ATTACKS ON SSL
A COMPREHENSIVE STUDY OF
CRIME, TIME, BREACH

CRIME

Compression Ratio Info-leak Made Easy (CRIME) is an attack on SSL/TLS that was developed by researchers Juliano Rizzo and Thai Duong. CRIME is a side-channel attack that can be used to discover session tokens or other secret information based on the compressed size of HTTP requests. The technique exploits web sessions protected by SSL/TLS when they use one of two data-compression schemes which are built into the protocol and used for reducing network congestion or the loading time of web-pages. Rizzo and Doung demonstrated it at the Ekoparty security conference in September 2012 after notifying major affected software vendors, including Mozilla and Google. CRIME is known to work against SSL/TLS compression and SPDY,19 although other encrypted and compressed protocols are also likely vulnerable.

HOW IT WORKS

Compression is a mechanism to transmit or store the same amount of data in fewer bits. The main compression method20 used in TLS to compress data is DEFLATE. DEFLATE21 consists of two sub algorithms: Lempel-Ziv coding or LZ77, and Huffman coding. LZ77 is used to eliminate the redundancy of repeating sequences, while Huffman coding is used to eliminate the redundancy of repeating symbols. It scans input, looks for repeated strings and replaces them with back-references to last occurrence as (distance, length) and wraps the content in a zlib-formatted stream. One of the important parameters in this compression technique is the window size. It has a value between 1 and 15; the higher the window size, the higher the compression ratio. Distances are limited to 32K bytes, and sequence lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes.

During a TLS handshake, in the ClientHello message, the client states the list of compression algorithms that it supports (by compression, we are discussing only TLS compression and SPDY, HTTP compression is not considered in the scope of this section). The server responds, in the ServerHello message, with the compression algorithm that will be used. Compression algorithms are specified by one-byte identifiers. When TLS compression is used, it is applied on all the transferred data, as a long stream. In particular, when used with HTTP, compression is applied on all the successive HTTP requests in the stream, including the header. CRIME is a brute-force attack that works by leveraging a property of compression functions, and noting how the length of the compressed data changes. The internals of the compression function are more sophisticated, but this simple example can show how the information leak can be exploited.

Suppose an HTTP request from the client looks like this:

The size of the content is length(encrypt(compress(header + body)). Even though the content is encrypted, the compressed content length is visible to the eavesdropper. The attacker also knows that the client will transmit Cookie: secretcookie= and wishes to obtain the following secret value. So by means of JavaScript, attacker issues a request containing Cookie: secretcookie=0 in the query string. Now the HTTP request from the attacker looks like this:

Here the input has attacker-controlled data, i.e. secretcookie=0, and is part of the request. Because of the redundancy with secretcookie, after compression the length will be smaller than if

the attacker-controlled data did not match any existing string in the request. The idea is to change input and measure and compare lengths to guess the secret value.
When compression function processes the request, it will recognize the repeated ``secretcookie='' sequence and represent the second instance with a very short token (one which states previous sequence has length thirteen (``secretcookie='') and was located n bytes in the past); the compression function will have to emit an extra token for the `0' however. Now, the attacker tries again, with secretcookie=1 in the request header. Then, secretcookie=2, and so on.
All these requests will compress to the same size, except the one which contains secretcookie=7, which probably22 compresses better (16 bytes of repeated sequence instead of 15 bytes), and thus will be one byte shorter in the content length. The request with cookie value starting with `7' compresses better is an indication that `7' is the first character of the secretcookie value. Thus after few requests, the attacker can guess the first byte of the secret value. Repeating
this process (secretcookie=70, secretcookie=71, and so on) the attacker can obtain, byte by byte, the complete secret.
The maximum size of TLS record is 16 kilobytes. When the record size is more than 16 Kbytes, TLS splits the record into separate records and compresses each record individually. If an attacker knows the location of the secret, then by adding proper padding in the request path, attacker can force TLS to split the request in such a way that the first record will have only one unknown byte. Once the attacker finds a possible match, the attacker removes 1 byte of
padding, and TLS again splits the record such that it has only one unknown byte in the first record, and continually repeats until attacker obtains the complete secret.
The attack above can also be optimized. If the secret value is in base64, i.e.. there are 64 possible values for each unknown character, the attacker can make a request containing 32 copies of Cookie: secretcookie=X (for 32 variants of the X character). If one of them matches the actual cookie, the total compressed length will be shorter. Once the attacker knows which half of the supplied alphabet the unknown byte is part of, the attacker can try again with a 16/16 split, and so on. In 6 requests, this homes in on the unknown byte value.

