being header and data frames. Header frames encode and compress HTTP headers using HPACK [45], and data frames contain the body of HTTP messages. Nearly all other frames are used for management purposes, and we refer to them as control frames. Most browsers only support HTTP/2 over TLS. Usage of HTTP/2 is negotiated using the Application Layer Protocol Negotiation (APLN) extension of TLS. This extension is sent unencrypted, meaning we can easily detect if a connection uses HTTP/2.

To determine the size of a resource transmitted using HTTP/2 over TLS, we have to predict the total overhead created by the 9-byte frame header (see Figure 3). Moreover, we need to be able to filter away control frames. Both Apache and nginx send control frames in separate TLS records, and these records can be detected by their length and position in the TLS connection, allowing us to recognize and filter these frames. To calculate the overhead created by the 9-byte frame header, we need to predict the number of HTTP/2 data frames that were used to transmit the resource. For Apache this is easy since it always sends data frames with a payload of 2^{14} bytes, except for the last frame. For nginx, the number of data frames can be predicted based on the number of TLS records. This means that for both servers we can predict the amount of overhead HTTP/2 introduces. The size of the HTTP/2 header frame can be predicted similar to the HTTP/1.1 case, with the addition that the HPACK compression has to be taken into account. Finally, we found that multiple streams are active in one TCP connection only when loading a page. By waiting until the HTTP/2 connection is idle before letting the victim's browser fetch the resource, the only active stream will be the one downloading the resource. All combined, these techniques allowed us to accurately predict the size of resources sent using HTTP/2. Note that if the server uses gzip, deflate, or similar, we learn the compressed size of the resource. In Section 4, we show that this is sufficient to perform attacks, and can even be used to extend an attacker's capabilities.

3.3.2 Encrypted Wi-Fi Networks

Wireless networks are an attractive target for traffic monitoring attacks. For instance, our attack against TLS can be directly applied against open wireless networks. However, these days many wireless networks are protected using WPA2 [71]. This means that all packets, including their IP and TCP headers, are encrypted. Hence we can no longer use these headers to isolate and inspect TLS connections. Nevertheless, we show it is possible to uncover the size of an HTTP message even when Wi-Fi encryption is used on top of TLS.

In the Wi-Fi protocol, the sender first prepends a fixed-length header to the packet being transmitted, and then encrypts the resulting packet [28]. To encrypt and protect a packet, the only available ciphers in a Wi-Fi network are WEP, TKIP, or CCMP. Note that WPA1 and WPA2 are not ciphers, but certification programs by the Wi-Fi Alliance, and these programs mandate support for either TKIP or CCMP, respectively. Since both WEP and TKIP use RC4, and CCMP uses AES in counter mode, padding is never added when encrypting a packet. Therefore, no matter which cipher is used, we can always determine the precise length of the encrypted plaintext. Finally, Wi-Fi encryption is self-synchronizing, meaning that a receiver can decrypt packets even if previous ones were missed or blocked.

Similar to our attack against TLS, we determine when the resource is being transmitted by signaling our own server before and after we fetch the targeted resource. However, we can no longer easily determine which packets correspond to the requested resource as Wi-Fi encrypts the IP and TCP headers. Consequently, any background traffic will interfere with our attack. One option is to execute the attack only if there is no background traffic. Unfortunately, if the user is actively browsing websites or streaming videos, periods without traffic are generally too short. In other words, it is hard to predict whether a period without traffic will be long enough to fetch the complete resource. Our solution is to wait for a small traffic pause, and extend this pause by blocking all packets that are not part of the TCP connection that will fetch the resource. Blocking packets in a secure Wi-Fi network is possible by using a channel-based man-in-the-middle (MitM) attack [61]. Essentially, the attacker clones the Access Point (AP) on a different channel, and forwards or blocks packets to, and from, the real AP. The channel-based MitM also has another advantage: if the adversary misses a packet sent by either a client or AP, the sender will retransmit the packet. This is because the cloned AP, and cloned clients, must explicitly acknowledge packets. Hence our attack is immune to packet loss at the Wi-Fi layer. Once we start measuring the size of the resource, we only forward packets that could be part of the connection fetching this resource. First, this means allowing any packets with a size equal to a TCP SYN or ACK. Second, we have to allow the initial TLS handshake and the HTTP request that fetches the resource. Since both can be detected based on the length of Wi-Fi packets, it is possible to only forward packets that belong to the first TLS handshake and HTTP request. By blocking other outgoing requests, servers will refrain from replying with new traffic. Hence we can still fetch our targeted resource, but all other traffic is temporarily halted.

In experiments the above technique proved highly successful. Even when the victim was browsing websites or streaming YouTube videos, it correctly isolated the TLS connection fetching the resource. We also tested the attack when the victim was constantly generating traffic by sending ping requests of random sizes. Since the size of these packets rarely matches that of a TCP ACK/SYN or TLS handshake packet, all ping requests were blocked, and the correct connection was still successfully isolated.

The next step is to subtract the overhead added by Wi-Fi and TLS. Since none of the cipher suites in Wi-Fi use padding, it is straightforward to remove padding added by the Wi-Fi layer. However, we cannot count the number of TLS records sent as their headers are now encrypted. Nevertheless, for both nginx and Apache with HTTP/1.1, we found that a new TLS record is used for every 2^{14} bytes of plaintext. This allows us to predict the number of TLS records that were used, and thereby the overhead created by these records. We discovered only one exception to this rule. If an Apache server uses chunked content encoding, each chunk is sent in a separate TLS record. This means that the number of TLS records become application-specific, and the attacker has to fine-tune his prediction for every targeted resource. We remark that this behavior of Apache is not recommended, because it facilitates chunked-body-truncation attacks against browsers [5].

