ATTACKS ON SSL
A Comprehensive Study of beast, crime, time, breach, lucky 13 & rc4 biases

Pratik Guha Sarkar – psarkar@isecpartners.com
Shawn Fitzgerald — shawn@isecpartners.com

iSEC Partners, Inc
123 Mission Street, Suite 1020
San Francisco, CA 94105
https://www.isecpartners.com

August 15, 2013

Abstract

Over last few years, a number of vulnerabilities have been discovered in the Transport Layer Security protocol. The purpose of this paper is to serve as an analysis of recent attacks on SSL/TLS and as a reference for related mitigation techniques; particularly as they relate to HTTPS.

1 Introduction

Transport Layer Security (TLS), the successor of Secure Sockets Layer (SSL), is the most popular and widely used application of practical cryptography in the world. The most commonly use is for securing web browser sessions, but it has widespread application to other tasks, such as securing email servers or any kind of client-server transaction. TLS can also be used to tunnel an entire network stack to create a VPN, and to provide authentication and encryption of Session Initiation Protocol (SIP). SSL/TLS can be used to provide strong authentication of both parties in a communication session, strong encryption of data in transit between them, and ensures integrity of the data in transit. Over the last few years, various types of attacks have been designed to take advantage of select properties of the SSL/TLS architecture, design and weaknesses of the cipher suites used in SSL/TLS for encryption and key establishment. The attacks discussed in this paper are:

- Browser Exploit Against SSL/TLS (BEAST) attack
- Compression Ratio Info-leak Made Easy (CRIME) attack
- Timing Info-leak Made Easy (TIME) attack
- Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) attack
- Lucky 13 attack
- RC4 biases in TLS

These attacks undermine the security mechanisms provided by SSL/TLS; however, mitigations can be designed without overhauling the basic structure of the protocol.

The paper is organized as follows: Section 2 to 7 describes the six attacks mentioned — each of the sections contains some background and simple explanation of the attack, likelihood of the attack scenario and its mitigation. Sections 8 and 9 explore the overarching recommendations and support and protection offered by TLS 1.2.

2 BEAST

The ability to mount an adaptive chosen plaintext attack with predictable initialization vectors (IVs) against SSL/TLS using cipher block chaining (CBC) was known in 2004,[1] but until late 2011 was thought to be largely theoretical. Researchers Thai Doung and Juliano Rizzo[2] found a way to exploit the vulnerability and demonstrated a live attack against Paypal[3] at the Ekoparty security conference in September of 2011. Doung and Rizzo had notified and had been working with major software vendors including Mozilla and Google to release a patch (CVE-2011-3389).[4] Although the vulnerability is cryptographic in nature, it requires certain conditions to be successful. The proof of concept code presented at the conference also required a Java-based Same Origin Policy[5] (SOP) bypass that they had found during their research and which has been patched by Oracle.[6]

2.1 How it works

BEAST leverages a type of cryptographic attack called a chosen-plaintext attack. The attacker mounts the attack by choosing a guess for the plaintext that is associated with a known ciphertext. To check if a guess is correct, the attacker needs access to an encryption oracle[7] to see if the encryption of the plaintext guess matches the known ciphertext.

To defeat a chosen-plaintext attack, popular configurations of TLS use two common mechanisms: an initialization vector (IV) and a cipher block chaining mode (CBC). An IV is a random string that is XORed with the plaintext message prior to encryption — even if you encrypt the same message twice, the ciphertext will be different, because the messages were each encrypted with a different random IV. The IV is not secret; it just adds randomness to messages, and is sent along with the message in the clear. It would be cumbersome to use and track a new IV for every encryption block (AES operates on 16-byte blocks), so for longer messages CBC mode simply uses the previous ciphertext block as the IV for the following plaintext block.

The use of IVs and CBC is not perfect: a chosen-plaintext attack can occur if the attacker is able to predict the IV that will be used for encryption of a message under their control and the attacker knows the IV that was used for the relevant message they are trying to guess.[8]

This new research demonstrated that the above attack can be mounted against TLS under certain conditions. When a SSL 3.0 or TLS 1.0 session uses multiple packets, subsequent packets use an IV that is the last ciphertext block of the previous packet, essentially treating the session as one long message. This allows an attacker who can see

[1] http://eprint.iacr.org/2004/111.pdf

[2] http://vnhacker.blogspot.com/2011/09/beast.html

[3] Paypal was targeted due to its high profile, and use of strong security features like HTTP Strict Transport Security, which make simple attacks like SSL stripping ineffective. There was nothing special or insecure about Paypal's SSL deployment.

[4] http://www.cvedetails.com/cve/CVE-2011-3389

[5] *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.