ATTACKER'S PERSPECTIVE

In a single session the same secret/cookie is sent with every request by the browser. TLS has an optional compression feature where data can be compressed before it is encrypted. Even though TLS encrypts the content in the TLS layer, an attacker can see the length of the encrypted request passing over the wire, and this length directly depends on the plaintext data which is being compressed. Finally, an attacker can make the client generate compressed requests that
contain attacker-controlled data in the same stream with secret data. The CRIME attack exploits these properties of browser-based SSL. To leverage these properties and successfully implement the CRIME attack, the following conditions must be met:

- The attacker can intercept the victim's network traffic. (e.g. the attacker shares the victim's (W)LAN or compromises victim's router)
- Victim authenticates to a website over HTTPS and negotiates TLS compression with the server.
- Victim accesses a website that runs the attackers code.

When the request is sent to the server, the attacker who is eavesdropping sees an opaque blob (as SSL/TLS encrypts the data), but the compressed plaintext's length23 is visible. The only thing the attacker can fully control is the request path.
In order to perform the attack, the attacker must load attack code to be loaded into the victim's browser, perhaps by tricking the victim into visiting a compromised or malicious website or by

injecting it into the victim's legitimate HTTP traffic when connected over an open wireless network. Once this is done, the attacker can force the victim's browser to send repetitive requests to the target HTTPS website using following options:

• Cross-Domain requests
• Moving the payload to the query string in a GET request
• Using <img> tags (a method used by Rizzo/Duong)

The attacker can then compare every request that the attack code is sending to the server. Every request with a correct guess of the secret will be shorter than requests with incorrect guesses. Thus by comparing the content length, the attacker can obtain the values of correct guess and hence, complete secret.

FEASIBILITY

This attack is feasible on all browsers and servers that support TLS compression. According to the Qualys SSL Lab's SSL Pulse24 test data showed 42% of servers and 45% of the browsers supported TLS compression when the attack was released. Internet Explorer, Safari, and Opera were not affected, as they did not support TLS compression.

Among the widely used web browsers, Google Chrome (NSS) and Mozilla Firefox, as well as Amazon Silk supported TLS compression as they implement DEFLATE. The attack also worked against several popular Web services that support TLS compression on the server side, such as Gmail, Twitter, Dropbox and Yahoo Mail. This attack worked for all TLS versions and all cipher suites (AES and RC4) and even if HSTS is active and preloaded by the browser vendor.

CRIME is a the brute-force attack, so it requires O(W) requests where W is cookie charset, with the possibility to optimize to O(log(W)). The modified version of SSL Strip25 by Moxie Marlinspike can be used in a public network to launch a man-in-the-middle attack which will satisfy one requirement of the attack. This tool strips the ongoing SSL/TLS session and performs a man-in-the-middle attack by acting as a proxy. The proof of concept code by Krzysztof Kotowicz26 is also useful to simulate the attack. Duong and Rizzo's pseudo code works well in practice, but does not include a mechanism to sync the JavaScript with the program observing lengths on the network.

Browsers still support HTTP compression, and this attack is possible on HTTP compressed sessions. Timing Infoleak Made Easy (TIME) is an extension of this attack. Recently Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) introduced a new targeted techniques to reliably retrieve encrypted secrets.