When HTTP/2 is used, the situation becomes more tedious. Here we have to predict both the number of TLS records, as well as the number, and types, of HTTP/2 frames. We found that these numbers are predictable for the first HTTP/2 response in a TLS connection. Since all browsers limit the number of open TCP connections, we first close existing connections by requesting several pages hosted on different domains. After doing this, a new connection will be used to fetch the targeted resource, meaning we can predict the amount of overhead. Apache always uses HTTP/2 data frames with a payload of 16348 bytes, even when chunked content encoding is used. Furthermore, the TLS records always have a payload length of 1324, except for every 100th TLS record, which has a length of 296. Finally, Apache always sends the same three HTTP/2 control frames, spread over two TLS records, before sending the resource itself.

For new TLS connections, nginx sends three initial HTTP/2 control frames in either one or two TLS records, where most of the time only one TLS record is used. Then it enters an initialization phase where the first 10 TLS records have a predictable size, with each size taken from the set $\{8279, 8217, 4121, 4129\}$. After this initial phase, it repeats the sequence $[16408, 16408, 16408, 16408, 96]$, with the exception that at relatively infrequent and random times a TLS record of size 60 is used instead of 96. However, as this is only a small difference, it generally affects the number of TLS records by at most one. All combined, if we assume the least number of TLS records are used, we underestimate the actual number of TLS records by at most two. In fact, most of the time no extra records are used. Hence an attacker can make multiple measurements, and pick the most common length as being the one without the extra (one or two) records.

3.4 Browser-based Techniques

Over the last few years, one of the most important evolutions on the web is the increase of support for mobile devices. This advancement requires that all the characteristics that are specific to mobile devices are properly accommodated. For instance, mobile devices travel along with their users, which means that every now and then the devices become disconnected, preventing the user from accessing any web-based content. Recent advancements in browser design aim to tackle this problem with a promising API named ServiceWorker [66]. The core idea behind the SeviceWorker API is to allow websites to gracefully handle offline situations for their users. For example, a news website might download and temporarily store news articles when users are connected, allowing them to still access these while being disconnected. Note that although we mainly focus on the ServiceWorker API, all attacks can also be applied by using ApplicationCache [63], the caching mechanism that ServicerWorker aims to replace.

Algorithm 1 Uncover the size of resources by abusing the per-site quota limit

```
$response \leftarrow fetch(url)$
$fillStorage()$
$size \leftarrow 0$
loop
    $freeByteFromCache()$
    $size \leftarrow size + 1$
    $storageResult \leftarrow cache.put(response)$
    if $storageResult == True$ then
        return $size$
    end if
end loop
```

3.4.1 Per-site quota

For caching operations, the ServiceWorker API provides a specific set of interfaces, named Cache API, which can be used to store, retrieve and delete resources. A noteworthy aspect of the Cache API is that it allows one to cache any resource, including cross-origin responses. Furthermore, to limit misuse cases where a malicious player takes up all available space, the per-site[2] storage is restricted. This restriction is shared among a few other browser features that allow persistent data storage, for instance `localStorage` and IndexedDB. The way per-site quota is applied, is decided by the browser vendor; for the most popular browsers this is either a fixed value in the range of 200MB to 2GB, or a percentage - typically 20% - of the global storage quota [22, 42, 32].

For the purpose of exposing the size of resources, having full control over the cache, and the fact that this cache

[2]According to the current specification of the Storage API, a site is defined as `eTLD+1`, meaning `foo.example.org` and `bar.example.org` belong to the same site, whereas `foo.host.com` belongs to a different site [70].

is limited by a fixed quota, are two very interesting aspects. An adversary can directly leverage these two features to expose the size of any resource by means of the pseudo-code listed in Algorithm 1. In the attack, the resource is first downloaded using the Fetch API, which will result in an "opaque" Response. Next, the adversary makes sure that the site's available storage is filled up to the quota. In practice, we found that by storing large data blobs using the IndexedDB API, the storage speed approaches the maximum writing speed of the hard disk, allowing the attacker to reach the quota in a few seconds. In a final step, the adversary will free up one byte from the cache and attempt to store the response. This storage attempt will only succeed if sufficient quota is available, otherwise more bytes should be freed. Eventually, the attacker learns the exact size of the resource by the number of bytes that were freed until the resource could be stored. Note that the resource only needs to be downloaded once, resulting in a significant speed-up of the attack. In our experimental setup, the initial attack could be executed in less than 20 seconds, and subsequent size-exposing attempts were performed in less than a second as the quota had already been reached.

3.4.2 Global quota

In addition to the storage restrictions of sites, browsers also enforce a global storage quota to ensure normal system operations are not affected. When this global quota is exceeded, the storage operation will not be canceled, but instead the storage of the least-recently used site will be removed. As a result, the two features required to expose the size of a resource, i.e., full control over the cache and an indication when the quota is exceeded, are present. In comparison to the size-exposing attack that leverages the per-site quota, this vulnerability is considerably harder to successfully exploit: the attacker needs to reach the global quota limit, which needs to be spread over multiple sites, and has to take into account that the global quota can fluctuate as a result of unrelated system operations. Nevertheless, for the purpose of creating an improved design, it is important to consider all flaws of the current system. Furthermore, on systems with a limited storage capacity, e.g., mobile devices, some of these restrictions may not apply, increasing the feasibility of an attack.

