BREACH: REVIVING THE CRIME ATTACK

YOEL GLUCK, NEAL HARRIS, AND ANGELO (ÁNGEL) PRADO

Contents

1. Introduction

At ekoparty 2012, Thai Duong and Juliano Rizzo announced CRIME, a compression side-channel attack against HTTPS traffic [4]. An attacker with the ability to

- inject partial chosen plaintext into a victim's HTTP requests
- measure the size of encrypted traffic

Date: July 12, 2013.

yoel.gluck2@gmail.com, neal.harris@gmail.com, angelpm@gmail.com.

B.R.E.A.C.H.: Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext.

can leverage information leaked by compression to recover targeted parts of the plaintext.

Compression side-channel attacks are not new. Indeed, ten years before Duong and Rizzo's presentation, a paper appeared describing such attacks in general [5]. However, CRIME gave a concrete, real-world example of such a vulnerability. Subsequently, at Black Hat Europe 2013, Be'ery and Shulman announced TIME [2], an attack against compressed HTTPS responses. Their work uses timing – rather than ciphertext length – to harvest the information leaked by compression.

The particular demonstration given at ekoparty showed how an attacker might execute this attack to recover the headers of an HTTP request. Since HTTP headers contain cookies, and cookies are the primary vehicle for web application authentication (after login), this presents a significant attack. However, the attack also relied on a relatively little-used feature of TLS: compression. By disabling TLS/SSL-level compression – which was already little-used, and in fact disabled in most browsers – the attack as demonstrated at ekoparty is completely mitigated.

After the conference, the discussion around Duong and Rizzo's work essentially concluded that disabling compression at the TLS level completely solved the problem. From [1]:

> As of September 2012, the CRIME exploit has been mitigated by the latest versions of the Chrome and Firefox web browsers, and Microsoft has confirmed that their Internet Explorer browser wa s not vulnerable to the exploit.

Unfortunately, this is simply not true.

Our work shows that TLS-protected traffic remains vulnerable to realistic compression side-channel attacks. Several major web applications, including Microsoft's Outlook Web Access, can be attacked to recover secrets previously believed to be protected by TLS.

We achieve this by attacking HTTP *responses* instead of HTTP requests. Even if TLS-level compression is disabled, it is very common to use gzip at the HTTP level. Furthermore, it is very common that secrets (such as CSRF tokens) and user-input are included in the same HTTP response, and therefore (very likely) in the same compression context. This allows essentially the same attack demonstrated by Duong and Rizzo, but without relying on TLS-level compression. We remark that Duong and Rizzo anticipate this problem in their original presentation. See slides 38-40 of [4]. Furthermore, we also remind the reader that TIME is a timing-based compression side-channel attack against HTTPS responses. See [2] for details.

The paper is organized as follows: we begin with an overview of the attack, including a description of the necessary ingredients, as well as non-requirements. We then describe some technical roadblocks, and our methods for overcoming them. Next, we discuss some potential mitigations. Finally, we close with some remarks on future research.

2. The Attack

2.1. **Overview.** Many web applications routinely deliver secrets – such as CSRF tokens – in the bodies of HTTP responses. It is also common for web applications to reflect user input – such as URL parameters – in HTTP response bodies. Since DEFLATE [3] (the basis for gzip) takes advantage of repeated strings to shrink the compressed payload, an attacker can use the the reflected URL parameter to guess the secret one character at a time.

Consider Microsoft's Outlook Web Access. If a user with an active session makes the following request

```
GET /owa/?ae=Item&t=IPM.Note&a=New&id=canary=<guess>
```

she receives the following in the HTTP response body:

```
<span id=requestUrl>https://malbot.net: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>
```

In this case, `canary` is a CSRF token.

Note that the string `canary=` is repeated in the response body. DEFLATE can take advantage of this repetition; in fact, if the first character of the guess matches the first character of the actual value of `canary` (`d` in this case), then DEFLATE will compress the body even further. The upshot is that fewer bytes go over the wire when the guess is correct. This provides an oracle that an attacker can exploit to recover the first character of `canary`. Then, the attacker proceeds in the same manner to recover `canary` byte-by-byte.

