HEIST: HTTP Encrypted Information can be Stolen through TCP-windows

Mathy Vanhoef and Tom Van Goethem
iMinds-DistriNet
mathy.vanhoef@cs.kuleuven.be - tom.vangoethem@cs.kuleuven.be

Over the last few years, a worryingly number of attacks against SSL/TLS and other secure channels have been discovered. Fortunately, at least from a defenders perspective, these attacks require an adversary capable of observing or manipulating network traffic. This prevented a wide and easy exploitation of these vulnerabilities. In contrast, we introduce HEIST, a set of techniques that allows us to carry out attacks against SSL/TLS purely in the browser. More generally, and surprisingly, with HEIST it becomes possible to exploit certain flaws in network protocols without having to sniff actual traffic.

HEIST abuses weaknesses and subtleties in the browser, and the underlying HTTP, SSL/TLS, and TCP layers. In particular, we discover a side-channel attack that leaks the exact size of any cross-origin response. This side-channel abuses the way responses are sent at the TCP level. Combined with the fact that SSL/TLS lacks length-hiding capabilities, HEIST can directly infer the length of the plaintext message. Concretely, this means that compression-based attacks such as CRIME and BREACH can now be performed purely in the browser, by any malicious website or script, without requiring a man-in-the-middle position. Moreover, we also show that our length-exposing attacks can be used to obtain sensitive information from unwitting victims by abusing services on popular websites.

Finally, we explore the reach and feasibility of exploiting HEIST. We show that attacks can be performed on virtually every web service, even when HTTP/2 is used. In fact, HTTP/2 allows for more damaging attack techniques, further increasing the impact of HEIST. In short, HEIST is a set of novel attack techniques that brings network-level attacks to the browser, posing an imminent threat to our online security and privacy.

1 Introduction

With initiatives like Let's Encrypt, and CloudFlare's Universal SSL, we are (finally) reaching a stage where most of our online web traffic is encrypted. Unfortunately, we are not

quite there yet. Over the last few years, there has been a trend where every few months a serious vulnerability on SSL/TLS is discovered. Although this poses an immediate and significant threat to our online security, there have been few reports where these vulnerabilities are in fact widely exploited (attacks by government agencies being the exception here). Most of the large service providers are relatively fast at minimizing the threat, either by applying the right defenses or applying sufficient preventive measures (unfortunately, exceptions apply here as well). Another limiting aspect that prevents wide exploitation, is that in the typical threat model, the adversary should be able to observe or alter the network traffic between the client and the server. Probably, this will not hold back certain state-sponsored agencies or the occasional attacker that manages to get on the same wireless network as the victim. However, if these attacks would no longer require network access, all bets are off. For example, if the only requirement would to be to make the victim run malicious JavaScript, anyone with malicious intents can perform the attack, regardless of geographic location. With HEIST, we introduce a multi-purpose set of attack vectors that show this has in fact become reality: physical network access is no longer a requirement to exploit network-based web attacks.

We show that by combining weaknesses and unexpected behavior in the interaction of mechanisms at various layers, including the browser, HTTP, SSL/TLS and TCP, it is possible to uncover the length of any (cross-origin) response in the browser. More concretely, this means that because SSL/TLS does not hide the length of the clear-text message (a weakness that has been well-known to the security community since 1996 [10]) adversaries can directly infer the length of the response *before* encryption. One of the most well-known, and still highly prevalent attacks that exploits this weakness, is BREACH [6]. With HEIST, we show that BREACH can now be performed by any malicious website, without requiring network observation capabilities. Moreover, we extend the reach of length-exposing attacks, and propose attacks that can extract sensitive information about the victim by exploiting various endpoints.

We evaluate the practicality of HEIST and introduce techniques that can be used to significantly improve its performance. By analysing the consequences of switching to the new, "improved" version of HTTP (HTTP/2), we find that even more attack scenarios can be exploited. Finally, motivated by the pervasiveness and severe consequences of HEIST, we discuss possible defense mechanisms, both on the side of the client as well as on the server.

2 HEIST attack

In this section, we first give a brief introduction to the different mechanisms that contribute to HEIST, and show this can be exploited. Next we show how the basic attack can be further extended to become universally applicable. Finally, we explain the consequences of upgrading to HTTP/2, and introduce attacks that are specific to that protocol.

2.1 The foundations of HEIST

One of the most recent evolutions in the architecture of browsers came with the introduction of Service Workers [8]. The main goal of this set of new APIs is to provide developers with a more flexible way to influence the way requests and responses are handled. With regards to the network layer, the most important interface to look at, is the Fetch API [7]. In terms of functionality, this API has a lot of similarity with `XMLHttpRequest` as both APIs allow the web developer to make arbitrary requests. The main difference is that the Fetch API is built as a starting point for other APIs, such as the Cache API. This means that it should be possible to `fetch()` any resource, including authenticated cross-origin ones. Because of the Same-Origin Policy principle it is obviously not possible to simply read out the response of a cross-origin request.

Another difference between the two APIs, is that the Fetch API works with `Promises` instead of `Events`. This means that when a request is fetched, a `Promise` object is immediately returned. The `Promise` object can then either resolve or fail, depending on the outcome of the `fetch()` process. An interesting feature of the resolution of this `Promise`, is that it happens as soon as the first byte of the `Response` is received. Concretely, this means that after the initial TCP handshake and SSL/TLS negotiation, the browser sends out the `GET` or `POST` request to the server, and waits for a response. As soon as the first byte of the response enters the browser, the web page is notified (by means of resolving the `Promise`), and can start working with the `Response` that is still streaming in. Basically, this works as follows:

```
fetch('https://example.com/foo').then(
    function(response) {
        // first byte of `response` received!
    }
);
```