A simplified, unoptimized method that can be used to expose the size of an arbitrary resource is provided in Algorithm 2. Similar to the per-site quota attack, the adversary first downloads the resource and temporarily stores it in a variable. Next, a site is filled with a certain amount of bytes (*storageAmount*) which should be larger than the size of the resource. In a following step, the adversary will need to fill the complete quota. Since

Algorithm 2 Uncover the size of resources by abusing the global quota limit

```
$response \leftarrow fetch(url)$
$storageAmount \leftarrow 5\text{MB}$
$site_0.addBytes(storageAmount)$
$i \leftarrow 1$
while $!isEvicted(site_0)$ do
    $storageResult \leftarrow site_i.addBytes(1)$
    if $storageResult\ != True$ then
        $i \leftarrow i + 1$
    end if
end while
$site_0.cache.put(response)$
$remainingBytes \leftarrow 0$
while $!isEvicted(site_1)$ do
    $site_0.addBytes(1)$
    $remainingBytes \leftarrow remainingBytes + 1$
end while
$size \leftarrow storageAmount - remainingBytes$
```

for most major browsers, the global quota is set to 50% of the total available space on the device, and the per-site quota is set to either a percentage of the global quota or a fixed size, the adversary will need to divide this over multiple domains. As soon as the eviction of the first site is triggered, the adversary knows the exact amount of freed space, namely *storageAmount*. Finally, the adversary adds the resource to an empty site and fills it until the global quota is reached again, which can be observed by checking for the eviction of the next least-recently used site, i.e., $site_1$. The size of the resource can then be calculated as the original size of the first site subtracted by the number bytes required to reach the global quota again (*remainingBytes*).

3.4.3 Quota Management API & Storage API

The last attack involving browser storage abuses the Quota Management API [65], and the similar Storage API [70]. These APIs aim to give web developers more insight into their website's storage properties, more specifically the number of bytes that have been stored and the space that is still available. At the time of writing, the Storage API is still being designed, and will consolidate the storage behavior of all browsers into one agreed-upon standard.

The functionality provided by the Quota Management API is the direct source of a size-exposing vulnerability that is worryingly trivial to exploit. An adversary can simply request the current storage usage, add a resource to the cache, and retrieve the storage usage again. Since the Quota Management API will return the usage in bytes, the *exact* resource size can be obtained by subtracting the two usage values. Although the Quota

Management API has only been adopted by the Google Chrome browser, this browser alone accounts for approximately 48% of the market share [56], leaving hundreds of millions of internet users vulnerable to this highly trivial size-exposing attack vector. Despite our efforts of reporting these findings to the Chrome team, all up-to-date versions of the Google Chrome browser remain allowing this API to be used by any website, without the user's knowledge.

Because the per-site quota is related to the global quota[3], the Quota Management API can also be used to infer the caching operations of a different website. For instance, a malicious iframe that is embedded on a website could observe changes in the available quota, and infer the length of cached resources. This information could in turn be used to either analyze the interactions of the user on the website, or disclose private information based on the length of the cached resources. A similar attack scenario is discussed in more detail in Section 4.4. Another interesting case occurs when making the observation that the per-site quota is also related to the total free disk space. The byproduct of this behavior is that an adversary can also observe the disk operations of other, possibly security-sensitive, processes. As this issue is unrelated to size-exposing techniques, we do not explore this vulnerability in more detail.

The functionalities provided by the Quota Management API are directly responsible for the vulnerabilities discussed in this section. It is unclear why this API was developed without taking into account potential security and privacy implications. In essence, these findings serve as a strong indicator that new browser features should be thoroughly reviewed for security and privacy flaws. Since the Storage API provides the same functionality as the Quota Management API, the same issues arise there as well. At the time of writing, the Storage Standard deviates from the Quota Management API in the sense that it states that a "rough estimate" should be returned. Because the term "rough estimate" is not formally defined, implementations of this specification are likely to still be vulnerable to statistical attacks, as the quota limit can easily be requested thousands of times. In Section 5.1 we propose a new API design that protects against all browser-based size-exposing techniques we discussed in this paper.

4 Real-world Consequences

In contrast to prior work on size-exposing techniques, which is mainly focused on passive network observation, the attacks presented in this paper leverage the ability to request arbitrarily chosen resources in the victim's browser. To provide more insight into the consequences and potential attack scenarios, we explore a selection of real-world cases where one of the size-exposing techniques can be used to extract private and sensitive information from the victim. The list of attacks that are discussed, is by no means the exclusive list of possible targets. Instead, we made a selection of attack scenarios to provide a variety in methodology, type of disclosed information, and category of web service.

Ethical Considerations To evaluate the severity and impact of size-exposing techniques on internet users, it cannot be avoided to evaluate these attacks on real-world services. To prevent any nefarious consequences of this evaluation, all attacks were manually tested, and were performed exclusively against our own accounts. As a result, from the perspective of the tested services our analysis only generated a restricted amount of legitimate traffic. Moreover, users of the analyzed websites were not directly involved in our attacks. For the quantitative case-studies, we only obtained publicly available information, and present it in anonymized form. Given the above-mentioned precautions, we believe our evaluation of real-world services did not have any adverse effects on the tested subjects.

4.1 User Identification

Virtually every online social network provides its users with their own profile page. Depending on the user's privacy settings, these profile pages typically are completely or partially available to anyone. In the attack scenario where the adversary is interested in learning the identity of the victim, the adversary first collects the publicly available data from (a subset of) the users of the social network. Later, during the actual size-exposing attack, he tries to associate the data obtained from the victim to a single entry from the public data, allowing him to expose the victim's identity. To evaluate the feasibility in a real-world environment, we exemplify the attack scenario on Twitter, one of the largest social networks.

By default, the profile of each Twitter user is public, and contains information on the latest tweets that were created by the user, the list of followers and followees, the tweets that were "liked" by the user, and the lists he/she follows and is a member of. Except for the user's tweets, each type of information can be accessed by a link that is shared by all Twitter users, e.g., the page located at `https://twitter.com/followers` lists the last 18 accounts that follow the user. For each follower, the name, account name and short biography is shown.