[6] http://www.oracle.com/technetwork/topics/security/javacpuoct2011-443431.html

[7] A way to encrypt arbitrary plaintext messages with the relevant encryption key.

[8] If the attacker Mallory wants to know the target message P_{real} that used a known initialization vector IV_1, she knows that the resulting ciphertext (known to Mallory) is equal to: $E_k(IV_1 \oplus P_{real})$. If Mallory also knows the encryption oracle will use IV_2 for the next encryption, Mallory can make a guess P_{guess} by injecting the plaintext message $P_{guess} \oplus IV_1 \oplus IV_2$, which will produce the ciphertext of: $E_k(P_{guess} \oplus IV_1 \oplus IV_2 \oplus IV_2) = E_k(IV_1 \oplus P_{guess})$. If the ciphertexts match, the attacker knows that $P_{real} = P_{guess}$.

encrypted messages sent by the victim to see the IV used for the session cookie, the because cookie's location is predictable,[9] and also know the IV that will be used at the beginning of the next message packet (the last ciphertext block from the current message packet). If the attacker can also "choose" a plaintext message sent on behalf of the victim, they can make a guess at the session cookie and see if the ciphertext matches.

The ability of an attacker to mount a chosen plaintext attack against SSL/TLS with predictable IVs is well known.[10] In essence the underlying cryptographic construct in SSL version 3.0 and TLS version 1.0 creates IVs by always using the last ciphertext block of packet number j as the IV for packet number $j+1$. What this means for an active attacker on a network is that they are able to sniff the IV for each record.[11]

Suppose the attacker wants to guess that message m_i may be secret x. She picks the next plaintext block to be $m_j = m_i \oplus c_{(i-1)} \oplus c_{(j-1)}$. Recall that the last block on ciphertext becomes the IV for the next block and that XORing two identical values cancels them out. c_j then becomes:

$$c_j = E(k, c_{(j-1)} \oplus m_j) = E(k, c_{(j-1)} \oplus c_{(j-1)} \oplus m_i \oplus c_{(i-1)}) = E(k, m_i \oplus c_{(i-1)}).$$

If $c_j = c_i$ then the guess of secret x is correct. This attack, while interesting from a theoretical perspective, is not very practical. This is because an attacker can only guess whole blocks; assuming the secret x is a random value, then an attacker would have to guess on the order of the block size of the algorithm (*e.g.* 2^{128} for AES). However, if an attacker can split the secret over blocks under her control, she would be able to fixate on one byte of secret in a block that otherwise contains known information. The search space is therefore reduced to one byte times the number of bytes to guess.[12]

Guessing the entire 16 bytes of a random session cookie is impractical, but the attack cleverly makes it easier by controlling the block boundaries (by adjusting URL parameters for example) in order to guess the secret one byte at a time. For example, say the actual plaintext session cookie is:

Company_Session_Token:Edx38NesewqeNI872Def32sfe

It would be difficult to guess 16-bytes of the unknown session cookie value if you have the ciphertext associated with the secret portion "Edx38NesewqeNI87". Instead, the attacker can try to adjust the encryption block boundaries and guess the plaintext value of a ciphertext block associated with an easier plaintext block: "_Session_Token:?" which only contains a single unknown character that is much easier to guess.[13] After the first character has been correctly guessed as "E", the attacker can guess the next character by shifting the block boundary to align with the plaintext block "Session_Token:E?" and continue the attack until the entire cookie has been recovered. If the cookie is a random value that is base64 encoded, the attack would only take about 32 rounds per character in the cookie.

2.2 Attacker's perspective

If we consider the case of a standard client/server based browser interaction, the following are requirements for a successful attack:

Passive Network Eavesdropping: The attacker must be able to capture encrypted HTTPS requests sent by the client. This will allow them to capture both the ciphertext of the encrypted secret value and the IV for any given encryption block.

Chosen-Boundary Format Privilege: The attacker can control the block-based boundary of the secret. In the case of a standard web interaction, the attacker can control where in the message the cookie is sent by adding URL parameters, or adjusting headers. By adjusting the boundaries of the encryption blocks, the attacker can try and

[9] This is because in the context of HTTP messages the cookie is sent as a header value which is generally static.
[10] http://eprint.iacr.org/2004/111.pdf
[11] Except for the initial IV, which is not transmitted but generated from the ClientHello.random and ServerHello.random.
[12] Note that we are simplifying complexity of the byte-wise attack for the sake of clarity.
[13] Recall that the attack will only let the attacker know if they have guessed the entire 16-byte plaintext block correctly.

create blocks that are easier to guess. The attacker will create a block that only contains one unknown byte (and 15 known byte values such as the name of the cookie) that will be easier to guess.