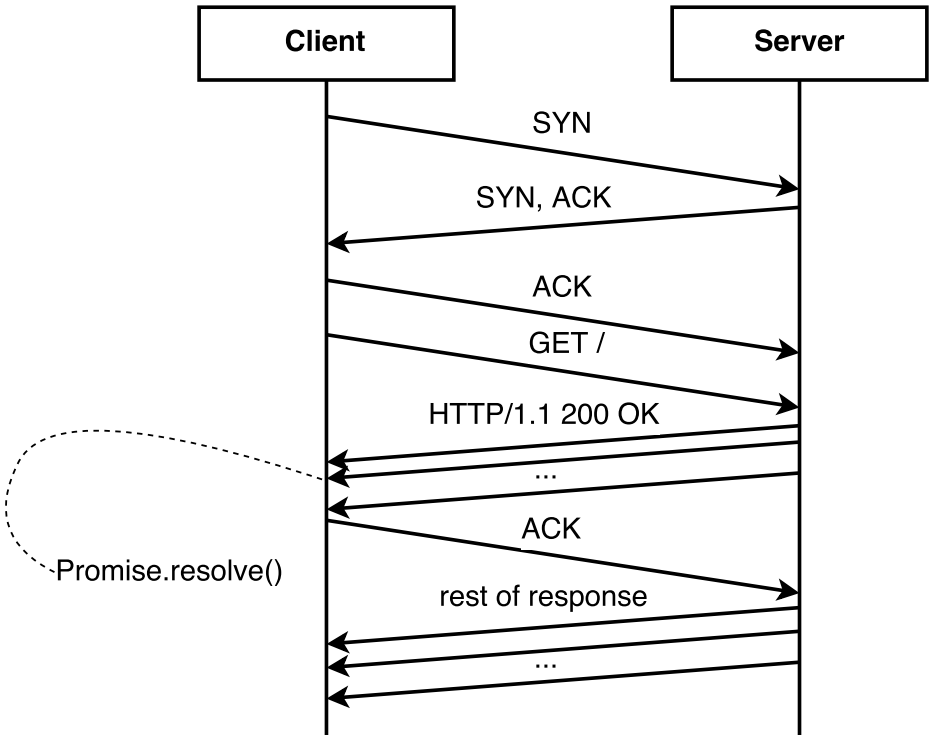

Figure 1: A typical TCP flow for an HTTP request with corresponding response

At first sight, this behavior does not introduce any vulnerability, and in fact improves the browser performance as the browser can start processing the response even before it has been completely downloaded. However, when we take a closer look at the TCP internals [1] and introduce another browser API, things start looking a lot more gloomy. Let's first zoom in on what happens at the TCP level for a typical HTTP request. After the three-way handshake, the client sends a TCP packet containing the request, which typically consists of only a few hundred bytes. As soon as this TCP packet reaches the server, the web server generates a response and sends it back to the client. When this response grows larger than the "maximum segment size" (MSS) the server's kernel will split up the response in multiple segments. These segments will then be sent according to the TCP Slow Start algorithm. In practice, this means that an initial number of TCP segments (predefined by the initial congestion window (`initcwnd`) setting, which is typically set to 10) are sent [4]. For each acknowledged packet, the next congestion window is then increased to allow for a higher bandwidth.

Looking back at what this means for the `Promise` returned by `fetch()`, we can see that in fact the time the `Promise` resolves coincides with the receipt of the initial congestion

```
fetch('https://example.com/foo').then(
    function(response) {
        // first byte of `response` received!
        T1 = performance.now();
    }
);
setInterval(function() {
    var entries = performance.getEntries();
    var lastEntry = entries[entries.length - 1];
    if (lastEntry.name == 'https://example.com/foo') {
       T2minT1 = lastEntry.responseEnd - T1;
    }
}, 1)
```

Figure 2: An example of how to obtain $T_2 - T_1$

window[1] (see Figure 1). This means that if we know when the resource has been completely downloaded, we can in fact find out if the response fits in a single TCP window, or required multiple. For this, we can resort to the Resource Timing API [9], whose purpose is exactly that: providing developers with performance metrics that show when a request was initiated, and when it was complete. Using `performance.getEntries()` we can get the `PerformanceResourceTiming` of the corresponding request, and discover the time the response was completely downloaded by looking at the `responseEnd` attribute. An example of how this would look like in JavaScript code, is provided at Figure 2.

We can make arbitrary requests to any website (at T_0), discover when the first byte (and TCP window) was received (T_1), and find when the response was fully received (T_2). By looking at the time interval between the time the first byte was received, and when the response was completely downloaded, we can find out whether the response took up a single window or required multiple. Figure 3 shows a timeline of an HTTP request and its corresponding response, where the response fits in a single window. The same is shown in Figure 5, but for a request that required two windows. When looking at both timelines, it is clear that when only a single window is used, $T_2 - T_1$ is very small; in practice this is in the range of 1ms. In case a second TCP window is required, $T_2 - T_1$ is increased with an additional round-trip. In practice the value for $T_2 - T_1$ of a "two window" response is significantly higher than for a "single window" response.

[1] This is because the network congestion is several orders of magnitude smaller than the round-trip time

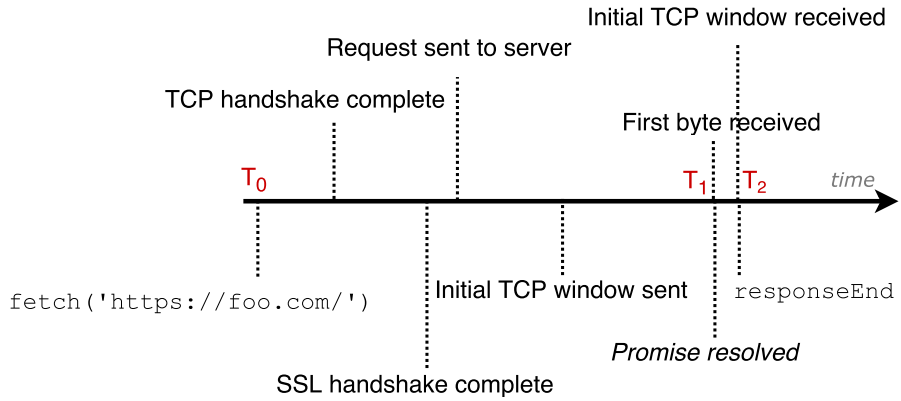

Figure 3: Timeline of an HTTP request whose response fits in the initial TCP congestion window

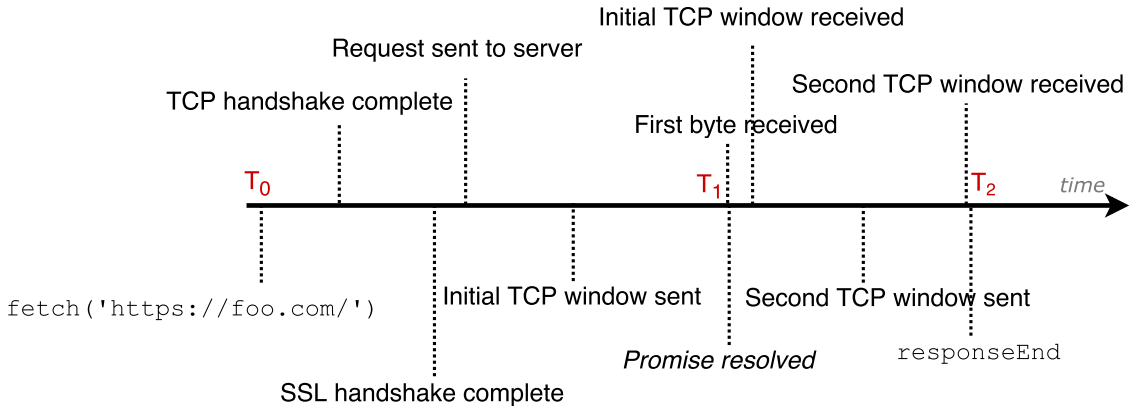

Figure 4: Timeline of an HTTP request whose response requires two TCP windows

In summary, we can discover whether a response is smaller or larger than the initial congestion window. For most installations, which use the default `initcwnd` value of 10 and Maximum Segment Size (MSS) of 1460, this boils down to finding out whether a response is smaller or larger than approximately 14kB. While this could perhaps be used to find out if a user is logged on at a certain website (small error message when not logged in, large response when logged in), it is not that severe. In the following sections, we will show how we can leverage several techniques to drastically increase the impact of HEIST.