In the case of OWA, we are able to reliably ($\approx$ 95% of the time) recover the entire CSRF token, and often in under 30 seconds.

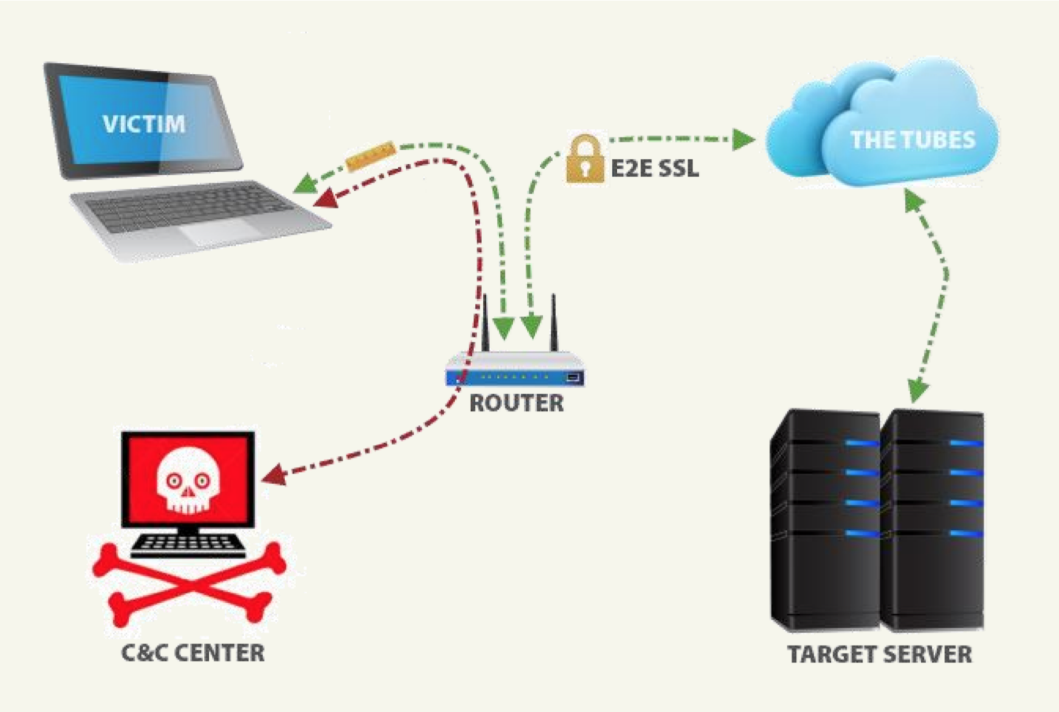
2.2. **Ingredients, and non-Ingredients.** In order for the attack to be successful, several things are required. To be vulnerable to this side-channel, a web app must:

- Be served from a server that uses HTTP-level compression
- Reflect user-input in HTTP response bodies
- Reflect a secret (such as a CSRF token) in HTTP response bodies

Additionally, while not strictly a requirement, the attack is helped greatly by responses that remain mostly the same modulo the attacker's guess. This is because the difference in size of the responses measured by the attacker can be quite small. Any noise in the side-channel makes the attack more difficult (though not impossible).

We remark that the attack is agnostic to the version of TLS/SSL, and does not require TLS-layer compression. Additionally, the attack can work against any cipher suite. If the victim uses stream ciphers, the attack is simpler; the difference in sizes across response bodies is much more granular in this case. If a block cipher is used, additional work must be done to align the blocks in the response.

2.3. **The Setup.** We assume the attacker has the ability to view the victim's encrypted traffic. An attacker might accomplish this with ARP spoofing. We also assume the attacker has the ability to cause the victim to send HTTP requests to the vulnerable web server. This can be accomplished by coercing the victim to visit an attacker-controlled site (which will contain invisible `iframe`'s pointing to the vulnerable server), or by intercepting and modifying *any* non-secured traffic.



We note that this is an active, online attack. The attack proceeds byte-by-byte; the attacker will coerce the victim to send a small number of requests to guess the first byte of the target secret. The attacker then measures the size of the resulting HTTP responses. With that information, the oracle determines the correct value for the first character of the secret. Because the attack relies on LZ77, the first character of the target secret must be correctly guessed before the second character can be attempted.