The main assumption in this attack scenario is that the combined length of all parts that constitute to the

[3]The per-site quota is 20% of the global quota in Google Chrome; for Firefox this is the case as well when the disk space is less than 20GB.

resource, i.e., the names, account names and bios of the last 18 followers, is relatively unique. To validate this assumption, we performed an experiment that reflects an adversary's actions in an actual attack scenario. For this experiment, we obtained publicly available information of 500,000 users, which were selected at random from the directory of public profiles provided by Twitter[4]. More specifically, we downloaded the resources located at `/following`, `/followers`, `/likes`, `/lists` and `/memberships`, and recorded the associated resource size, both with and without gzip compression.

Next, we grouped together Twitter accounts that share the same resource length, e.g., if the `/following` resource is 281026 bytes for only two users, these users form a group of size 2. In Figure 4 we show the percentage of Twitter accounts for all group sizes, for the compressed and uncompressed resource size. Note that a logarithmic scale is used for the percentage of Twitter accounts on the y-axis. This graph clearly shows that when the size of multiple resources is combined, the majority of Twitter accounts can be uniquely identified. By exposing the size of the uncompressed `/following` and `/followers` resources, 89.66% of the 500,000 Twitter accounts can be uniquely identified. When the size of all five resources is known, the identity of 97.62% of the Twitter accounts can immediately be uncovered. The graph also clearly shows that when gzip compression is applied, the group sizes of individual resources becomes larger, which is most likely due to the reduction in entropy of resource sizes. Nevertheless, when the size of multiple compressed resources are combined, a uniqueness comparable to the size of uncompressed resources is achieved: 81.69% Twitter accounts can be uniquely identified when the size of the `/following` and `/followers` resources is combined; for all five resources, this is 99.96%. The most likely explanation for this is that in case a resource is virtually empty, i.e., the account name is the only dynamic part of the resource, not only the length but also the content of the account name is reflected in the compressed resource size.

Although the viability of this attack was only evaluated on a subset of all Twitter accounts[5], this experiment does suggest that adversaries can immensely narrow down the number of possible candidates for the user's identity by knowing the size of just five resources. Furthermore, various techniques exist that can uniquely identify a user among a limited set of accounts [33, 26], making user-identification by exposing the size of resources well within the reach of a moderately motivated attacker.

[4]`https://twitter.com/i/directory`

[5]Twitter has approximately 320 million active accounts.

4.2 Revealing Private Information

Next to revealing the identity of a web user, adversaries may also be interested in learning private information. A particular type of information that, in general, is considered highly sensitive, is information concerning medical conditions. To evaluate whether our novel size-exposing techniques can be used to also disclose this type of data, we explored the performance of such techniques on WebMD, one of the leading health information services websites. One of the features provided by WebMD is "Health Record", a web service that allows users to organize their personal health records[6]. More precisely, users can add, and keep track of, their medical conditions, medications, allergies, etc. For each entry, the user can choose among an exhaustive list of terms. For instance, there are 4,105 different medical conditions that can be selected.

At any point in time, users can download their own medical report, either as automatically generated PDF or in plain text format. It should be noted that the types of medical records that are shown in this report is specified by the user (or attacker), and that the PDF is sent without compression, whereas the textual report is served with gzip compression. Although there is some variety in the length of the possible terms, it is insufficient for an adversary to determine which medical conditions the user suffers from: on average, a certain length is shared among 124.59 possible medical conditions. However, if the adversary can obtain the resource size both with and without compression, this can significantly improve his attack: in this case, the group size can be limited to 35.50 on average. This can be achieved by various methods, e.g., by obtaining the length from two resources that share the same content, where one is served with compression and the other without, or by tricking the server in sending the resource without compression[7], or even by combining the browser-based attacks with the network-based attacks. In case the sensitive content is present on multiple compressed resources (in this case, this can be triggered by varying the types of medical records that are reported), the group size can be reduced even further. In the attack scenario against WebMD, a single iteration of this technique, i.e., including the medical condition on a compressed resource with other known content, reduces the average group size to 18.73. By applying multiple iterations, each with slighly different content, it becomes possible to uniquely identify the user's medical condition in most cases.

[6]`https://healthmanager.webmd.com/`

[7]When a resource is included as a `<video>` element, the `Accept-Encoding` header will be either absent or set to `identity`, causing most web servers to send it without compression.

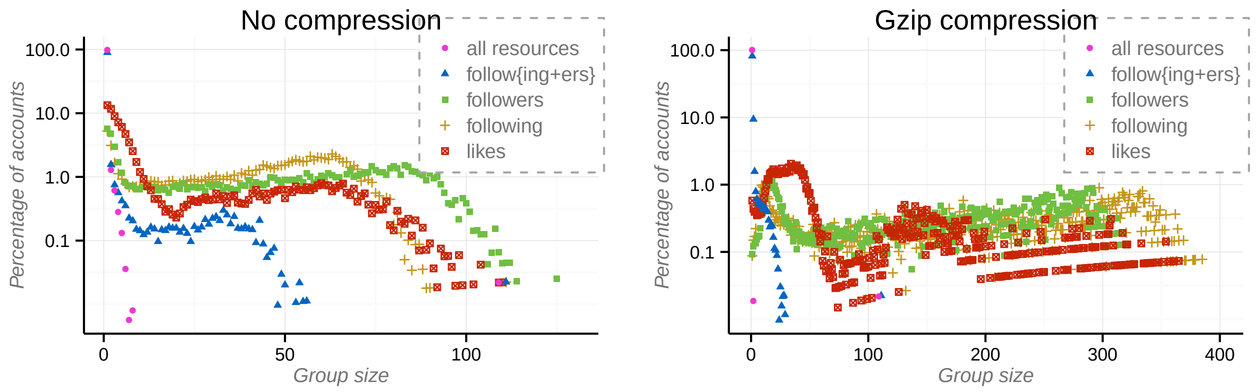

Figure 4: Percentage of Twitter accounts that share the same resource length with a group of varying size.