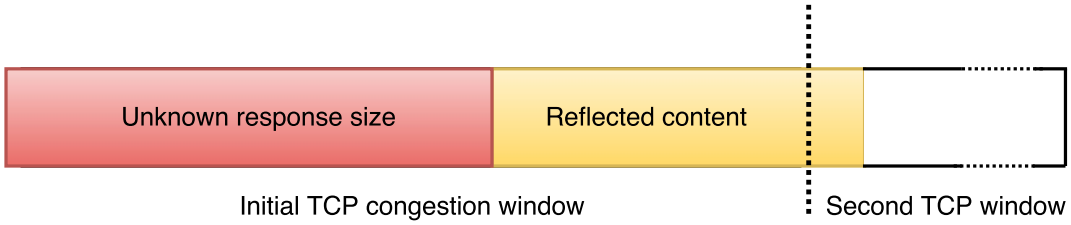

Figure 5: Finding the tipping point to differentiate between one or two TCP windows.

2.2 Determining the exact size of responses

Other than knowing a rough approximation of the length of a response, typically it is much more useful for an attacker to know the exact size of a response. Note that with exact size, we mean the size after `gzip` compression, and after encryption (for stream ciphers, the length remains the same; for block ciphers, the length is rounded up to the next full block). A clear example that shows the severity of knowing the exact length of a response is the BREACH attack. This attack leverages the compression rate of HTTP responses to infer secret data on the page. For this, it needs to know the exact response size after compression. We explore this attack in more detail in Section 3.1. In this section, we give an indication of techniques that can be used to infer the exact size of a response.

In a first technique, we'll show that when a parameter of the request (either `GET` or `POST`) is reflected in the response, we can leverage this to find the exact size. Conceptually, we can split up the response in two parts: a part of which we try to find the exact length, and the parameter that is reflected. The latter is completely controlled by the attacker. To find the exact size of the unknown part, we can repeatedly choose the reflected parameter in such a way that we try to find the largest possible size for which the response still fits in the initial TCP window. We can then compute the size of the unknown part by subtracting the length of the reflected parameter from the size of initial TCP windows (which is a fixed value for each web server). After this, we simply need to subtract the overhead of HTTP and SSL/TLS headers (the length of both are predictable). An example: we find that when the reflected parameter is 708 bytes long, it fits in a single TCP window; for 709 bytes, the total response needs 2 TCP windows. Given an initial TCP window of 14600 bytes ($= 10 \cdot \text{MSS}$), 528 bytes of HTTP response headers, 26 bytes of SSL/TLS overhead; we find the size of the response as $14600 - 529 - 26 - 708 = 13337$ bytes.

As we can divide the search space in 2 with every choice of reflected content length, searching for the exact size requires a logarithmic number of steps. Looking in the range

of 14600 possible values then takes 14 requests. However, we can make two major improvements on this algorithm. First, the browser allows for 6 parallel connections to a single host. If we use these 6 concurrent connections, we can divide the search space in 7 with each iteration. As a result, we reduce the time required to perform the attack (on average, a speedup of 280%) at the cost of making more requests. To further improve the attack, we make the observation that usually the largest part of the response is static or predictable. This allows us to reduce the search space, often to a few hundred bytes. This improvement is also of key importance, as we'll explain in Section 3.1.

Although there are many instances where a parameter is reflected on the web page, there may also be cases where this is not possible. Nevertheless, it may still be possible to resort to case-specific alternatives when there are no reflected parameters. For instance, consider that an adversary is interested in knowing the length of an email of the victim (we explore this example in more detail in Section 3.2). For this, he wants to know the size that is returned by the `POST` request to `https://mail.provider.com/search`, with parameters `from:bank.com AND intitle:Password`. Unfortunately for the attacker, the search query is not reflected in the result set. However, using an attack against a social network website, the adversary managed to obtain the email address of the user. The attacker then sends a large number of emails to the victim[2], each with a different length. Now, instead of using a reflected parameter, the adversary can just include his own emails in the result-set by modifying the search query, and use that to infer the exact size. A search query would then become something like

```
(from:bank.com AND intitle:Password)
  OR intitle:length1203
  OR intitle:length7632
```

This allows the adversary to apply the same technique as with the reflected parameter, by just changing the notion of reflected content as their own email. It should be noted that this is a case-specific technique, and similar (or different) techniques can be applied depending on the target.

2.3 Applying HEIST to larger responses

In the previous sections, we have shown how HEIST can be applied to obtain the exact length for responses that fit in the initial TCP window. While this may be sufficient for

[2]The attacker includes keywords such as "viagra", "cialis" and "Nigerian prince" to make sure the emails end up in the spam folder of the victim, thus raising no suspicion.

certain use-cases, it is definitely not universally applicable. This section will focus on how we can extend the reach of HEIST to being applicable on virtually any website, even with relatively large responses. For this, we go back to the TCP Slow Start mechanism. As mentioned before, this algorithm forces TCP to start with an initial congestion window of a predefined size (default: 10). In order to allow for higher bandwidth, TCP Slow Start provides a mechanism to increase this window. More precisely, for every packet that is acknowledged, the size of the congestion window is incremented by 1.

As such, we can systematically increase the TCP congestion window by first sending a bogus request to a resource of a known size. For instance, if we first request a resource that fits in exactly 4 TCP packets, and then request the resource of which we want to know the size, the server will answer with up to 14 TCP packets in the initial window. To analyse what happens exactly with the congestion window, we created a kernel module that intercepts TCP-related kernel functions and prints out the current state of the TCP connection. In particular, it prints the number of unacknowledged packets ("packets in flight") and the current congestion window (`CWND`).

Figure 6 shows the output of our kernel module. In this specific example, we first send out a request (`Request 1`) for which we know the response size (21 TCP packets). Upon receipt of the request from the client (just before the timestamp 195.719), the server sends back the first part (10 TCP packets) of the response. These packets are then acknowledged by the client, resulting in the growth of the congestion window to 20. Upon receipt of the `ACK`s from the client, the server sends the remaining 11 TCP packets of `Response 1`. As soon as the client completely received the response, it sends off `Request 2`, for which the response spans 15 TCP packets. Because the congestion window had been increased to 20, the server will now send all 15 packets at once.

To sum up: it is possible to increase the congestion window to an arbitrary value (starting from the `initcwnd` value). In an actual attack scenario, the adversary would typically first try to find the lowest number of TCP packets that still fits the response. This can be done in a similar way as finding the "tipping point" of the reflected parameter, meaning it can also be done in a logarithmic number of requests. Once this number has been found, the adversary can fall back to the same methodology to determine the exact size of the response. Interestingly, increasing the congestion window to fit a resource, comes with an additional benefit. By using this method, the length of content that needs to be reflected in order to find the exact size is at most $1 \cdot \text{MSS}$ (1460 bytes). This is particularly useful when a `GET` parameter is reflected in the response, as some servers restrict the maximum allowed length of the URL.