2.4. **Technical Challenges.** Despite the simple, straightforward nature of the vulnerability, actually exploiting it can be a bit difficult. In this section, we describe some challenges that we faced. Our solutions to the challenges are likely non-optimal, and are an excellent opportunity for further research.

2.4.1. *Huffman Coding and the Two-Tries Method.* DEFLATE has two main components: LZ77 and Huffman coding. One of the first difficulties that one is sure to face when trying to exploit this kind of compression side-channel is due to Huffman coding.

Huffman coding is a variable length prefix code. Each source symbol is represented by a code. The mapping is chosen such that source symbols appearing more frequently in the source are encoded by shorter codes. For a more complete description, we encourage the interested reader to consult [3].

Why is this a problem for our attack? Note that our attack takes advantage of information leaked by the *other* component of DEFLATE: LZ77. The naïve approach to making the attack work goes something like: when the attacker guesses the next character correctly, the resulting plaintext will have a longer repeated string, and will therefore compress more. Unfortunately for the attacker, this is not always true. If an attacker guesses a character that is incorrect, but happens to be a very heavy symbol, then it's possible that Huffman coding will cause the compressed data to be as small or smaller than the compressed data for the correct guess.

Solving this is straightforward. Duong and Rizzo dealt with this problem with what they call the 'Two-Tries Method'. To understand their solution, we first make the problem explicit. What we'd like to be true is the following: smaller compressed data implies longer repeated substrings. But Huffman coding will treat each guess slightly differently, since the relative frequencies of each symbol matter. Our oracle is being polluted by the effects of Huffman coding; we need to isolate the effects of repeated substrings.

So, instead of sending a single request for each guess, we send two. Using the example of OWA described above, one request will take the form

```
GET /owa/?ae=Item&t=IPM.Note&a=New&id=
    canary=<guess><padding>
```

while the other is of the form

```
GET /owa/?ae=Item&t=IPM.Note&a=New&id=
    canary=<padding><guess>.
```

The padding should be a collection of symbols taken from the complement of the alphabet for the target secret. For example, if the secret is known to be hex, then a candidate for padding is '{}'. The important thing to note here is that the Huffman tree should be very similar (if not identical) between the two responses.

The resulting oracle works as follows: if the guess is incorrect, then both guesses should elicit responses with repeated substrings of the same length, which should result in compressed data of the same size. If the guess is correct, then the first request described above will have a longer repeated substring, which the LZ77 part of DEFLATE can take advantage of to produce slightly smaller compressed data.

It is important to note that we are not comparing sizes across guesses. To make this concrete, consider the following four requests, along with the sizes of their corresponding responses:

Request	Response Size
`canary=a{}`	99
`canary={}a`	100
`canary=b{}`	98
`canary={}b`	98

In this case, our oracle would tell us that `a` is the correct guess, since the attempt with the `a` adjacent to the known repeated substring resulted in a smaller response. The fact the responses for `b` are smaller – but both the same length – indicates that `b` is likely a more common symbol than `a` in the response, but not the correct guess. Otherwise, LZ77 would have been able to take advantage of a longer repeated substring (in addition to the fact that `b` is a more common symbol).

2.4.2. *Guess Swap.* A variation on the Two-Tries method, the Guess-Swap method helps mitigate oracle confusion due to Huffman coding. Suppose the oracle has determined that the first three characters of the target secret are `4bf`. Then, to determine the next character, we'd submit two requests per guess, with payloads as follows:

`canary=4bf`**`d`**

and

`canary=4b`**`d`**`f`

Now, we can attach a score to the guess for `d`: the difference in size between the second response and the first. Note that if `d` is the correct value for the next character in the secret, then the second response should be larger than the first. (So a larger score means the character is more likely to be correct.) In this variation, we send a pair of requests for each candidate character; the character with the highest score is the winner, and the oracle takes that to be the next character in the secret.

2.4.3. *Charset Pools.* By adding all candidate characters to each request, we attempt to keep the effects due to Huffman coding constant across guesses.