Chosen-Blockwise Plaintext Injection Privilege: The attacker can guess plaintext values associated with ciphertext blocks obtained from passive network eavesdropping. Specifically, the attacker can mount a chosen plaintext attack by injecting plaintext blocks of their choice to be encrypted. This essentially means that the attacker can use the client browser as an encryption oracle. In addition this requires that the same SSL/TLS session (*i.e.* same session key) be maintained across multiple injections. In practice, this privilege likely requires a Same Origin Protection bypass to be successful in a browser environment, but such a requirement is not a documented part of browser's security model, and is even less likely to be a safe assumption on non-browser TLS systems.

2.3 Feasibility

A successful implementation of the attack requires browser or web technologies to meet the above criteria. For clarity, we walk through the example as follows:

The network attacker (who we will call Mallory) has the ability to eavesdrop on the network (*e.g.*, over a wireless network) and coerces Alice to visit http://mallory.com perhaps through phishing, advertising or another attack. The malicious website contains an attack script that forces Alice's browser to make a request to http://bob.com[14] and Mallory records the encrypted cookie. Using a technology that allows Mallory to adapt the attack through a SOP bypass or technology that allows multi-origin communication,[15] Mallory now tries to guess the session cookie as the first block of subsequent requests and sees if the resulting ciphertext matches the previously recorded session cookie ciphertext.

The actual attack is likely very difficult because of the browser's enforcement of the SOP, but it is possible that some web technologies provide a mechanism for cross-domain communication. Doung and Rizzo list several such technologies that could potentially be used to create the attack script such as HTML5 WebSocket API, Java URLConnection API (as demonstrated in their presentation through exploitation of a separate SOP bypass), and Silverlight WebClient API. It should be noted other than using the Java SOP bypass vulnerability, no other exploit avenues have been publicly disclosed.

2.4 Counter Measures

This class of attack is well known enough that it was mitigated in TLS version 1.1; however, due primarily to client compatibility reasons neither TLS 1.1 or 1.2 are widely supported on the web and most vendors still require support (*i.e.* fall back) for SSL v3.0 and TLS v1.0[16]

Browser vendors have attempted to implement a workaround to address the vulnerability at the implementation level while still remaining compatible with the SSL 3.0/TLS 1.0 protocol. These initially included inserting empty fragments into the message in order to randomize the IV as in the case of OpenSSL, and when that proved problematic to reliably implement, 1/n-1 record splitting where a single byte of the plaintext is injected in each record. The resulting padding added to complete the block (16 or 15 bytes) is random, and its search space is too high for an attacker to guess.

With this in mind we recommend the mitigations:

- Ensure users have updated to patch browsers and vulnerable technologies such as Java.
- Enable TLS version 1.1 and preferably version 1.2.

[14]We assume that Alice has already authenticated to bob.com and has an established session cookie.

[15]Multi-origin communication permits scripts running on pages originating from the different site to access each other's methods and properties with no specific restrictions.

[16]List of browsers support for different TLS version — https://en.wikipedia.org/wiki/Transport_Layer_Security#Web_browsers

- Migrate TLS protected services without compatibility issues, like some VPNs, to TLS 1.2.
- Disable cross-origin requests[17] on the server side when they aren't needed.
- Track how many customers do not support TLS version 1.2 and disable support of lower versions of TLS, when possible. This is likely very possible for internal tools such as TLS-based VPN or corporate intranet webservers because of the ability to influence the clients being used.

2.5 Software supporting the mitigation

The following software supports 1/n-1 record splitting at patches against the vulnerability. Although as noted there is currently no known exploit vector for this vulnerability.

2.5.1 Browsers

- Google: Update to Chrome 16 or later.
- Microsoft: Ensure that MS12-006 has been applied.
- Mozilla: Update to Firefox 10 or later.

2.5.2 Libraries

Libraries supporting TLS 1.1 or higher :

- OpenSSL: Ensure that openssl-1.0.0e-cve-2011-3389.patch has been applied.
- NSS: Update to version 3.13 or later.
- Microsoft: Ensure that MS12-006 has been applied.

3 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 (DEFLATE and gzip) 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 (CVE-2012-4929[18]). CRIME is known to work against SSL/TLS compression and SPDY,[19] although other encrypted and compressed protocols are also likely vulnerable.