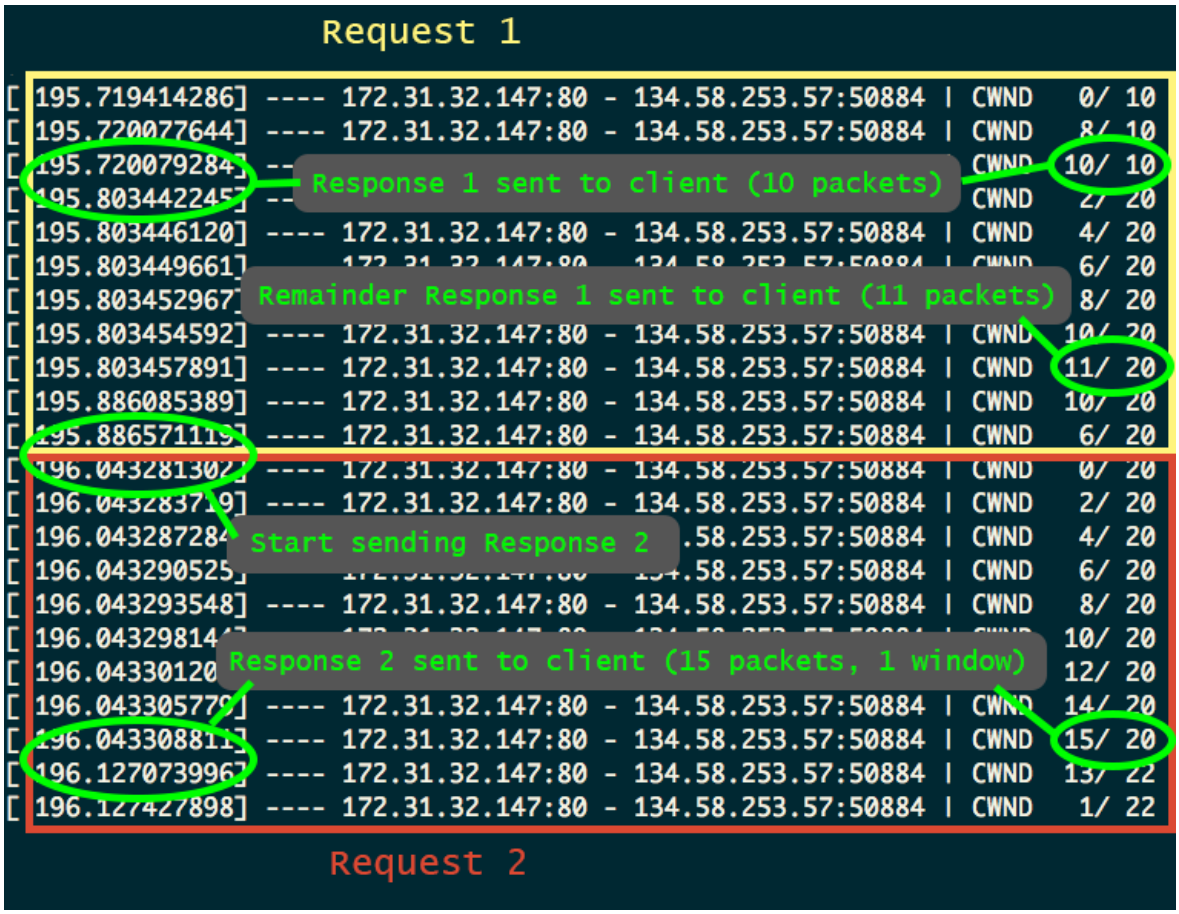

Figure 6: Analysis of the TCP congestion window

Combining all of the above techniques allows us to determine the exact size of any resource, as long as a parameter is reflected in the response (or an alternative approach is possible). In the next section, we will show that in the case of HTTP/2, we can relax these restrictions even further.

2.4 Consequences of HTTP/2

Another recent evolution is the introduction and adoption of HTTP/2. Designed to be a drop-in replacement of the older and common HTTP/1.1 protocol, it is steadily getting more traction [2]. For example, most major browsers now support HTTP/2, and prominent websites such as Twitter, Google, Facebook,... also support HTTP/2. Since we expect that more and more websites will move to HTTP/2, we investigated how it affects our attacks. Surprisingly, we found that not only do our attacks remain possible, we can even increase the damaging effects of our attacks by abusing new features of HTTP/2.

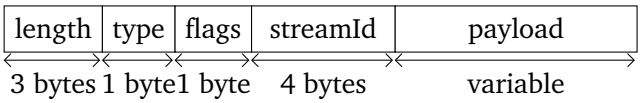

Figure 7: Simplified HTTP/2 frame layout.

2.4.1 A quick introduction to HTTP/2

The goal of HTTP/2 is to provide a more efficient transportation of HTTP messages. The semantics of HTTP messages are not modified. For example, a client still constructs traditional `GET` and `POST` messages, to which the server replies with appropriate responses. This also means that existing websites and JavaScript code continues to work exactly as before. In particular, the service workers and Fetch API that we currently rely on, can still be abused in our attacks. However, on the network level, request and response messages are now encoded and handled in a new manner. This does effect the execution of our attacks.

First, HTTP/2 is designed so a single TCP connection can be used to send and receive parallel requests. This means browsers no longer have to open multiple parallel TCP connections with a server. Instead, all requests are made in a single HTTP/2 connection. Internally HTTP/2 supports parallel requests by creating so-called *streams* for each request and response pair. Naturally, multiple parallel streams can be created in a single HTTP/2 connection. The basic transmission unit in a stream is called a *frame*, whose format in shown in Figure 7. Each frame has a field that identifies the stream it belongs to (called the `streamId` field). Several types of frames exist, with the two most common being `header` and `data` frames. `Header` frames encode and compress HTTP headers using HPACK, and `data` frames contain the body of HTTP messages. Nearly all other frames are used for management purposes, and we call them control frames. One common control frame is the `settings` frame. Endpoints (the client or server) use this to inform the other endpoint about specific HTTP/2 features (or configuration values) that it supports.

An example `GET` request is shown in Figure 8. The client starts by sending a "magic" frame to indicate that HTTP/2 will be used. Then it sends a `settings` frame to inform the server about the maximum number of concurrent streams it supports, the maximum supported frame size, etc. In the same TCP packet the client includes the `GET` request in the form of a `headers` frame. In response, the server first acknowledges the settings of the client by replying with an empty `settings` frame that has the HTTP/2 `ACK` bit set,

Source	Destination	Protocol	Info
10.33.227.220	54.164.68.129	HTTP2	Magic
10.33.227.220	54.164.68.129	HTTP2	SETTINGS, HEADERS
54.164.68.129	10.33.227.220	HTTP2	SETTINGS, SETTINGS
10.33.227.220	54.164.68.129	HTTP2	SETTINGS
54.164.68.129	10.33.227.220	HTTP2	HEADERS, DATA

Figure 8: Packets transmitted during a HTTP/2 request as shown by Wireshark.

and then sends his own `settings` frame to the client. After the client acknowledged the `settings` of the server, the server replies with the HTTP headers of the response, and finally with a `data` frame that contains the body of the HTTP response.