The technique works as follows: suppose that the secret is a hex string. Then guesses would include payloads such as:

`canary=4bf`**`d`**`{}-a-b-c-e-f-0-1-2-3-4-5-6-7-8-9`

In the examples above, `4bf` is the part of the secret which has already been guessed by the oracle, and **`d`** is in the 'guess' position. With this method, we will send sixteen requests to the server; each will have one of the candidate characters in the guess position. Then, the smallest response is deemed to correspond to the correct guess. In this case, we *do* compare the sizes of responses across guesses, in contrast to the Two Tries method. Also, note that this technique only requires one request for each guessed character, instead of two as with the Two Tries method.

We call attention to the delimiters (-) in the payload. These are there to reduce the chances of some part of the pool matching with some other part of the response. Without these, the guess above would contain the payload

`canary=4bf`**`d`**`{}abcef0123456789`

while another guess would contain the payload

`canary=4bf`**`e`**`{}abcdf0123456789`

If the substring `bce` appears elsewhere in the document, then note that the first of these two would likely compress more than the second. This difference would confuse the oracle, and might lead it to report that `d` was the correct next character even if it isn't. By choosing a delimiter that is unlikely to appear in the rest of the response body, we can reduce the chances of this happening.

2.4.4. *Compression Ratio for Guesses.* While not particularly common, occasionally patterns in the target secret can confuse the oracle due to compression *within* a guess. For example, suppose the target secret is `ABCAB123`. Further, suppose the oracle correctly guesses the first five characters of the secret: `ABCAB`. Then, when trying to guess the sixth character, it will likely return `C` and `1` as correct. This is because `ABCABC` alone can be compressed by LZ77, while `ABCAB1` will match with the actual secret, and also be compressed by LZ77.

To detect and mitigate this, we look for patterns in our guess that are not likely to happen in random secrets. A simple way to accomplish this is to see how well our guess compresses *by itself.* If a particular guess compresses (in isolation, as opposed to in the context of the rest of the response) beyond a certain threshold, we discard it.

2.4.5. *Block Ciphers.* Block ciphers pose an additional challenge to the oracle, since the plaintext size variations may not always result in ciphertext length variations. So, before we attempt to guess the secret, we first try to identify a 'tipping point'. A tipping point is the point at which any additional incompressible character causes the ciphertext to overflow into an additional block. When we are aligned to a tipping point, we hope that only correct guesses will fit within the current block, while incorrect guesses will cause the input to overflow into the next block, allowing us to use our compression oracle the same way we do for stream ciphers.

In order to get our response aligned to a tipping point, we need to add a filler string to the request. If the string is not chosen carefully, it can have other effects on the oracle. One approach to choose a filler string is to simply use a random sequence of characters that are outside of the target secret's alphabet. This avoids the situation where the filler string compresses against the target secret and guesses.

2.4.6. *Guess Windows.* It can be helpful to keep response sizes relatively stable between rounds of the attack. This is particularly helpful when the TLS connection uses a block cipher, since otherwise we'd have to continually realign to find a new tipping point. By only including the last n discovered characters of the secret, we can control the variation of the size of responses between rounds.

2.4.7. *Encoding Issues.* The contexts in which the target secret and our reflected guesses appear are important. For example, some pages embed the CSRF token as follows:

```
<input type="hidden"
  value="f710362da663742f76c71ed21c719c258d1b94f3"
  name="authenticity_token" class="authenticity_token
  ">
```

In contrast to OWA, note that the known prefix (`value=`) and the secret are separated by `"`. This means that in order for the attack to proceed, we need to either: have a request parameter reflect an unencoded `"`, or find another part of the page that will include our request parameter in the `value` attribute of some other HTML tag. Note that reflecting user-submitted `"` unencoded in responses often results in a cross-site scripting vulnerability. So, in order for the attack to not assume some other vulnerability, we are reduced (in the case where secret is presented as above) to requiring a request parameter to be reflected as the value of the `value` attribute in some other tag.

If the known prefix for the secret and the secret's value are separated by a character that will not be reflected from user input, we require some other coincidence in the way our guess is reflected (as described above). Otherwise, we lack an effective way to bootstrap the oracle. The separating character plays a role similar to the padding characters {} in the Two Tries method, but this time is working against us.