[17] `http://www.w3.org/TR/cors/`

[18] `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4929`

[19] SPDY is another open networking protocol developed primarily at Google for transporting web content. SPDY is similar to HTTP, with particular goals to reduce web page load latency and improve web security. SPDY achieves reduced latency through compression, multiplexing, and prioritization. Although the mechanics are different, SPDY also compresses an entire stream of data. SPDY uses zlib for compression which performs the basic steps where it "output these literal bytes" and then "go back x bytes and duplicate y bytes from there", and thus it is also vulnerable. More on CRIME attack on SPDY: `https://www.imperialviolet.org/2012/09/21/crime.html`.

3.1 How it works

Compression is a mechanism to transmit or store the same amount of data in fewer bits. The main compression method[20] used in TLS to compress data is DEFLATE. DEFLATE[21] 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:

```
POST / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0) Gecko/20100101 Firefox/14.0.1
Cookie: secretcookie=7xc89f94wa96fd7cb4cb0031ba249ca2
Accept-Language: en-US,en;q=0.8

(... body of the request ...)
```

Listing 1: *HTTP request of the client*

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:

```
POST /secretcookie=0 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0) Gecko/20100101 Firefox/14.0.1
Cookie: secretcookie=7xc89f94wa96fd7cb4cb0031ba249ca2
Accept-Language: en-US,en;q=0.8

( ... body of the request ...)
```

Listing 2: *HTTP request modified by the attacker*

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.

[20] http://www.iana.org/assignments/comp-meth-ids/comp-meth-ids.txt
[21] https://tools.ietf.org/html/rfc1950

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 probably[22] 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.

```
--- record 1 ---
GET /PADDING_TO_FILL_RECORD<…>PADDING HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0) Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=7

--- record 2 ---

xc89f94wa96fd7cb4cb0031ba249ca2
Accept-Language: en-US,en;q=0.8
```

Listing 3: *Record splitting using chosen boundary*

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 (because $2^6 = 64$).

3.2 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.

[22] Huffman's coding constructs characters in variable bit length code. If an alphabet which includes only English lower case letters and space, and the commonly used letter in the English language is encoded with only 3 bits, while less frequent letters are encoded with 7 bits. So the difference in size of the incorrect guess might be unobservable if the addition of the incorrect guess character is smaller than a byte.

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 length[23] 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.

3.3 Feasibility

This attack is feasible on all browsers and servers that support TLS compression. According to the Qualys SSL Lab's SSL Pulse[24] 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 Strip[25] 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 Kotowicz[26] 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 Info-leak 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.

3.4 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

[23]The attacker sees the encrypted content's length with byte granularity when the connection uses RC4; with block ciphers there is a bit of padding, but the attacker can adjust the contents of the requests so that attacker may phase with block boundaries, so, in practice, the attacker can still know the exact length of the compressed request.

[24]https://community.qualys.com/blogs/securitylabs/2012/09/14/crime-information-leakage-attack-against-ssltls

[25]http://www.thoughtcrime.org/software/sslstrip/

[26]https://gist.github.com/koto/3696912

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 service[27] (look for "Compression" in the "Miscellaneous" section) or using iSEC Partners' SSL scanning tool sslyze.[28]

3.5 Software supporting the mitigation

3.5.1 Browsers

All the latest versions of browsers disable TLS compression by default, as of the versions listed below:

- Internet Explorer: No versions of IE support SSL/TLS compression
- Chrome: 21.0.1180.89 and above
- Firefox: 15.0.1 and above
- Opera: 12.01 and above
- Safari: 5.1.7 and above

3.5.2 Libraries

- Apache 2.2.x using mod_SSL[29]: Apache 2.2.24 has support for the SSLCompression flag. SSLCompression is on by default - to disable it specify "SSLCompression off".
- Apache using mod_gnutls: In mod_gnutls user can specify the GnuTLSPriorities flag to disable compression. Specify "!COMP-DEFLATE" to disable TLS compression.
- IIS: Microsoft IIS does not support TLS compression (including IIS 7.5+/Server 2008 R2).
- Amazon Elastic Load Balancers: iSEC Partners has confirmed with Amazon that Elastic Load Balancers do not support TLS compression.

4 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

[27] https://www.ssllabs.com/
[28] https://github.com/iSECPartners/sslyze/downloads
[29] http://svn.apache.org/viewvc?view=revision&revision=1395231

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).

4.1 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 content[30] 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 win-

[30] https://grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information
[31] http://www.erg.abdn.ac.uk/~gorry/eg3567/inet-pages/ip-fragmentatiion.html
[32] http://tools.ietf.org/html/rfc879

dow(*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.

4.2 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.

4.3 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 script[33] 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

[33] https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

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"[34]header 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.