2.4.2 Determining exact response sizes for HTTP/2

The attack of Section 2.2, where reflected content was used to determine the exact size of a resource, can also be performed when HTTP/2 is used. More precisely, an attacker can still abuse reflected content to find out at which point the first TCP window is completely filled, and a second TCP window is required to receive the complete response. The only difference is that HTTP/2 introduces additional overhead, which the attacker must take into account when calculating the size of the targeted resource.

The first type of overhead is caused by control frames sent by the server. For example, in the example HTTP/2 request of Figure 8, there are in total two `settings` frame sent by the server. No other control frames are present. We found that against both nginx and Apache, the number of control frames, as well as their length, can be predicted. And since we know how many frames are sent in total, we can also calculate the amount of overhead that is introduced by the 9-byte header of in every HTTP/2 frame (see Figure 7).

All that is left, is to predict the length of the `headers` frame sent by the server. This frame contains the HTTP headers of the response, compressed using HPACK. The first time a header field (or value) is transmitted, HPACK transfers the original value of the field, and assigns an index to this particular value. Whenever this same field or value is again transmitted in the same HTTP/2 connection, only the assigned index is transferred instead of the complete value. An example of this is shown in Figure 9. Here both the `server` and `x-powered-by` header were already transferred once in a previous `header` frame, meaning they can now be encoded using just a single-byte index. Interestingly,

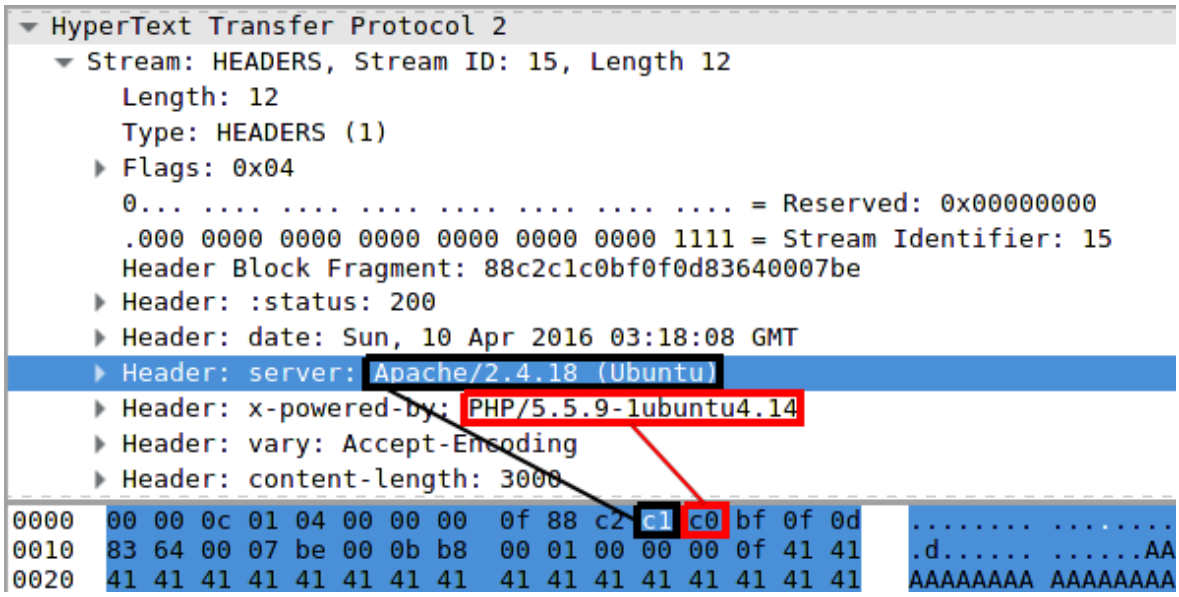

Figure 9: Example where HPACK compresses header values to a single-byte index.

this means HPACK makes it *easier* to predict the length of the `header` frame. Indeed, we no longer have to predict the length of all header fields and values, since HPACK compression replaces them by a 1-byte index. This is rather surprising: while HPACK was designed to make length-based attacks such as CRIME harder, it makes it easier to predict the length of the `headers` frame itself, which in turn makes our attacks easier.

For example, say we find that when the reflected parameter is 1141 bytes long, it fits in a single TCP window. But when it's 1142 bytes, the total response needs 2 TCP windows. Given an initial TCP window of 14600 bytes ($= 10 \cdot \text{MSS}$), we subtract $0 + 6$ for the body of the two HTTP/2 `settings` frames, $10 \cdot 9$ for the ten HTTP/2 frame headers, and finally 26 bytes of SSL/TLS overhead. This means the size of the targeted resource is $14600 - 6 - 90 - 26 - 1141 = 13337$ bytes.

2.4.3 Abusing reflective content in another resource

In our original attack of Section 2.2, we had to rely on reflected content in the targeted resource itself. When HTTP/2 is used, we can relax this condition by abusing reflected content in *another* resource on the same website. Recall that reflective content was used fill the TCP window until the complete response no longer fitted into the first TCP window.

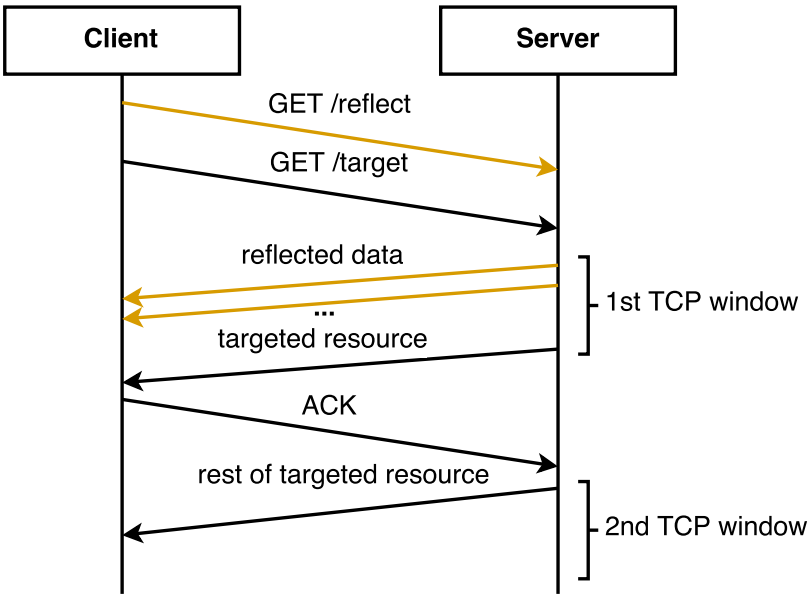

Figure 10: Using reflected data in a parallel HTTP/2 request to fill the first TCP window.

If we know that HTTP/2 is used, we can let the browser simultaneously request the targeted resource, and another resource that contains reflected content. Since HTTP/2 is used, both requests are sent in parallel to the server, and the server replies to them in parallel as well. This is illustrated in Figure 10. Here both requests and responses are sent simultaneously in the same TCP connection, and hence share the same TCP window[3].