Finally, note that we could attempt to guess the first three characters of the secret to bootstrap the attack. However, this is clearly quite error-prone.

2.4.8. *False Negatives and Dynamic Padding.* Occasionally, there are cases where the oracle fails to identity a candidate for the next character in the target secret.

Due to some of the subtle inner workings of DEFLATE, it is possible for a longer repeated substring to *not* result in a smaller compressed payload.

Somewhat surprisingly, changing the amount of padding used sometimes allows the oracle to recover. Concretely, if sending requests with payloads of the form

```
canary={}x
```

and

```
canary=x{}
```

one might instead send requests of the form

```
canary={}{}{}{}{}x
```

and

```
canary=x{}{}{}{}{}
```

Currently, our only implementation of this randomly resizes the padding in cases where the oracle fails to identify any candidates for the next character. Of course, this technique does not always allow the oracle to recover. After some number of padding resize attempts, we abandon this recovery method.

2.4.9. *False Positives and Looking Ahead.* We remind the reader that the oracle does not compare sizes of requests across guesses, but rather looks for a differential in compression when padding is placed before and after the guess. If the request with padding coming after the guess elicits a smaller response, then the guess is deemed to be correct by the oracle. Because of this, it is possible that the oracle may find more than one 'correct' guess in a given round of the attack.

In such cases, it is sometimes possible to recover by having the attack branch temporarily. We proceed with the attack in multiple directions. For example, suppose that the oracle has produced `xyz` as the first three characters of the target secret. Now, suppose that in the fourth round of the attack, the characters `a` and `b` are deemed by the oracle to be correct guesses. We proceed with two parallel attacks, which assume that `a` and `b` are the most recently found characters in the target secret. That is, we will send requests with payloads `canary=xyza`{}`1` and `canary=xyza1`{} as well as `canary=xyzb`{}`1` and `canary=xyzb1`{}. (Of course, we also include requests for every other character of the secret's alphabet.)

Suppose that `a` is the actual fourth character of the secret. Let's suppose that `5` is the fifth character of the target secret. Then when we submit the pair of requests `canary=xyza`{}`5` and `canary=xyza5`{}, we should see a difference in compressed size of the corresponding responses. This is in contrast to the case with the requests `canary=xyzb5`{} and `canary=xyzb`{}`5`; since `b` is not the correct fourth character for the secret, `canary=xyz` is the longest substring that matches against the target secret in both cases. Ideally, in this case Even the correct guess for the fifth character (`5` in this case) will not produce a positive result if the fourth character is incorrect.

We remark that in practice, it is sometimes necessary to advance the attack several rounds before an incorrect branch can be trimmed.

3. Mitigation

We offer several tactics for mitigating the attack. Unfortunately, we are unaware of a clean, effective, practical solution to the problem.

3.1. **Length Hiding.** The crux of the attack is to be able to measure the length of the ciphertext. So, a natural attempt at mitigation is to hide this information from the attacker. It seems as though this should be simple and easy; one can simply add a random amount of garbage data to each response. Surely then the true length of the ciphertext will be hidden.

While this measure does make the attack take longer, it does so only slightly. The countermeasure requires the attacker to issue more requests, and measure the sizes of more responses, but not enough to make the attack infeasible. By repeating requests and averaging the sizes of the corresponding responses, the attacker can quickly learn the true length of the cipher text. This essentially boils down to the fact that the standard error of the mean in this case is inversely proportional to $\sqrt{N}$, where N is the number of repeat requests the attacker makes for each guess. For a discussion of the limits of length-hiding in a slightly different context, see [7]. We also comment that there is an IETF working group developing a proposal to add length-hiding to TLS [6].

3.2. **Separating Secrets from User Input.** One approach that completely solves the problem is to put user input in a completely different compression context than that of application secrets. Depending on the nature of the application and its implementation, this may be very tedious and highly impractical. It's likely that the easiest approach here is to move secrets away from any contexts that will be compressed against user-controlled data. For example: an application might use a secrets servlet, which serves files that contain secrets such as CRSF tokens, but no user-controlled data.