4.3 Search-Oriented Information Leakage

Many web applications allow their users to search the data they (in)directly entered. For instance, web-based e-mail clients provide the functionality to search for certain messages. In a recent study, Gelernter et al. show that this functionality can be abused by attackers to disclose sensitive information, such as the user's identity and credit-card numbers [20]. In their attacks, the researchers leverage the fact that in certain cases query parameters are reflected in the results. Consequently, when a search query has several matches, the resulting resource size will be considerably larger than with an empty result-set, allowing an adversary to resort to timing attacks to determine whether a certain search query yielded results. Several service providers that were shown to be vulnerable to these attacks implemented a mitigation by preventing query parameters to be reflected in the search results. Although these measures effectively thwart the above-mentioned attacks, the web services remain vulnerable to the size-exposing attacks proposed in this paper, as these disclose the size of a resource with 1-byte precision.

4.4 Cross-Origin Cache Operations

Telegram is a popular cloud-based instant messaging service, particularly known for its security and encryption features. Not surprisingly, these features have attracted terrorist organizations to use the service as a secure communication channel [53]. This, in turn, makes Telegram a valuable target for intelligence agencies to find members of terrorist groups. Since all exchanged messages are encrypted using MTProto, which was shown to only suffer from minor theoretical attacks, plaintext-recovery is considered to be unlikely [30].

Next to the mobile and desktop versions of the Telegram application, a web-based version is provided as well[8]. An interesting feature of this web-based version is that when a photo is shared in a group, the web application will use the File API [46] to cache two thumbnails of the photo. Because the storage used by the File API counts towards the global cache quota, it is possible to infer whether a resource is being cached as per the attacks discussed in Section 3.4.2 and Section 3.4.3.

In an attack scenario where the adversary tries to determine group membership of the victim, the attacker first lures the victim to his malicious web page. On this web page, the adversary includes the page of the target group in an iframe. Telegram does not use the `X-Frame-Options` header, but instead makes the content invisible by default through CSS, and uses JavaScript to make it visible in case no framing is detected (a popular Clickjacking defense proposed by Rydstedt et al. [51]). As a result, the page's content will be loaded, but remains invisible, and impossible to interact with[9]. If the user is member of the targeted group, the Telegram website will download and cache thumbnails of the latest media items that have been shared in the group, resulting in a change of the available quota. Otherwise, a message is shown stating that the user is not a member of the group. As an additional verification step, the adversary could post another photo in the group, and witness a change in the available quota. By leveraging our novel size-exposing techniques, we found it was trivial to detect group membership. Because the MTProto scheme only provides very limited padding, group membership can also easily be detected by analyzing the size of HTTP responses.

5 Defense Mechanisms

In this section we discuss various mechanisms that can be used to thwart size-exposing attacks. Due to space limitations, we only focus on a limited set of defense mechanisms, which were selected on the basis of completeness, novelty, amount of overhead and ease of adoption.

[8]`https://web.telegram.org`

[9]The `<iframe>` element should have a `sandbox` attribute set to `"allow-scripts allow-same-origin"` to prevent top level navigation, while ensuring the page is loaded properly.

5.1 Hardening Browser Storage

As was shown in Section 3.4, several features related to the storage operations in browsers can be abused to expose the size of cross-origin resources. At the time of writing, there exists no universal specification that standardizes these operations. However, the Storage API specification is being developed with the purpose of designing a unified definition that will be adopted by all browsers. In its current state, the Storage API consolidates the current browsers behavior regarding the quota limit per website. Furthermore, it incorporates the functionalities offered by the Quota Management API.

We propose a countermeasure that extends the Storage API. To make adoption by browsers feasible, we aim to provide a usable solution, i.e., normal application behavior should not be jeopardized. As a result of the feedback provided by the communication with specification editors and browser vendors, we opted for an approach where "virtual padding" is applied to resources. To prevent an adversary from learning the size of a resource, either by abusing the storage limit or by requesting the available quota, this size should be masked with a random value. However, it is a well-known fact that by adding a random value, the mechanism becomes subject to statistical attacks. Because resources can be added to the cache extremely fast, an adversary is able to obtain a large number of observations in a limited amount of time, putting him in a very strong position.

Inspired by a mitigation for web-based timing side-channels proposed by Schinzel [52], and by making the observation that in contrast to caching operations, downloading a resource takes a considerable amount of time, we propose the following defense. When a resource is downloaded as the result of a `fetch()` operation, we associate a unique identifier, *uid*, with the `Response` object. Next, we compute $q = \lceil size + hash(secret + uid) \rceil_\Delta$, where *size* is the size of the resource, $hash()$ a uniformly distributed hash function yielding integers in the range $[0, p_{max}]$, and *secret* a cryptographic random number that is associated to a single browsing session[10]. The total size q is then rounded up towards the nearest multiple of Δ to prevent an attacker from learning the bounds of the added padding. When the `Response` is added to the cache, the per-site and global quota will be increased by q. This value should also be stored as part of the `Response` object to ensure that for each cache operation the same value is either added or subtracted from the quota. As a result, the only way for an adversary to obtain a new observation is to download the same resource again. It should be noted that the padding that is added for each cache operation is virtual, in the sense that these bytes are not actually written to the disk, but are just kept as a type of bookkeeping.

It is clear that the overhead on the quota and the security guarantees provided by this defense method are directly related to the values of p_{max} and Δ. In fact, this provides a trade-off between security/privacy and usability, for instance, the larger the value of p_{max}, the harder it will be for an adversary to uncover the size of resources (within certain boundaries), but on the other hand, a large p_{max} will entail a smaller storage capacity due to the amount of padding. We argue that with an analysis on the typical use-cases of caching operations, these values could be defined to accommodate legitimate behavior while preventing attacks. Furthermore, it could be taken into account that this mechanism generates a virtual loss in storage capacity, and therefore the quota could be increased to account for this. In addition, it is possible to apply a rate-limiting approach to limit the amount of observations that can be made by an adversary. For instance, if the reported quota is only updated once every minute, statistical attacks can be largely mitigated, which in turn allows for smaller values of p_{max}, and restricts the (already virtual) overhead.