Notice that in Figure 10, the response to the resource containing reflected data (the request to `/reflect`) fills in a large part of the initial TCP congestion window. The response to the targeted resource uses the remaining space of the window. This allows us to abuse the `/reflect` resource to fill up the initial TCP congestion window, until there is no longer sufficient space to also include the targeted resource in the window. By using HEIST to measure when a second TCP congestion window is needed, we can again determine the size of the targeted resource. To sum up, when HTTP/2 is used, the only requirement to perform our attack is to find reflected content in *any* page of the website.

[3]Note that if we would initiate two requests at the same time in HTTP/1.1, the two requests would be handled using two different TCP connections, whom each would have their own TCP windows.

3 Attack scenarios

3.1 BREACH

We demonstrate that well-known compression-based attacks such as CRIME or BREACH (but also lesser-known ones [5]) can be executed by merely running JavaScript code in the victim's browser. This is possible because HEIST allows us to determine the length of a response, without having to observe traffic at the network level. In particular we focus on BREACH, since many websites are (still) vulnerable to this attack.

The BREACH attack abuses gzip compression of HTTP responses to recover secret tokens in these responses. To carry out the original BREACH attack, an attacker must be able to [6]:

1. Inject chosen plaintext into the server's response.
2. Observe network traffic to measure the size of encrypted responses.

The idea behind the attack is that if the attacker injects a prefix of the secret token, gzip will detect this repetition, and will therefore more efficiently compresses the response. It was demonstrated this can be used to decrypt, among other things, CSRF tokens. For example, in a demonstration of the attack, they exploited the observation that Microsoft's Outlook Web Access reflects parameters of the URL in the response. So a request to

```
https://site.com/owa/?ae=Item&t=IPM.Note&a=New&id=canary=<guess>
```

would result in

```
<span id=requestUrl>https://site.com:443/owa/forms/
        basic/BasicEditMessage.aspx?ae=Item&amp;t=IPM.Note&
        amp;a=New&amp;id=canary=<guess></span>
        ...
<td nowrap id="tdErrLgf"><a href="logoff.owa?
    canary=d634cda866f14c73ac135ae858c0d894">
        Log Off</a></td>
```

Here `canary` contains the CSRF token used by the web service. If the first character(s) of `guess` match that of the CSRF canary, gzip can effectively remove the second appearance of the substring, resulting in a shorter compressed response. In other words, if our guess for the next character of the token is correct, the resulting compressed response is shorter.

Using HEIST it becomes much easier to meet the second requirement: we can measure the size of the encrypted response in the browser, instead of needing a man-in-the-middle position to observe network traffic. In other words, the BREACH attack against Microsoft Outlook Web Access can be executed purely in the browser using (malicious) JavaScript. The attack can also be applied against other sensitive information that is present on web pages, such as email addresses, usernames, or other (static) personally identifying information.

A few technical difficulties arise when attempting to perform the attack in practice[4]. We will briefly discuss the most important ones, other technicalities were already discussed in the original BREACH paper [6].

Huffman Encoding. Gzip compression is based on the DEFLATE algorithm, which uses a combination of LZ77 and Huffman encoding [3]. It's the LZ77 component that searches for duplicate substrings, and hence is the component that makes BREACH possible. For example, LZ77 would compress the input

```
Secret csrf=d634cda866f14 and reflected=csrf=d634cd<padding>
```

by replacing the second occurrence of `csrf`, together with the other matching characters in the reflected input, with a reference to the first occurrence:

```
Secret csrf=d634cda866f14 and reflected=@(11,33)<padding>
```

Here `@(11,33)` is a reference to the 11-character string that started 33 characters before the current position. Clearly, if we correctly guess the next character, the response will further shrink in size. The problem is that Huffman encoding could *also* cause the response to shrink in size. Recall that the goal of Huffman encoding is to more efficiently compress common symbols (bytes). Essentially, frequent byte values are encoded using

[4]Naturally we restricted ourselves to attacks against our own accounts to avoid harming innocent servers and users.

a short bitstring, while infrequent byte values are encoded using a larger bitstring. For example, the letter `e` occurs a lot in English texts, and is likely represented using a short bitstring. So if our next guess would be `d634cde`, the compression could be shorter simply because `e` is represented using a shorter bitstring after Huffman encoding (and not because it is a correct guess). We can solve this problem by issuing two requests for each guess. In our example we would try both `d634cde|` and `d634cd|e`. Then there are two possibilities:

1. If the guess is incorrect, both `e` and `|` are represented using a Huffman code. This means both guesses will result in compressed responses of the same length.
2. If the guess is correct, then the `e` in `d634cde|` is compressed with LZ77 using a reference, and only the character `|` is represented using Huffman encoding. However, in `d634cd|e`, both characters are compressed using a Huffman code. Therefore a correct guess means the compressed length of `d634cde|` and `d634cd|e` differ.

To summarize, we have a correct guess if `d634cde|` and `d634cd|e` result in a different compression length.

Maximum reference distance. In DEFLATE and gzip, the LZ77 component can insert a reference to a string that occurred at most $32 \cdot 1024$ bytes before the current position [3]. This means that in order to execute the (original) BREACH attack, the reflected input must be close enough to the targeted token.

Block Ciphers The attack requires that we are able to detect a 1-byte difference in the length of the (compressed) responses. Even when block ciphers are used, the following technique can be used to reliably detect this difference. The idea is to first pad the response using reflected content so that, if the response would be one byte shorter, the last block is no longer needed. This boundary point can be detected using HEIST by sending at most $\log_2(16) = 4$ requests, assuming the cipher operates on blocks of 16 bytes. If a guess now results in a response that is one byte shorter, the last block is no longer needed, and hence can be detected.

3.2 Web-based length exposure

In this section, we show that next to extracting secret tokens from a web page by using CRIME or BREACH attacks, length-exposing vulnerabilities can also be exploited in

different ways. To give some insight into the potential attack scenarios, we present two examples. In the first example, we will show how a website's search functionality can be abused by an attacker to steal sensitive content such a credit card information, passwords,... For the second example, we focus on personal health websites, and demonstrate how adversaries could apply HEIST to obtain our private health information.

For this section, there are two important things to note. First, we only present two different attack scenarios. By far this is not the definitive set of possible attack scenarios. The type of attack that can be performed on a website is typically based on the functionality it offers. We made a selection of attack scenarios to include based on their severity and prevalence in popular web services. A second thing to note is that the examples presented here are fictitious. The reason behind this decision is that the attacks presented here are generally applicable and do not require a specific implementation method by the web server. We managed to successfully perform very similar attacks against multiple of highly popular (billions of users) web services[5]. The attacks reflect the general functionalities provided by these services, and thus the consequences are frighteningly close to reality.