COUNTER MEASURES

There are several possible mitigations that vary in difficulty, feasibility of implementation, and loss of functionality. As CRIME exploits the compression algorithm, it is easy to suggest defenders to stop using TLS compression. Even to disable compressing any content containing secrets or to disable compressing secret data and attacker-controlled data in the same compression stream without flushing the compression state between the two will be tricky; given the complexity of modern web sites and the large set of user controlled attack vectors. So the approach of disabling TLS compression to mitigate the vulnerability has the most supporters. Implementation of this mitigation is fairly easy and can be patched through updates from the vendors with minimum manual intervention.

Both Chrome and Firefox have disabled TLS compression (and SPDY compression if used) in their browsers, and various other server software packages disabled it after disclosure of the attack by Rizzo and Doung — although browsers still support HTTP compression. TLS

compression will only compress HTTP requests and response headers — a small percentage of the traffic compared to the body of web application pages which is compressed when HTTP compression is enabled.
This attack is still possible with vulnerable client on certain web servers which still support TLS compression, as compression is not disabled by default (except for IIS, which doesn't support compression at all). Servers can be tested for TLS compression using the SSL Labs service27 (look for ``Compression'' in the ``Miscellaneous'' section) or using iSEC Partners' SSL scanning tool sslyze.28

TIME

Timing Info-leak Made Easy (TIME) is an attack that was developed by Tal Be'ery and Amichai Shulman of Imperva, and is a chosen plaintext attack on HTTP responses. The attack model of CRIME gives information about plaintext based on length of encrypted and compressed data. TIME uses this model and timing information differential analysis to infer the compressed payload's size. In TIME's attack model, the attacker only needs to control the plaintext, theoretically allowing any malicious site to launch a TIME attack against its visitors, to break SSL/TLS encryption and/or Same Origin Policy (SOP).

HOW IT WORKS

The CRIME attack has two practical drawbacks.
1. The CRIME attack is aimed at HTTP requests, specifically learning a victim's cookie value. TLS compression has been disabled both in major browsers and server software, thus rendering the CRIME attack irrelevant in those contexts.
2. CRIME requires the attacker to perform a man-in-the-middle attack against the victim.
TIME addressed these two limitations. The Internet today utilizes HTTP response compression and recommends it as a best practice for speed and bandwidth gains — so TIME shifted the attack target from HTTP requests to HTTP responses.
Before diving more deeply into the TIME attack, let's briefly review a few factors that play a key role in the successful execution of this attack.
Same Origin Policy (SOP) is a mechanism that governs the ability for JavaScript and other scripting languages to access Document Object Model (DOM) properties and methods across domain names. In order to prevent malicious scripts served from an attacker site to learn data from another site, browsers apply SOP. But SOP doesn't apply to multimedia tags like img. This creates a strange situation where a page has programmatic control over parts of its contents, but not others. Well documented SOP information leaks are available to the embedding page in this situation: success or failure of the load of the embedded content30 and the content's load time. This breaks SOP and allows some data to leak from one domain to another.
Round-Trip Time (RTT) represents the time it takes for a IP datagram to reach its intended recipient, plus the time it takes for the sender to receive the recipient's acknowledgment. This time delay is the transmission times between the two points. The datagram, whose size is lesser than Maximum Transmission Unit (MTU), will have lesser RTT than the datagram larger than MTU. At its core, the TIME attack leverages RTT timing differences and HTTP compression in order to infer the content of an HTTP request sent by the browser which may contain sensitive data. Maximum Transmission Unit (MTU) is the largest size of an IP datagram which may be transferred using a specific data link connection. The MTU value is a design parameter of a LAN and is a mutually agreed value (i.e. both ends of a link agree to use the same specific value) for