Given the generality of the defense, its strong security guarantees, and the low overhead, we feel confident that this approach, or a similar derivative thereof, will be incorporated into the HTML specification, and encourage browser vendors to mitigate the attacks presented in Section 3.4 in this manner.

5.2 Detecting Illicit Requests

In essence, the size-exposing techniques presented in this paper require the ability to initiate authenticated cross-origin requests, and rely on the targeted web service to handle the request in the same way it would for legitimate requests. This means that when either part is removed, i.e., either authenticated cross-origin requests are disabled, or the web server answers with a static error message, the complete class of size-exposing techniques will be mitigated. To accomplish this, it is possible to resort to existing, and well-established techniques in related research fields. For instance, by blocking third-party cookies, which is typically used to prevent tracking on the web [50], the cross-origin requests initiated by the adversary will be sent without the cookie. As a result, the website will handle the request as if the user was not logged in, preventing the adversary from learning anything about the user's state at the website. Mozilla and the Tor Browser project are working on minimizing the limitations imposed by blocking third-party cookies, by implementing a feature name double-keyed cookies, which binds cookies to the origin pair (first-party, third-party), and aims to prevent the risks of breaking sites

[10]To prevent an adversary from linking two browser sessions, *secret* is changed whenever the browser session changes.

caused by blocking cookies [9, 59]. Similarly, certain browsers provide the ability to attach third-party cookies only if these were set during top-level navigation, and block these otherwise. While this technique can be used to prevent tracking by unknown parties, it does not adequately prevent the attacks presented in this paper as the targeted third-party services are the ones that are actually used by the victim.

On the side of the server, solutions similar to those that prevent Cross-Site Request Forgery (CSRF) attacks could be applied. A well-known method, as proposed by Barth et al., to accomplish this, is to analyze the `Origin` and/or `Referer` headers and only allow requests from trusted origins [2].

5.3 Network-based Countermeasures

Padding can be used to hide the length of resources during their transmission. Since general-purpose padding schemes are already well-studied, we do not discuss them further. Instead, we focus on countermeasures that fit our use-case, where only the size of sensitive dynamically generated resources must be protected. This allows us to provide a countermeasure with low overhead and high security guarantees, at the cost of requiring some effort on the web administrator's part.

Our idea is to add an amount of padding based on the hash of the session cookie, the URL, and any parameters that affect the generation of the resource. More formally, $padding = hash(cookie + url + params)$. If the user is not logged in, no padding is added. For each resource, the parameters that influence the generation of the resource must be manually specified. Other parameters should not be included, otherwise an adversary can add bogus parameters to obtain a new padding value for the same resource. This construction assures that sensitive resources, for any specific user, receive an amount of padding that is unpredictable by an attacker. However, this padding remains identical over several requests, meaning it even guarantees protection against statistical attacks. Information can only be leaked if the resource changes over time. This can happen when the attacker was able to affect the generation of the resource on the server, or simply because the information contained in the resource has changed over time. In this situation an observer can learn the difference in resource size. If the resource does not contain variable content, such as dynamic advertisements, this attack can be mitigated by including the content of the resource in the hash function. Similar to hardening the browser (see Section 5.1), the security guarantees depend on the value of p_{max}. Provided the hash function is uniformly distributed, this countermeasure introduces on average $\frac{p_{max}}{2}$ bytes of overhead.

For wireless networks, where we assume Wi-Fi encryption is used on top of TLS, we can rely on the previously mentioned techniques to protect the TLS connection. Additionally, an identifier-free wireless protocol can be used, making it more difficult for an attacker to attribute Wi-Fi packets to specific clients [23, 18, 3, 8].

6 Related Work

Size-exposing techniques have surfaced in several research areas, ranging from timing attacks, to network traffic analysis, to browser-based and cross-VM side-channel leaks. As part of an in-depth analysis, which lead to the discovery of multiple novel attack methods, we already touched upon a variety of related work, as discussed in Section 3. In this section, we give a brief overview of the most relevant work, and discuss it in the context of our findings.

Prior research that analyses methods that can expose the size of an attacker-specified resource, is mainly focused on leveraging timing as a side-channel information leak [19, 7, 14, 20, 60]. Because timing attacks measure the time required to download or process a resource, which is often influenced by various factors such as network irregularities or background noise, these attacks have certain limitations with regards to the accuracy of the uncovered resource size. In our research, we presented novel techniques that leverage the browser-imposed quota to reveal the *exact* size of any resource.

An interesting class of vulnerabilities where the size of resources is exploited, are compression side-channel attacks [31]. These attacks generally leverage the compression rate that is achieved when compressing an unknown value in a larger corpus of known values, allowing an adversary to uncover information about the unknown value from the resource size after compression. More recently, researchers have shown how similar attacks can be applied to various compression mechanisms used on the web [49, 21].

In the context of privacy-violating cross-origin attacks, Lee et al. have shown that the ApplicationCache mechanism can be used to uncover the status code that is returned for cross-origin resources [34]. Their attack exploits certain intricacies of ApplicationCache, which exhibits a different behavior based on the returned status code of referenced endpoints. The researchers did not explore vulnerabilities originating from the imposed quota and storage limits. Another type of attack that violates the principle of Same-Origin Policy is Cross-Site Script Inclusion (XSSI), first introduced by Grossman in 2006 [24], and recently analyzed on a wide scale by Lekies et al. [35]. In XSSI attacks, a dynamically generated JavaScript (or CSV [58]) file from a vulnerable website is included as a `<script>` element on the web page of the attacker. The often sensitive content that is present in these files can then be obtained out by the adversary as a result of the modifications the script makes to the attacker-controlled DOM.