4.5 Software supporting the mitigation

4.5.1 Browsers

X-Frame-Options support is present in the following and above versions of the browsers[35]

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	4.1.249.1042	3.6.9 (1.9.2.9)	8.0	10.5	4.0
ALLOW-FROM support	Not supported	18.0 (18.0)	8.0	?	Not supported

5 BREACH

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH[36])(VU#987798[37]) is an application of a compression side-channel CRIME style of attack on HTTP responses. Yoel Gluck, Neal Harris, and

[34] https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options
[35] https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options#Browser_compatibility
[36] http://breachattack.com/
[37] http://www.kb.cert.org/vuls/id/987798

Angelo Prado demonstrated this attack at BlackHat[38] 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.

5.1 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:

```
GET /product/?id=12345&user=CSRFtoken=<guess> HTTP/1.1
Host: example.com
```

Listing 4: *Compromised HTTP request*

```
<form target="https://example.com:443/products/catalogue.aspx?id=12345&user=CSRFtoken=<guess>" >
...
<td nowrap id="tdErrLgf">
<a href="logoff.aspx?CSRFtoken=4bd634cda846fd7cb4cb0031ba249ca2">Log Off</a></td>
```

Listing 5: *HTTP 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: `GET /product/?id=12345&user=CSRFtoken=`**`a`** `HTTP/1.1` 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 are used. As an example for some already guessed string *CSRFtoken=4bf*, the next character would be guessed using the payload:

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

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.

[38] http://www.blackhat.com/us-13/briefings.html#Prado

5.2 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.[39]
- 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 website[40] 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.

5.3 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.

5.4 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.

[39] http://rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf

[40] Gluck, Harris, and Prado have been able to run BREACH without JavaScript/ iframes, simply with image redirects on an email. So BREACH can also be triggered from a malicious email payload just by viewing the email in Outlook. The attacker still needs traffic visibility to measure encrypted responses.

[41] Section 2.4 of "BREACH - SSL, gone in 30 seconds" http://breachattack.com/resources/BREACH-SSL,gonein30seconds.pdf

- **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 $S' = R||(R \oplus S)$, 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.

5.5 Software supporting the mitigation

While this is an ongoing issue, current mitigations included the following:

After the release of the attack Django issued a security advisory[43] to temporarily mitigate BREACH attack against Django's CSRF protection. The advisory suggested:

- Disabling Django's GZip middleware.[44]
- Disabling GZip compression in users web server's config. In Apache, disable mod_deflate;[45] and in nginx disable the gzip module.[46]

It should be noted that until various software mitigations have been standardized and their implications well understood, they should be analyzed both from a security perspective and a feasibility perspective before being applied in a blanket manner.

[42]http://tools.ietf.org/pdf/draft-pironti-tls-length-hiding-00.pdf
[43]https://www.djangoproject.com/weblog/2013/aug/06/breach-and-django/
[44]https://docs.djangoproject.com/en/dev/ref/middleware/#module-django.middleware.gzip
[45]http://httpd.apache.org/docs/2.2/mod/mod_deflate.html
[46]http://wiki.nginx.org/HttpGzipModule

6 Lucky 13

Nadhem AlFardan and Kenny Paterson of the Information Security Group at Royal Holloway, University of London, found a new timing attack in TLS/ DTLS[47] called Lucky Thirteen (CVE-2013-0169[48]). The attack allows a man-in-the-middle attacker to recover plaintext from a TLS connection when CBC-mode (cipher-block chaining) encryption is used. There is a subtle timing bug in the way that TLS data decryption works when using the (standard) CBC-mode ciphersuite. Lucky Thirteen uses the same attack mechanism as the padding oracle attack.[49]

6.1 How it works

A Message Authentication Code (MAC) is used to authenticate and to provide integrity of the message. The best practice is to encrypt a message first, then apply the MAC on the resulting ciphertext. However, in TLS it is done in a different fashion. The message is added in the block, a MAC is applied to the plaintext, and then up to 255 bytes of padding are added to grow the message to a multiple of the cipher block size (8 or 16-byte). This message block is finally CBC-encrypted. CBC mode decryption takes the current encrypted block, decrypts it, and XORs in the previous ciphertext block. After decryption, the padding is first validated, and after successful validation it is removed and then integrity of data is checked against the calculated MAC.

```
|  data   | MAC of plaintext | padding |
```

Listing 6: *Structure of message block used for encryption*

However, this method of CBC-encrypting in TLS has a problem with protecting the padding. The padding oracle attack is applicable to the implementations of SSL 3.0 and TLS 1.0. The padding is not protected by the MAC, so an attacker can tamper with the padding and perform a padding oracle attack. During decryption of the message, the padding is checked first. If there is a valid padding, only then the MAC is checked; otherwise the server throws an error stating whether that an invalid padding or MAC error has occurred. The padding oracle attack uses CBC decryption to determine the plaintext by modifying the previous ciphertext block. An attacker can modify the encrypted message based on these error messages, and after repetitive requests can eventually get the message decrypted by the server without the encryption key.