most WAN links. The size of MTU may vary greatly between different links (e.g. typically from 128 B up to 10 kB). The prevalent Path MTU on the Internet is now 1500 bytes.
When a packet is larger than MTU, IP Fragmentation happens.31 The RTT difference of the two packets (one inside the range of MTU and other at least 1 byte greater than the MTU) is then significant enough to measure.
Maximum Segment Size (MSS) is the maximum data octets in a TCP segment exclusive of TCP (or IP) header, which can be sent in a single IP datagram over the connection. Theoretically, MSS can be of 65495 bytes, but in practice, the size of MSS is sum of TCP and IP header bytes lesser than the outgoing interface MTU. MSS is calculated as:
MSS = MTU - sizeof(TCPHDR) - sizeof(IPHDR)32 [where TCPHDR is TCP header and IPHDR is IP header] TCP Sliding Window System is a protocol that allows optimization of the byte stream by allowing a sending device to send all packets within the agreed upon widow size before receiving an ACK.
The motive of the attacker is to force the length of the compressed data to overflow into an additional TCP packet.
The attacker then pads the compressed data to align to the amount of TCP packets allowed within the TCP window(i.e. sliding window). When the TCP window is maxed out, any additional packet created due to incorrect guesses, causes an additional full round trip with a significant delay; as it has to first wait for ACK from previous packets before it can slide the TCP window and allow for another packet to be sent.
Now consider this example to describe the attack mechanism. Let us consider a user input ``secret element = unknown data'' which is the payload. secret element and its value is in the response, and whatever the user inputs also gets reflected within the response.
In the first iteration, the user input is anything arbitrary and the response size is 1024 byte. Now if the user input is ``secret element = a'', then the response size will be 1008 bytes because of compression. Hence, it will take less time than the first iteration. Likewise with multiple requests the attacker will find out the shortest response time for every character in the specific position of the payload which will happen only in the case of a correct guess. Those specific values will be the value of the secret element.
The attacker injects malicious code/ JavaScript capable of sending multiple requests with attacker controlled data to the target website and measuring the response time. The JavaScript works by taking advantage of the flaw in SOP. If payload length is exactly on the window boundary, the attacker can determine 1 byte differences as it will cause an additional RTT with significant delay. When the attacker-controlled data matches the correct guess, the response time does not consist of additional RTT. With every byte of correct guess of the secret in the response, the padding is adjusted to maintain 1 extra byte more than the boundary. Thus by calculating the time difference, the attacker can successfully obtain the complete secret. By using repetitive requests, the attacker can completely find out the secret with no eavesdropping. This secret is not restricted to only the session cookie, but covers all sensitive information like user name, CSRF token, bank account number, etc. that is embedded in the response.

ATTACKER'S PERSPECTIVE

To execute the TIME attack, the attacker needs to know some information about the HTTP response. The attacker needs to know the secret element's location, secret element's prefix/suffix (secret/cookie are often structured so they have a fixed prefix or suffix) and a location to insert a chosen plaintext (many applications embed user input as expressed by HTTP parameters within

their response). The attacker can get this information by studying a few responses from the target website. The motive of the attacker is to find out secret element value (secret/cookie value).
The attacker creates HTTP requests with JavaScript and response timing leaks the request size. To eliminate noise over the network, if the process is repeated for certain time, the minimal response time can be recorded from the set of values of response time.
An attacker can even use this timing information to find out whether the victim is logged into some specific application or not by sending few requests to the application through this attack. If the user is logged in, the response will contain more information relevant to the user account. If the user is not logged in, the response will be login page of the application which will contain less information than the previous case. Hence the response size will likely be higher and so will be the response time as well.

FEASIBILITY

The secret elements that can be exploited by the TIME attack include the position and known prefix/suffix of these elements. While the exact specifics of these elements are application specific, they are present in every application.
One important requirement of this attack to be successful is to reflect back users' input in the response (not to be confused with the reflected cross-site script33 attack). Given everything else in the response remains constant, varying the reflected user input varies the content and the size of the response. If the user's input containing the guess of the secret element is not reflected back, there will be no measurable changes in the compression size of the HTTP response and hence no measurable changes in response time.
Timing measurements are affected by random network noises. Congestion and packet loss in any of the elements (client, routers and server) within the routing path can introduce random latency. In order to eliminate this noise, it has been found that repeatedly sending the payload (say 10 times) and taking the minimal timing value will account for such random latency effects. User input is encoded before embedding into the response to protect against injection attacks. Therefore the attack target is limited mostly to alphanumeric characters.