3.2.1 Search-based information disclosure

The running example we use to illustrate this attack scenario, is based on a web-based mail provider named SnailMail who can be reached at `https://snailmail.org`. Just as any self-respecting mail provider is supposed to do, SnailMail makes sure their customers are well protected: all traffic is only sent over a secure TLS connection using the AES-GCM ciphersuite, HTTP Strict Transport Security and HTTP Public Key Pinning are enabled, and compression (both on SSL/TLS as HTTP level) is disabled to prevent BREACH and CRIME attacks. Of course, SnailMail also took the necessary precautions to completely prevent web attacks, thereby making it virtually impenetrable.

Just as any other web-based mail provider, SnailMail allows users to search through the emails that they have sent or received. Technically, this done with a `POST` request to the `/search` endpoint. This endpoint will return the results for the query (provided in the `q` parameter) in JSON format. This JSON data contains the original query[6] and metadata (sender, subject, date, first line of message, ...) for each message that matches they query.

[5]Of course, we performed these attacks against our own accounts, and neither humans, animals nor servers were harmed during our experiments.

[6]This is not a requirement when either HTTP/2 is used, or the attacker uses a technique as discussed in Section 2.2.

Alice, the victim in this example, is a frequent user of SnailMail, and entrusts them with storing her most important emails, including emails from her bank. Unfortunately, Alice's bank is notorious for not being too concerned with their customer's security[7]. For instance, they are known to send unmasked credit card information to their customers over email. Due to the security precautions taken by SnailMail, Alice has not experienced any nefarious consequences because of this. However, one day Alice decides to look for funny cat images, and ends up on a website (`catpicz.com`) that is under the control of the attacker. While browsing the innocuous-looking website, and laughing at the hilarious cat pictures, Alice's credit card information is being stolen.

By looking at the length of responses for different search queries on Alice's mailbox, the attacker is able to reconstruct the numbers on the credit card piece by piece. More concretely, the queries that the attacker makes Alice send, look as follows:

```
(from:bank.com AND intitle:"MasterCard") AND
  (5100 OR 5101 OR 5102 OR ...) OR
  (bogus_data_that_is_reflected_to_reach_initcwnd)
```

The JSON responses that contain results then look as follows:

```
{
  "query": "(from:bank.com AND intitle:\"MasterCard\") AND
          (5100 OR 5101 OR 5102 OR ...) OR
          (bogus_data_that_is_reflected_to_reach_initcwnd)",
  "results": [{
          "sender": "info@bank.com",
          "subject": "Your MasterCard information",
          ...
  }]
}
```

The attacker makes sure that when the response from SnailMail contains no results, the complete response fits in the initial congestion window. When the attacker then sees a response that requires two TCP windows, he knows that the correct number is part of the set he tried. The attacker then keeps on decreasing the search space for possible

[7] Alternatively, Alice may have emailed the credit card information to a friend, or even herself.

numbers until an exact match has been found. When using only a single connection, the complete credit card information (including CVV) can be retrieve with just 61 requests. Given that the round-trip time between Alice and SnailMail is approximately 50ms, the attack would only take 4.6 seconds. When making use of 6 concurrent connections, the attack may take up to 138 requests, but can be executed in 1.2 seconds.

3.2.2 Revealing personal user state

When visiting a website, users are typically provided a personal view that is customized based on the information they shared with the website. For instance, when looking at the home page of a social network website, users are presented with a stream of updates of their peers. In this section, we will show that by analysing the length of specific requests to websites, it is possible for attackers to infer the information the victim shared with the website. Again, we will explain this attack scenario by the means of a fictitious website. The fictitious company (PatientWeb) now involves a personal health website that allows users to track their health records and receive information about their disease. Again, the website is armed to the teeth to thwart attacks on the privacy of their users. The developers of PatientWeb are really concerned about the performance and security of their website. As a result, they enabled HTTP/2 and made sure BREACH could not be exploited by preventing data originating from the request to be reflected on a web page containing sensitive information.

The adversary in this example is a malicious party who sells sensitive health information to insurance companies. Unfortunately, the attacker does not care about the privacy concerns of the people they sell information on. As such, the attacker will even resort to web attacks to obtain personal information, and PatientWeb is definitely a gold mine for that. To gather information, the attacker first sets up a website where users may win coupons for a number of grocery stores, and drives users to this website by sending spam, and sharing it on social media.

When visiting the malicious website, the attacker first verifies which websites the user is logged in to, and launches specific attacks per target. For PatientWeb, the adversary will apply HEIST in order to determine from which condition the user suffers. In a preparatory phase, the attacker registered a large number of accounts at PatientWeb, and selected a different condition for each account. At regular intervals, the size (in bytes) of the news feed for each condition is collected.

To determine from which condition someone suffers, all the attacker needs to do is determine the size of the news feed that is returned for the victim. This will coincide with the size of a particular news feed that was collected in the preparatory phase, allowing the attacker quickly determine the exact condition. In the attack, the adversary will need to resort to the attack technique that is specific to HTTP/2 as there is no content of the request reflected in the response. To expose the exact response size, the attacker will either try to find a different endpoint for which content is reflected in the response, or combine existing resources (images, scripts, stylesheets, ...) to obtain a similar result. Even with an extensive list of possible conditions, the attacker is able to pinpoint the victim's condition in less than a second. An overview of the attack scenario can be found in Figure 11.

4 Countermeasures

The attack techniques behind HEIST span over multiple layers (browser, HTTP, SSL/TLS, TCP). In this section, we provide an overview of possible defense mechanisms that can be used to thwart attacks, or make it harder to exploit. We also discuss techniques that look promising but in fact fail to properly prevent attacks, or can be circumvented. The title of each countermeasure indicates the level of the completeness, ranging from infeasible to complete.

4.1 Browser layer

4.1.1 Prevent browser side-channel leak *(infeasible)*

Probably the most obvious countermeasure against HEIST is to prevent an attacker from finding out when the first byte of a response has been received. This could be achieved by for instance resolving the `Promise` only when the complete response has been received. However, we argue that this method is insufficient to properly prevent attacks leveraging HEIST. The main reason for this, is that there are in fact multiple techniques that can be used to infer the moment the initial congestion window has been received. For instance, the presence or absence of certain response headers cause browsers to expose specific behavior. These actions occur as soon as response headers have been received (this coincides with receiving the first congestion window), and can be observed by the

