**BREACH ATTACK**

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) is an instance of the CRIME attack against HTTP compression. It was first announced in 2013 at Black Hat USA conference by Angelo Prado, Neal Harris and Yoel Gluck [no.7]. The attack is a side-channel attack to HTTPS that targets HTTP compression with the aim of disclosing secrets such as CSRF tokens and victims’ credentials under certain conditions [no.7,8].

**Overview**

CRIME attack targeted HTTP requests, while BREACH attack targeted HTTP responses [no.3]. The initial CRIME attack focused on request headers and relied on TLS compression, so by disabling it, the attack was mitigated. However, this mitigation method did not work for BREACH. BREACH attack used the same concept as CRIME, but focused on secrets that are within HTTP response bodies. HTTP responses are compressed using the common HTTP compression, which allows the attack to be carried out without relying on TLS-level compression and without tampering with or downgrading SSL [no.5].

HTTP compression is based on DEFLATE which works by eliminating repetitions in strings of text. The more repetitions, the more potential there will be for compression to reduce the overall size [no.6]. It uses a combination of LZ77 and Huffman Coding. LZ77 reduces redundancy by replacing occurrences of three or more characters with pointer values to reduce space. Huffman Coding, on the other hand, replaces the more common bytes with shorter bytes, usually symbols, to optimize the description of the data to the smallest size possible [no.10]. BREACH works by attacking the LZ77 compression while minimizing the effects of Huffman Coding.

It is common for web applications to not only deliver secrets such as CSRF tokens in the HTTP response body, but also reflect user input such as URL parameters within the response body [no.2]. Since DEFLATE takes advantage of repeated strings for compression, an attacker can guess the secret, one character at time, using the reflected URL parameter in the response body. For instance, assuming the first character of the attacker’s guess matches the first character of the CSRF token, DEFLATE compresses the response more efficiently, and serves as an oracle for the attacker. The attacker is then able to repeat this process to recover the entire CSRF token.

While LZ77 makes this attack easy, Huffman Coding presents a challenge for the attacker. When you replace common bytes with shorter sequences, it ultimately compresses the overall size to something smaller. The oracle in this case becomes confused as to whether the overall size is smaller due to LZ77 which would indicate a match with the attacker’s string, or if the smaller size is as a result of Huffman Coding since the character is very common in the response. If isolation between the two components is not performed, the result will be too many false positives, which reduces the overall effectiveness of the attack [no.10].

**Prerequisite for Breach Attack**

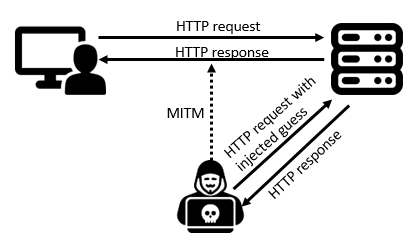
For a web application to be considered vulnerable to the BREACH attack, it must possess the following features:

* The server uses HTTP-level compression, for instance DEFLATE.
* User input should be reflected in the body of the HTTP response.
* The HTTP response body should reflect a secret, for instance a CSRF token.

BREACH attack is further aided if the size of the responses largely stays the same. Any noise in the channel makes the attack more difficult [no.2,3].

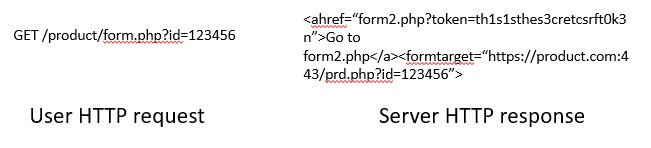
**Attack Implementation**

BREACH attack involves a victim, an attacker, and a server as depicted in Figure "X". The victim and the attacker need to be on the same network to allow the attacker to see the victim’s traffic (man in the middle).

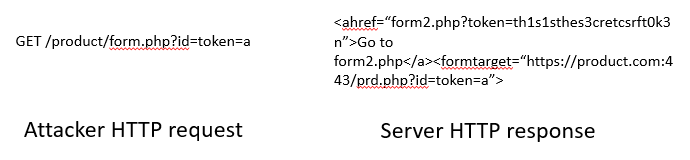


The attack leverages compression to extract data from a SSL/TLS channel by taking advantage of HTTP compression used in the HTTP response bodies. The attacker injects a guess into the HTTP request and measures the size of the compressed encrypted responses. A smaller response size indicates that the guess closely matches the secret value. The attacker then repeats this, one character at a time, while closely monitoring the size, until a perfect match is obtained [no.10].