4.4 COUNTER MEASURES

This timing attack does not directly exploit any vulnerability in SSL/TLS, but the timing information leaks and existence of TIME attack defies the purpose of integrity and confidentiality of SSL/TLS. Below are the countermeasures which deal with changing how the application is developed instead of changing how the protocol works.
• Adding random timing delays to the decryption for any timing attack can be reasonable to disrupt statistical analysis, but it is not completely effective. An attacker who can gather enough samples can average many observations which make the randomness disappear. Adding random delay can only makes the attack take longer, but not impossible.
• Browser should support and respect ``X-Frame-Options"34header for all content inclusion (not just IFRAME); this way an application can restrict the displaying of simple multimedia content, like images on other applications.
Thus, allowing applications to take control over the presentation of their content on other domains.
• Applications should have a strict restriction on the reflection of user input in the response. The application should also be able to handle unknown variable input (like an extra variable in the URL) and prevent it from reflecting back in the response.

• Enabling anti-automation techniques like CAPTCHA, CSRF tokens etc. can be useful to restrict repetitive requests from an attacker.

BREACH

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH36)(VU#98779837) is an application of a compression side-channel CRIME style of attack on HTTP responses. Yoel Gluck, Neal Harris, and Angelo Prado demonstrated this attack at BlackHat38 USA 2013. The CRIME attack model gives information about session cookies to the attacker who is able to inject chosen plaintext into the victim's HTTP request and measure the size of these requests. In September 2012 when major vendors disabled TLS compression in browser and server-side, the CRIME attack was effectively mitigated. TIME resurrected the CRIME attack focusing on the time differential in the HTTP response, occurring due to the difference of size as an effect of HTTP compression of the response body.
Finally, BREACH revived the CRIME attack by targeting the size of compressed HTTP responses and extracting secrets hidden in the response body.

HOW IT WORKS

Once again, like the CRIME attack, BREACH exploited the compression and encryption combination used to interact with users and web-servers. The working mechanism of BREACH is similar to CRIME, except CRIME targeted TLS compression, while BREACH targets HTTP compression. HTTP response compression compresses the body of responses but not header information. The algorithm used, DEFLATE, is comprised of two components. LZ77
replaces occurrences of three or more characters with ``pointer'' values to reduce space. Huffman coding replaces characters with symbols in order to optimize the description of the data to the smallest size possible. BREACH works by attacking the LZ77 compression while minimizing the effects of Huffman coding. If this isolation is not performed, too many false positives will result, reducing the effectiveness of the attack.
At a high level the attack works by injecting guesses in HTTP requests and measuring the sizes of the compressed and encrypted responses. The smallest response size indicates that the guess matches the secret value. This is then repeated on a character by character bases. For example, let us consider the following dummy HTTP request and response:
``*CSRFtoken="*is reflected back in the response body as the value of the ``user'' parameter which is controlled by the attacker. The goal of the attacker is to learn the value of the CSRF token. So in the first request the attacker will send:---
and the attacker will measure the response size. Because of the redundancy with ``CSRFtoken=", after compression the length will be smaller than if the attacker-controlled data did not match any existing string in the request. The idea is to change input and measure and compare lengths to guess the correct secret value, repetitively until the whole secret is recovered.
In order counter the effects of Huffman coding, techniques such as guess swaps, padding and charset pools areused. As an example for some already guessed string CSRFtoken=4bf, the next character would be guessed using the payload:

Here the {} represent padding and d the guess value. This will be sent with the 16 possible values. The smallest response will represent the correct guess. Mounting the attack against stream cipher does not require any alignment, but with block ciphers, the guesses are block wise

aligned such that a correct guess will fit withing the target block while an incorrect guess will carry over to the next block.