This padding oracle problem was fixed by eliminating any explicit error messages that could indicate to the attacker whether the padding check or the MAC check is what caused a decryption failure. This solution left the implementation vulnerable with the possibility of a timing attack, as the attacker can notice the time differences in the server responses in cases of bad padding. In TLS 1.1 and above, anytime a record fails to decrypt (due to a bad MAC or padding error), the TLS server kills the session. This was implemented to prohibit an attacker from repetitively sending requests to decrypt the encrypted message, but padding oracle attacks can work across different sessions, provided the victim re-initiates the session after it drops, and the secret appears in the same position in each stream. The design of browsers and HTTPS satisfy both of these requirements. To solve this time differential problem, the TLS specification 1.2[50] states that even if padding fails, the MAC should be validated considering that the value of the padding is null. When the padding check fails, there is no way to figure out the size of actual message and the number of padding bytes to strip; hence no way to calculate the correct MAC. As per the specification the whole blob is used to calculate the MAC. As a result, the MAC computation can take a little bit longer when the padding is damaged. And this subtle timing bug is exploited by the Lucky Thirteen attack to decrypt the encrypted message.

TLS typically uses HMAC with MD5, SHA1 or SHA256 as the hash function. Each of these hash functions process messages in 64-byte blocks. The hash functions incorporate an 8-byte length field plus some special hash function

[47] Datagram TLS (DTLS) is a protocol based on TLS that is capable of securing datagram transport (for example, UDP). DTLS is well suited for securing applications that are delay-sensitive (and hence use datagram transport), tunneling applications (VPN), and applications that tend to run out of file descriptors or socket buffers.

[48] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0169

[49] http://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf

[50] https://tools.ietf.org/rfc/rfc5246.txt

padding, which actually means a one-block message can only contain about 55 bytes of real data (which also includes the 13-byte record header).

```
| Sequence | Header | App Data |
| 8-bytes  | 5-bytes|  n-bytes |
```

Listing 7: *Formatted message block to calculate MAC*

and the data encrypted is:

```
| Enc App Data | Enc MAC Tag | Enc Padding |
|    n bytes   |   20-bytes  |   p-bytes   |
```

Where p is the number of valid padding bytes. The sequence and header in the authenticated data add up to 13 bytes. That TLS MAC calculation includes 13 bytes of header information is, in part, what makes the attacks possible — this is where the name of the attack comes from. In the example given by Nadhelm and Kenny, the first application data packet is cut down to 85 bytes. The server interpreted the packet as containing:

```
|Header|IV|Enc App Data|Enc MAC Tag|Enc Padding|
|     5|16|      44 - p|         20|          p|
```

Listing 8: *Application Data Packet*

TLS requires the server to decrypt:

```
|Enc App Data|Enc MAC Tag|Enc Padding|
|      44 - p|         20|          p|
```

Listing 9: *Encrypted Message block*

Then, the authentication is performed on:

```
  |Sequence|Header|App Data|
  |       8|     5|  44 - p|
```

Listing 10: *Application Data Packet*

If no padding bytes are valid, then p is 0 bytes and the application data is 44 bytes. There are three scenarios to consider:

- The padding is wrong and 57 bytes are MACed.
- The last plaintext byte is 0×00 and 56 bytes are MACed.
- The last two plaintext bytes are 0×01, 0×01 and 55 bytes are MACed.

The attack tries different values in the second to last ciphertext block, this time in the last two bytes, in an attempt to force the last plaintext block to be the correct two bytes of padding. The attacker wants 55 bytes to be authenticated with HMAC-SHA1 because this can be distinguished computationally from 56 or 57 bytes as if it goes a single byte beyond 55, the hash function will have to run a whole extra round, causing a tiny (500-1000 hardware cycle) delay.

6.2 Attacker's perspective

A prerequisite for the attack is the ability to intercept the connection between the client and server to read the clear text TLS handshake messages and inject modified ciphertext. This is most commonly achieved on an open Wi-Fi network. To make a target browser initiate many connections, attacker can feed it some custom JavaScript (this doesn't necessarily need to come from the target webserver — it can even be served on an unrelated non-HTTPS page) or client-side malware. Due to design of the HTTP protocol, each of these connections will include cookies at a known location in HTTP stream. The malicious JavaScript can also control the location of the cookie such that there is only one unknown byte in the target block at each stage of the attack.