Assume the HTTP request and response shown in Figure “X”. Once the server verifies that the parameters submitted by the user are correct, it sends a response back with a secret called “token” being reflected in the response:



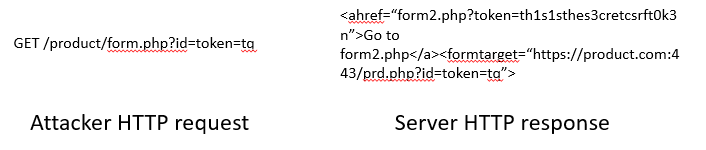
This functionality is leveraged by the attacker to guess the value of the token. In the first attack request, the attacker injects a guess “token=a” into the id parameter as shown in Figure “X”:



The attacker then proceeds to measure the size of the response sent back. Due to HTTP compression, if the attacker’s guessed value matches the first value of the actual token, the size of the response will decrease by the number of duplicate strings.

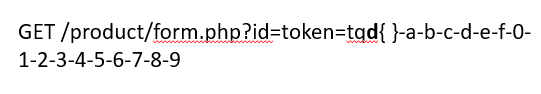
In this case, the size of the actual token “token= th1s1sthes3cretcsrftok3n” is 30. A correct guess from the attacker would result in a response size that decreased by 7. However, the attacker observes that the size decreased by 6, and can conclude that the guess was incorrect.

The attacker then proceeds to try different values for the token. When a correct token value is guessed, in this case “token=t”, the size of the response will decrease by 7. As a result, the attacker establishes that the first character of the token was successfully guessed, and accordingly retains this character as a constant. The attacker then tries different values for the second character, for instance “token=tq” as shown in Figure “X”:



The process is then repeated until the attacker successfully obtains the entire value of the token. The idea is to change the input and compare the size of the responses repetitively until the entire secret is recovered [no.10].

To counter the effects of Huffman Coding, an attacker can use padding as shown in Figure “X”:



In this instance, the characters “tq” are already known, ‘d’ is the attacker’s next guess and {} is padding. This is sent with 16 possible values, and the smallest response will represent the correct guess [no.10].

**Practicality**

BREACH attack is easily executable in less than a minute with just a few thousand requests, with the number of requests being dependent on the size of the secret [no.3]. It relies on the fact that before being sent to a user, a webpage is compressed in order to save bandwidth. Although the response will be encrypted, the size of the compressed response can still be obtained [no.8]. The attack was observed to work on all versions of SSL/TLS [no.10].

BREACH was performed on Microsoft Outlook Web Access. The entire CSRF token could be reliably recovered, approximately 95% of the time, often in under 30 seconds [no.2].

**Mitigation**

The following techniques can be used to mitigate BREACH attacks [10]:

* **Heal the BREACH (HTB):** This mitigation method modifies the compression library used by the web server to add randomness to the size of the response [no.8]. As a result, the length of the compressed HTTP response is modified, and this prevents an attacker from guessing the characters of the secret token.
* **Disabling HTTP compression:** The root cause of this attack is HTTP compression. Therefore, disabling this functionality mitigates the attack. However, this method comes with a caveat that the overall performance of the web application will be significantly affected [no.10].
* **Separating secrets from user input:** Putting secrets in a different compression context from the rest of the response body can mitigate this attack. In this case, changes to user input do not disclose secret information in the length of the compressed HTTP response body [no.10].
* **Masking secrets:** This method involves generating a onetime pad P, and embedding P||(P⊕S) in the page. P||(P⊕S) doubles the length of every secret and guarantees that the secret is not compressible [no.2]. In addition, masking the secret with a onetime random value with every request ensures that a new secret is generated every time.
* **Length hiding:** Adding random values to the compressed HTTP response body prevents an attacker from being able to calculate the size difference of the responses after compression [no.10].

**References:**

1. [~~https://en.wikipedia.org/wiki/BREACH~~](https://en.wikipedia.org/wiki/BREACH)
2. <http://css.csail.mit.edu/6.858/2020/readings/breach.pdf>
3. <https://breachattack.com/>
4. [~~https://iacr.org/archive/fse2002/23650264/23650264.pdf~~](https://iacr.org/archive/fse2002/23650264/23650264.pdf)
5. <https://www.researchgate.net/profile/Ashutosh-Satapathy/publication/310761924_A_Comprehensive_Survey_on_SSL_TLS_and_their_Vulnerabilities/links/58d1045e92851c1db43dfbfd/A-Comprehensive-Survey-on-SSL-TLS-and-their-Vulnerabilities.pdf>
6. <https://arstechnica.com/security/2013/08/gone-in-30-seconds-new-attack-plucks-secrets-from-https-protected-pages/>
7. <https://www.researchgate.net/publication/323352314_Implementing_a_proven-secure_and_cost-effective_countermeasure_against_the_compression_ratio_info-leak_mass_exploitation_CRIME_attack>
8. <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9754554>
9. [~~https://nakedsecurity.sophos.com/2013/08/06/anatomy-of-a-cryptographic-oracle-understanding-and-mitigating-the-breach-attack~~](https://nakedsecurity.sophos.com/2013/08/06/anatomy-of-a-cryptographic-oracle-understanding-and-mitigating-the-breach-attack)
10. <https://d.cxcore.net/InfoSec/ssl_attacks_survey.pdf>