Compared to prior work on the analysis of web traffic [57, 6, 12, 36, 11, 10, 17], our work is, to the best of our knowledge, the first to combine traffic analysis with the ability to execute code in the victim's browser. Similarly, traffic analysis works on Wi-Fi also assume a passive, instead of an active, adversary [8, 23, 3, 75, 74]. That is, we believe our work is the first to actively block specific Wi-Fi packets in order to measure the size of HTTP messages.

7 Conclusion

The size of resources can be used to infer sensitive information from users at a large number of web services. In our research, we performed an extensive analysis on the various operations that are performed on resources. As a result of this evaluation, we identified several new techniques that can be used to uncover the size of any resource. In particular, an attack that abuses the storage quota imposed by browsers, as well as a novel technique against Wi-Fi networks that can be used to disclose the size of the response associated with an attacker-initiated request. To provide more insight into how these attack methods can be applied in real-world attack scenarios, we elaborated on several use cases involving widely used web services. Motivated by the severe consequences of these size-exposing attacks, we proposed an enhanced design for the browser storage, which is likely to be adopted by browser vendors, and discussed a variety of other options that could be employed to prevent adversaries from stealing sensitive information.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This research is partially funded by the Research Fund KU Leuven. Mathy Vanhoef holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

References

[1] Al Fardan, N. J., and Paterson, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy* (2013).

[2] Barth, A., Jackson, C., and Mitchell, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 75–88.

[3] Bauer, K., McCoy, D., Greenstein, B., Grunwald, D., and Sicker, D. Physical layer attacks on unlinkability in wireless lans. In *Privacy Enhancing Technologies* (2009).

[4] Belshe, M., Peon, R., and Thomson, M. Hypertext transfer protocol version 2 (HTTP/2). RFC 7540, 2015.

[5] Bhargavan, K., Lavaud, A. D., Fournet, C., Pironti, A., and Strub, P. Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Security and Privacy (SP)* (2014).

[6] Bissias, G. D., Liberatore, M., Jensen, D., and Levine, B. N. Privacy vulnerabilities in encrypted HTTP streams. *Lecture notes in computer science 3856* (2006), 1.

[7] Bortz, A., and Boneh, D. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, pp. 621–628.

[8] Brik, V., Banerjee, S., Gruteser, M., and Oh, S. Wireless device identification with radiometric signatures. In *Mobile computing and networking* (2008).

[9] Bugzilla. Bug 565965 - (doublekey) key cookies on setting domain * toplevel load domain. `https://bugzilla.mozilla.org/show_bug.cgi?id=565965`, May 2010.

[10] Cai, X., Zhang, X. C., Joshi, B., and Johnson, R. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 605–616.

[11] Chapman, P., and Evans, D. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 263–274.

[12] Chen, S., Wang, R., Wang, X., and Zhang, K. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 191–206.

[13] Cheng, H., and Avnur, R. Traffic analysis of SSL encrypted web browsing. *URL citeseer. ist. psu. edu/656522. html* (1998).

[14] Crosby, S. A., Wallach, D. S., and Riedi, R. H. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC) 12*, 3 (2009), 17.

[15] Dierks, T., and Rescorla, E. The transport layer security (TLS) protocol version 1.2. RFC 5246, 2008.

[16] Duong, T., and Rizzo, J. Here come the xor ninjas. In *Ekoparty Security Conference* (2011).

[17] Dyer, K. P., Coull, S. E., Ristenpart, T., and Shrimpton, T. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Security and Privacy (SP)* (2012).

[18] Fan, Y., Lin, B., Jiang, Y., and Shen, X. An efficient privacy-preserving scheme for wireless link layer security. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE* (2008).

[19] Felten, E. W., and Schneider, M. A. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security* (2000), ACM, pp. 25–32.

[20] Gelernter, N., and Herzberg, A. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1394–1405.

[21] Gluck, Y., Harris, N., and Prado, A. BREACH: reviving the CRIME attack. In *Black Hat Briefings* (2013).

[22] Google Chrome. Managing HTML5 offline storage. `https://developer.chrome.com/apps/offline_storage`, February 2016.

[23] Greenstein, B., McCoy, D., Pang, J., Kohno, T., Seshan, S., and Wetherall, D. Improving wireless privacy with an identifier-free link layer protocol. In *Mobile systems, applications, and services* (2008).

[24] Grossman, J. Advanced web attack techniques using GMail. `http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html`, 2006.

[25] Hintz, A. Fingerprinting websites using traffic analysis. In *Privacy Enhancing Technologies* (2003), Springer, pp. 171–178.

[26] Homakov, E. Using Content-Security-Policy for evil. `http://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html`, January 2014.

[27] ICSI. The ICSI certificate notary. Retrieved 23 Jan. 2016, from `http://notary.icsi.berkeley.edu`.

[28] IEEE Std 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 2012.

[29] Jagatic, T. N., Johnson, N. A., Jakobsson, M., and Menczer, F. Social phishing. *Communications of the ACM 50*, 10 (2007), 94–100.

[30] Jakobsen, J. B., and Orlandi, C. *A practical cryptanalysis of the Telegram messaging protocol*. PhD thesis, Master Thesis, Aarhus University (Available on request), 2015.

[31] Kelsey, J. Compression and information leakage of plaintext. In *Fast Software Encryption* (2002), Springer, pp. 263–276.

[32] Kitamura, E. Working with quota on mobile browsers. `http://www.html5rocks.com/en/tutorials/offline/quota-research/`, January 2014.