The attack here is to intercept a message and modify it, including the TLS padding, in such a way as to make it fall above that 55 byte boundary. However, the same message with padding properly removed would fall below it. When an attacker tampers with the message (damaging the padding), the decryption process will perform MAC operation on the longer version of the message; resulting in a measurably higher computation time than when the padding is valid. By repeating this process many thousands of times to eliminate noise and network jitter, it is possible to get a clear measurement of whether the decryption succeeded or not. Once the attacker obtains that information, attacker just needs to perform a standard padding oracle attack.

6.3 Feasibility

The attack applies to all implementations that conform to TLS version 1.1 or 1.2, or DTLS version 1.0 or 1.1. It also applies to implementations of SSL 3.0 and TLS 1.0 that have countermeasures designed to defeat a previous padding oracle attack discovered several years ago. All TLS and DTLS ciphersuites that include CBC-mode encryption are potentially vulnerable.

The latency generated by various infrastructures on the Internet is likely to make the attack completely infeasible. However, it may well be practical against fast internal networks as the latency is low or negligible. Thus the only practical limitation on such a cookie attack is the time it takes for the server to re-initiate all of these connections. TLS handshakes aren't fast, and this attack can take tens of thousands of connections per byte. So in practice the TLS attack would probably take days.

6.4 Counter Measures

Even though the following mitigation techniques were proposed by Nadhelm and Kenny, the mitigations also have their restrictions.

By careful implementation of all MAC-then-Encode-then-Encrypt(MEE) this Lucky 13 attack can be mitigated. If uniform processing time is implemented to decrypt ciphertexts of a given size, then the processing time of valid and invalid input will be same. This can be achieved by eliminating major time differences in MAC processing of the ciphertext, independent of the properties of underlying plaintext.

Adding random timing delays to the decryption for any timing attack can be reasonable. But this attack cannot be completely evaded by this countermeasure; it can only increase the number of samples required, and hence the duration of the decryption operation, and coupled with standard network latency this might make the attack highly unlikely.

Using an authenticated encryption algorithm, such as AES-GCM or AES-CCM is the ideal mitigation, however this is only implemented in TLS 1.2, which is not currently widely implemented.

6.5 Software supporting the mitigation

The researchers notified this issue to IETF TLS Working Group, IRTF Crypto Forum Research Group(CFRG) and individual vendors responsible for closed and open source implementation of SSL/TLS and they addressed the issue.

6.5.1 Libraries

Affected users should upgrade to the following versions of vendor specific implementation of SSL/TLS[51]:

- BouncyCastle addressed the attack in version 1.48 of the Java library. The release 1.8 of the C# version of BouncyCastle also corrected this issue.
- CyaSSL addressed the attack in version 2.5.0
- F5 indicated that their TLS dataplane traffic is not vulnerable due to cryptographic offload, but that local management ports and virtual editions may be vulnerable. F5 also said that the hotfix for this issue will follow shortly after OpenSSL issues their patch[52]
- GnuTLS: Attacks are addressed in versions 2.12.23, 3.0.28 and 3.1.7, released 04/02/13.
- Microsoft determined that their implementations are not impacted and already addressed.
- NSS addressed the attacks in version 3.14.3
- OpenSSL addressed the attacks in versions 1.0.1d, 1.0.0k and 0.9.8y
- Opera addressed the attack in Opera version 12.13
- PolarSSL addressed the attack in version 1.2.5
- Oracle (Java) addressed the attacks as part of a special critical patch update of JavaSE, released on February 19, 2012.

7 RC4 biases in TLS

RC4 is a stream cipher developed by Ron Rivest of RSA Security in the late 1980s. It was originally a proprietary cryptographic algorithm but was leaked in 1994 and is now is the most commonly used stream cipher. It has a long history of cryptographic analysis and cryptographic attacks, most notably the attack on its usage in the WEP protocol.[53] A number of additional weaknesses have been found over the years involving individual biases in the initial byte stream. Although none of these weaknesses were demonstrated to allow a practical attack against RC4 (especially as it is implemented in SSL/TLS), they are significant enough to warrant many in the cryptographic community to recommend against its use.[54] Nevertheless, due to performance and wide spread adoption, it has become a popular ciphersuite in SSL/TLS. However, recent research by Bernstein, Paterson, Poettering and Schuldt [55] as well as other unrelated research by Ohigashi et al. [56] appear to have developed full plaintext recovery attacks on RC4. One of these attacks, a broadcast attack, recovers the plaintext when it is sent repeatedly in many different sessions, such as the case when a new session key is negotiated within an existing channel in an existing HTTP session. The other attack recovers the plaintext when it is encrypted repeatedly in the same or several different sessions.