3.3. **Disabling Compression.** Disabling compression at the HTTP level also completely eliminates the side-channel, and therefore defeats the attack. Unfortunately, this solution can have a rather drastic impact on performance.

3.4. **Masking Secrets.** The attack relies on the fact that the targeted secret remains the same between requests. While it is usually impractical to rotate secrets on each request, there is a method due to Tom Berson which can synthesize this effect. Instead of embedding a secret S in a page, on each request, generate a new onetime pad P, and embed $P||(P \oplus S)$ in the page.[1]

[1] Here, we use $||$ to denote concatenation, and $\oplus$ to denote XOR.

We remark that $P||(P \oplus S)$ adds to the overall length of the traffic required to go over the wire. First, this doubles the length of every secret. Second, this guarantees that the secret will not be compressible. While secrets such as CSRF tokens should not be compressible in the first place, the attack can of course work against email addresses and other PII that may be compressible. In these cases, this approach will entail a genuine loss of compressibility.

3.5. **Request Rate-Limiting and Monitoring.** While the attack requires a non-ridiculous number of requests (thousands for OWA), it does require more requests in a short amount of time than a human would ever make. By monitoring the volume of traffic per user, and potentially eventually throttling users, the attack can at least be slowed down significantly.

3.6. **More Aggressive CSRF Protection.** An important feature of the attack is its reliance on the ability to coerce the victim to issue requests that elicit certain responses. In the example of OWA, the requests are simple GET requests that don't change state on the server. Until now, it has not been considered necessary to protect these requests from CSRF attacks. Furthermore, making such a change would be a significant retrofit for many web applications. However, we remark that requiring a valid CSRF token for all requests that reflect user input would defeat the attack.

4. Future Work

There's certainly more work to be done here. Our feeling is that this side-channel is poorly understood. There are likely other interesting variants that remain undiscovered, or at least undisclosed.

4.1. **Understanding DEFLATE.** We freely admit that our understanding of DEFLATE is somewhat pedestrian. While we've developed and refined various methods for making the oracle more reliable, it still occasionally fails to recover the target secret. Furthermore, some of our tactics (e.g. randomizing the amount of padding used) are a bit blunt and frankly poorly understood by the authors.

A careful study of DEFLATE will undoubtedly uncover improvements to the attack. Instead of randomizing the amount of padding used, perhaps there's a way to determine exactly how much padding should be used for a given request. Maybe there are other completely novel ways to remove the pollution due to Huffman coding.

4.2. **Other Protocols.** We remark that there is nothing particularly special about HTTP and TLS in this side-channel. Any time an attacker has the ability to inject their own payload into plaintext that is compressed, the potential for a CRIME-like attack is there. There are many widely used protocols that use the composition of encryption with compression; it is likely that other instances of this vulnerability exist.

4.3. **Something else?** Please send us pull requests! Our code can be found at `https://github.com/nealharris/BREACH`.

References

[1] [online]URL: `http://en.wikipedia.org/wiki/CRIME_(security_exploit)` [cited July 12, 2013].

[2] Tal Be'ery and Amichai Shulman. A perfect CRIME? Only TIME will tell. Technical report, Black Hat Europe, 2013. URL: `https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf`.

[3] P. Deutsch. Deflate compressed data format specification. RFC 1951, RFC Editor, May 1996. URL: `http://www.ietf.org/rfc/rfc1951.txt`.

[4] Thai Duong and Julianno Rizzo. The CRIME Attack, September 2012. URL: `https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/` [cited July 12, 2013].

[5] John Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276. Springer, February 2002.

[6] A. Pironti and N. Mavrogiannopoulos. Length hiding padding for the transport layer security protocol. Internet-Draft draft-pironti-tls-length-hiding-00, IETF Secretariat, February 2013. URL: `http://tools.ietf.org/pdf/draft-pironti-tls-length-hiding-00.pdf`.

[7] Cihangir Tezcan and Serge Vaudenay. On hiding a plaintext length by preencryption. In *ACNS'11 Proceedings of the 9th international conference on Applied cryptography and network security*, pages 345–358. Springer, 2011.