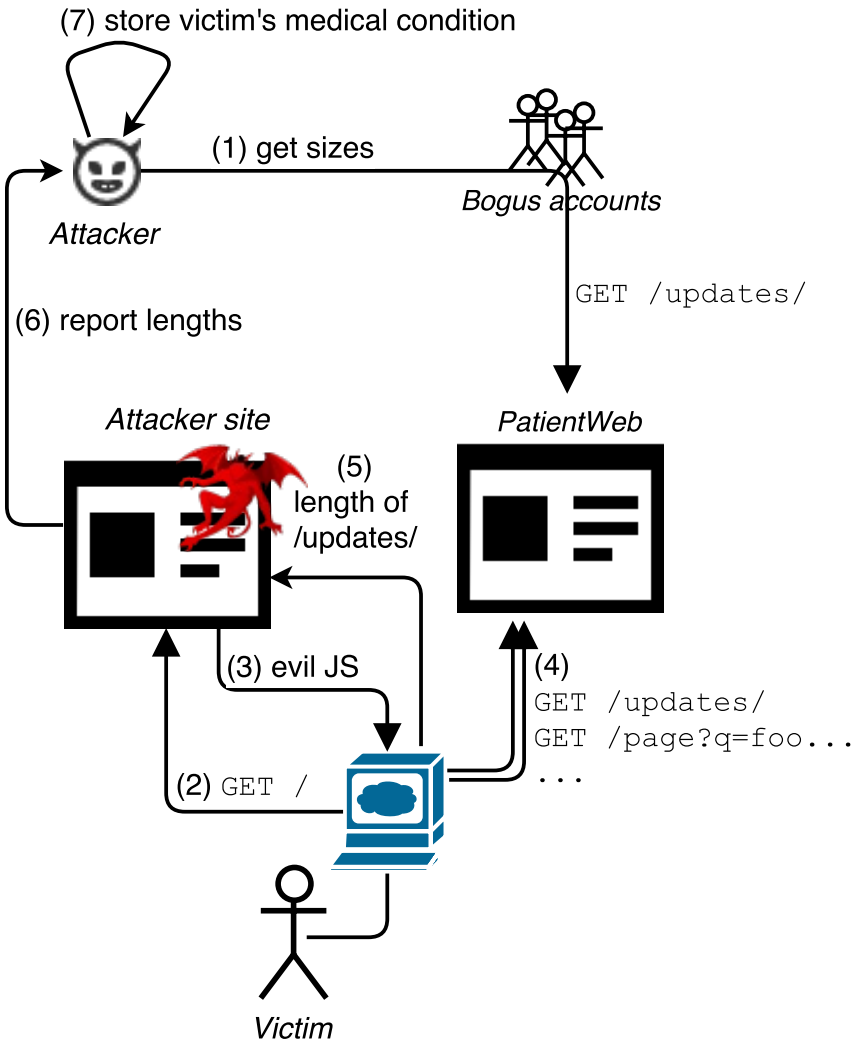

Figure 11: An overview of the attack scenario described in Section 3.2.2.

adversary. Although it may be possible to prevent all side-channel leaks, we consider this an unlikely event: the general browser architecture needs to be significantly altered, and any new feature may introduce a new side-channel.

4.1.2 Disable 3rd-party cookies *(complete)*

When the attacker makes the victim initiate requests to the target website, a user-specific response is returned. This is because the victim's cookies are included in the request, which means that from the website's perspective the requests are part of the victim's browsing session. By preventing these cookies from being included, the request would

be unauthenticated and no user-specific content will be returned, and thus can't be stolen. Obviously, it is not possible to block all cookies (as this would prevent us from logging in to any website), however it is possible to disable 3rd-party cookies. Concretely, this means that when navigating to `https://attacker.com`, cookies would only be included in requests (e.g. for images, scripts, ...) to the same origin.

Currently, most major browser vendors provide a way to disable 3rd-party cookies, but may prevent features of certain websites from working correctly. Nevertheless, we consider this approach one of the best ways to prevent possible attacks.

4.2 HTTP layer

4.2.1 Blocking illicit requests *(inadequate)*

Similar to the defense of blocking 3rd-party cookies, the web server could also block requests that are not legitimate. A popular way of doing this, is by analysing the `Origin` and/or `Referer` request headers. This allows the web server to determine whether the request originated from its own domain or a potentially malicious one. However, it is still possible to make requests without these two headers, preventing the web server to determine where to request originated from. As a result, this technique can not be used to prevent attacks.

4.2.2 Disable compression *(incomplete)*

HTTP responses are often served with compression (either at SSL level, or using gzip) to reduce the required bandwidth. It is a well-known fact that this leads to compression-based attack such as CRIME (SSL compression) and BREACH (gzip). Preventing these attack from being exploited in the browser can be done in the same way as preventing the generic attack, namely by disabling compression. Unfortunately, this does not prevent any of the other length-based attacks discussed in this report.

4.3 Network layer

4.3.1 Randomize TCP congestion window *(inadequate)*

The attack vectors included in HEIST uncover the length of responses by inspecting whether a response fits in a single TCP congestion window, or requires multiple. By randomizing the number of TCP packets included in a TCP congestion window for each connection (e.g. by selecting a random value for `initcwnd`, and growing the congestion window in an unpredictable manner), the attacker will not uncover the reason why a response required multiple congestion windows. Either the TCP congestion window was small, or the response was in fact large. However, the attacker can still try to obtain the length of a certain response multiple times. As the size of the TCP congestion window is relatively small, it's not possible to introduce a lot of variation. Although this countermeasure requires that the adversary needs to initiate more requests, an attack can typically be performed in just a few seconds. Even a tenfold increase in the number of requests would still leave attacks very practical. Therefore, we do not consider this a valid defense strategy.

4.3.2 Apply random padding *(inadequate)*

The idea behind this approach is similar to randomizing the TCP congestion window: prevent the attacker from directly learning the exact length of a response by masking it with a random value. As such, it also suffers from the same vulnerability, namely that the attacker just needs to make more requests to filter out the randomness. Another downside of padding is that the response can only become larger, and thus will have an impact on the bandwidth. In contrast to applying randomization on the congestion window size, adding random padding to the response works slightly better with regards to preventing the attacker from learning the exact response size. This is because the range of possible values can be much higher. However, in case the adversary does not need to know the exact size of a response, but just the range it is in, the number of additional requests that need to be made may be negligible. Hence, we do not consider this countermeasure appropriate in defending against HEIST-based attacks.

5 Conclusion

We have shown that it is possible to execute attacks on network protocols such as SSL/TLS purely by running (malicious) JavaScript inside the victim's browser. In particular, HEIST can be used to determine the exact size of HTTP responses, by increasing the amount of reflected content until the HTTP response no longer fits into the initial TCP congestion window. We have also shown that the introduction of HTTP/2 further worsens the situation, as it enables even more attack techniques. Therefore, we expect these types of attacks to be more prevalent as the adoption of HTTP/2 increases.

The impact of our findings have been illustrated by showing how compression-based attacks such as BREACH can be executed without requiring a man-in-the-middle position. Additionally, by using search-based oracles, personal information such as credit card numbers can be obtained, and the medical conditions of victims can be exposed. Finally, we have argued that it is difficult to defend against our attacks. One of the few, if not the only, adequate countermeasure is to disable third-party cookies.

References

[1] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, September 2009.

[2] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.

[3] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, 1996.

[4] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing TCP's initial congestion window. *Computer Communication Review*, 40(3):26–33, 2010.

[5] John Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption*, pages 263–276. Springer, 2002.

[6] Angelo Prado, Neal Harris, and Yoel Gluck. SSL, Gone in 30 seconds. *BREACH Attack*, 2013.

[7] Anne Van Kesteren and WHATWG. Fetch Standard. `https://fetch.spec.whatwg.org/`, January 2015.

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

[9] W3C. Resource Timing Standard. `https://www.w3.org/TR/resource-timing/`, February 2016.

[10] David Wagner, Bruce Schneier, et al. Analysis of the SSL 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 29–40, 1996.