ATTACKER'S PERSPECTIVE

To leverage the advantage of compression of response body for ex-filtration of the secret, the attacker need to have the ability to

• Monitor the encrypted traffic traveling between the victim user and website; this can be accomplished by ARP spoofing.

• Force the victim to visit attacker-controlled website. To achieve this the attacker needs to either inject an iframe, send phishing emails with a link to an attacker-controlled website40 or intercept and modify (like injecting image redirects) the plaintext HTTP traffic generated by the victim. The attacker-controlled website then forces victim's computer to send multiple request to the target website. Then the oracle process the byte-by-byte modified responses sizes to determine the correct guess of the secret.

FEASIBILITY

As BREACH focuses on the HTTP compression of the response body, it is possible to mount on all versions of SSL/TLS, and does not require TLS-layer compression. The cipher suite used during the session negotiation does not affect this attack. The number of requests required are proportional to the size of secret, but in general BREACH attack can be exploited with just a few thousand requests, and under a minute. In short, the scope of this attack includes a considerable portion of the HTTP traffic in the Internet as a large portion of enterprise applications and online websites use HTTP compression to optimize bandwidth.

The three main requirements for exploitation of the vulnerability to be effective are:

1. The application supports HTTP compression.
2. The response should reflect back user's input.
3. The response should have some sensitive/ secret information embedded in the body.

If the user's input is not reflected, there is no possible way to mount a chosen plaintext attack and measure the size of the responses. This attack targets the secret information in the response body (e.g. CSRF tokens), not the session cookie in the request header. So this is useful only if the the response of this attack contains sensitive information. Like CRIME and TIME, the oracle needs to be aware of Huffman coding scheme and overcome the false positives generated due to the same. In their research paper,41 Gluck, Harris, and Prado gave a detailed explanation on methods to overcome the aberrations caused by the subtle inner working of the DEFLATE and how they were able to optimize the attack.

COUNTER MEASURES

At present there is no perfect solution. To mitigate this issue, Gluck, Harris, and Prado suggested few of the mitigation options in their research paper.

• **Disabling Compression:** Disabling HTTP compression affects the root cause of the problem and hence completely mitigates it. Unlike TLS compression and SPDY, HTTP compression is an essential technology that can't be replaced or discarded without inflicting considerable overhead on both website operators and end users. Disabling HTTP compression will affect the speed and performance of the web-server to a significant level.

• **Length Hiding:** Adding garbage value to the compressed response body and obfuscating the actual length will affect the ability of the attacker to calculate the differences in the length, occurring due to compression of varying input. IETF working group is working on developing a

proposal to add length-hiding on TLS.42 However, this only increases the time of the attack and thefore does not mitigate it effectivly. The attacker needs to send more request to correctly measure the response sizes.

- **Separating Secrets from User Input:** Serving secrets in a different context than the rest of the body, during compression, can completely solve the issue. In this case, the length of the compressed response body will not be able to disclose secret information with the change of user input. However, implementation of this solution is complicated depending on the nature of the application.
- **Masking Secrets:** In practice, the secrets like CSRF tokens should be changed with every request. However, that is not always the case and this attack depends on the assumption that the secrets remains same in between the responses. So if the secrets in the body can be masked with a one time random value (new secret S0 = Rjj(R_S), where R is the one time random value and S is the secret) with every request, it will generate a new secret every time.
- **Request Rate-Limiting and Monitoring**: To implement the attack, attacker needs to send thousands of requests from victim user's computer to the web-server within a very short amount of time. It is not possible by human action to generate such volume of traffic. If the server-side monitoring system diagnoses such behavior, it can either i.throttle the user requests, which will slow down the attack or ii. invalidate the session and notify the user for a probable attempt of intrusion into the user's session. The warning might create panic among the users, but it will at least help the user to stop doing sensitive transaction over insecure network connection.