[33] Landau, P. Deanonymizing Facebook users by CSP bruteforcing. `http://www.myseosolution.de/deanonymizing-facebook-users-by-csp-bruteforcing/`, August 2014.

[34] Lee, S., Kim, H., and Kim, J. Identifying cross-origin resource status using application cache. In *NDSS* (2015).

[35] Lekies, S., Stock, B., Wentzel, M., and Johns, M. The unexpected dangers of dynamic JavaScript. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 723–735.

[36] Luo, X., Zhou, P., Chan, E. W., Lee, W., Chang, R. K., and Perdisci, R. HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows. In *NDSS* (2011).

[37] Marlinspike, M. New tricks for defeating SSL in practice. *BlackHat DC, February* (2009).

[38] Mather, L., and Oswald, E. Pinpointing side-channel information leaks in web applications. *Journal of Cryptographic Engineering 2*, 3 (2012), 161–177.

[39] Microsoft. Platform status. `https://dev.windows.com/en-us/microsoft-edge/platform/status/fetchapi`, February 2016.

[40] Miller, B., Huang, L., Joseph, A. D., and Tygar, J. D. I know why you went to the clinic: Risks and realization of HTTPS traffic analysis. In *Privacy Enhancing Technologies* (2014), Springer, pp. 143–163.

[41] Moore, T., and Edelman, B. Measuring the perpetrators and funders of typosquatting. In *Financial Cryptography and Data Security*. Springer, 2010, pp. 175–191.

[42] Mozilla Developer Network. Browser storage limits and eviction criteria. `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria`, October 2015.

[43] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 736–747.

[44] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox: Practical cache attacks in JavaScript. *arXiv preprint arXiv:1502.07373* (2015).

[45] Peon, R., and Ruellan, H. HPACK: Header compression for HTTP/2. RFC 7541, 2015.

[46] Ranganathan, A., and Sicking, J. File API. *W3C Working Draft* (2012).

[47] Rescorlan, E. HTTP over TLS. RFC 2818, 2000.

[48] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 199–212.

[49] Rizzo, J., and Duong, T. The CRIME attack. In *EKOparty Security Conference* (2012), vol. 2012.

[50] Roesner, F., Kohno, T., and Wetherall, D. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 12–12.

[51] Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web 2* (2010), 6.

[52] Schinzel, S. An efficient mitigation method for timing side channels on the web. In *2nd International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)* (2011).

[53] Segall, L. An app called Telegram is the 'hot new thing among jihadists'. `http://money.cnn.com/2015/11/17/technology/isis-telegram/`, November 2015.

[54] Sood, A. K., and Enbody, R. J. Malvertising: Exploiting web advertising. *Computer Fraud & Security 2011*, 4 (2011), 11–16.

[55] SSL Pulse. Survey of the SSL implementation of the most popular web sites. `https://www.trustworthyinternet.org/ssl-pulse/`, February 2016.

[56] StatCounter. GlobalStats. `http://gs.statcounter.com/#all-browser-ww-monthly-201501-201601`, January 2016.

[57] Sun, Q., Simon, D. R., Wang, Y.-M., Russell, W., Padmanabhan, V. N., and Qiu, L. Statistical identification of encrypted web browsing traffic. In *Security and Privacy* (2002).

[58] Terada, T. Identifier based XSSI attacks. `https://www.mbsd.jp/Whitepaper/xssi.pdf`, March 2015.

[59] Tor. Isolate HTTP cookies according to first and third party domain contexts. `https://trac.torproject.org/projects/tor/ticket/3246`, May 2011.

[60] Van Goethem, T., Joosen, W., and Nikiforakis, N. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1382–1393.

[61] Vanhoef, M., and Piessens, F. Advanced Wi-Fi attacks using commodity hardware. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM, pp. 256–265.

[62] VANHOEF, M., AND PIESSENS, F. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium* (2015).

[63] W3C. Offline web applications. `https://www.w3.org/TR/offline-webapps/`, May 2008.

[64] W3C. Same-origin policy. `https://www.w3.org/Security/wiki/Same_Origin_Policy`, January 2010.

[65] W3C. Quota management API. `https://www.w3.org/TR/quota-api/`, December 2015.

[66] W3C. Service Workers. `https://www.w3.org/TR/service-workers/`, June 2015.

[67] WAGNER, D., SCHNEIER, B., ET AL. Analysis of the SSL 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings* (1996), pp. 29–40.

[68] WANG, T., AND GOLDBERG, I. Comparing website fingerprinting attacks and defenses. Tech. rep., Technical Report 2013-30, CACR, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-30.pdf, 2014.

[69] WEBKIT. Implement fetch API. `https://bugs.webkit.org/show_bug.cgi?id=151937`, December 2015.

[70] WHATWG. Storage. `https://storage.spec.whatwg.org/`, August 2015.

[71] WIGLE. WiFi encryption over time. Retrieved 6 Feb. 2016 from `https://wigle.net/enc-large.html`.

[72] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 719–732.

[73] ZALEWSKI, M. *The tangled Web: A guide to securing modern web applications*. No Starch Press, 2012.

[74] ZHANG, F., HE, W., CHEN, Y., LI, Z., WANG, X., CHEN, S., AND LIU, X. Thwarting Wi-Fi side-channel analysis through traffic demultiplexing. *Wireless Communications, IEEE Transactions on 13*, 1 (2014), 86–98.

[75] ZHANG, F., HE, W., AND LIU, X. Defending against traffic analysis in wireless networks through traffic reshaping. In *Distributed Computing Systems (ICDCS)* (2011).

[76] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 990–1003.

[77] ZHOU, X., DEMETRIOU, S., HE, D., NAVEED, M., PAN, X., WANG, X., GUNTER, C. A., AND NAHRSTEDT, K. Identity, location, disease and more: Inferring your secrets from Android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1017–1028

.