[51] http://www.isg.rhul.ac.uk/tls/TLStiming.pdf
[52] http://support.f5.com/kb/en-us/solutions/public/14000/100/sol14190.html
[53] http://en.wikipedia.org/wiki/Fluhrer,_Mantin_and_Shamir_attack
[54] http://csrc.nist.gov/groups/STM/cmvp/index.html
[55] http://www.isg.rhul.ac.uk/tls/RC4biases.pdf
[56] http://home.hiroshima-u.ac.jp/ohigashi/rc4/Full_Plaintext_Recovery%20Attack_on%20Broadcast_RC4_pre-proceedings.pdf

7.1 How it works

RC4 is an extremely simple and elegant algorithm. The first phase is the key scheduling algorithm (KSA). This algorithm takes an initial array and initializes it to values 0 to 255. For each index of the array a shuffling occurs that mixes in the key. Once this algorithm runs, the output of the KSA is input to the pseudo-random generation algorithm (PRGA) that continually shuffles the array. The output of the PRGA is exclusive-ored with the plaintext to produce a cipher text.

These recent attacks have found strong biases in the first 257 bytes of encryption which will allow recovery of roughly the first 200 bytes of plaintext in approximately 2^{28} to 2^{32} encryptions of the same plaintext under unique keys (referred to here as the broadcast attack). The attack recovers the plaintext at each position by gathering the set of observed ciphertexts (each encrypted with a different key) for the corresponding position. It tries each of the 256 candidate plaintexts and computes the PRGA output byte by exclusive-oring the candidate with each ciphertext. It then calculates which plaintext candidate resulted in PRGA outputs which most closely matches the known PRGA bias for that position. To take an extremely simplified example, consider when the PRGA output always outputs the value one. The correct plaintext will be the one which, when exclusive-ored with each of the ciphertexts, always results in one.

Additionally, using previously discovered long-term biases in RC4, 50% of a 16-byte secret value can be extracted after analysis of $6 * 2^{30}$ encryptions of the same plaintext message using a single key. This attack is different to the broadcast attack since it can recover plaintext from a one or more encryption streams in which the same plaintext is sent repeatedly. The attack recovers plaintext using transitional biases in the PRGA stream that recur at fixed positions in the stream. For example, if the PRGA output is zero at any offset that is a multiple of 256-bytes, then the next output is more likely to also be a zero. The attack works by starting with a known plaintext byte that repeats at a fixed position and finding a candidate for the plaintext byte that repeats at the next position. The candidate that most closely matches the transition bias of the PRGA is selected. This process is then repeated to find the next unknown plaintext byte.

In both attacks the request structure such as the URL, which may be known or the location and beginning of the cookie as well as the plaintext structure, such as the language or HTML can be used to further optimize the attack. Additionally, many byte positions may quickly be probabilistically reduced to a limited number of candidate plaintexts. This includes positions that may have multiple biases. The candidate plaintexts can be used to attempt to authenticate to the system under attack in order to verify the correctness of the secret.

7.2 Attacker's perspective

Mounting the broadcast or single-byte bias attack in HTTP-based environments requires an active network attacker that can force the victim to repeatedly establish or re-negotiate session keys for an active HTTP session such that the same plain text is encrypted with different session keys under RC4.

For an authenticated HTTP session, the unknown value which the attacker is trying to get is the session ID or cookie. This value is preceded by standard HTTP header information. If the cookie value is in the first 256 bytes of plaintext it will be exposed during the attack.[57] Most modern web traffic does not contain sensitive data in the first 256 bytes (or 220 bytes of message data), which limits the applicability of the attack.

Experimental results indicate that approximately 2^{21} encryption of a fixed plaintext in unique keys using renegotiation per hour is reasonable. However, it is likely not possible for an attacker to force re-negotiations via JavaScript. This then would force an attacker to repeatedly establish and tear down a full SSL/TLS handshake. This involves significant overhead and currently limits the practicality on the single-byte bias attack.

[57] It should be noted that some ciphersuites such as RC4 with SHA-1 will only allow the first 220 bytes to be recovered due to an initial 36 bytes of unpredictable "Finished" message.

The longer term bias attack does not require an active network attacker because it can be mounted using a single key. However, it does require the target plaintext to be repeatedly sent at the same place in the message. This can be performed by forcing a user to run the attackers JavaScript while authenticated to a legitimate web server. The average running time for this attack is on the order of 2000 hours. Although many cookies today are long lived, this attack is still at the limits of practicality and likely would be noticed.

7.3 Feasibility

In the current and un-optimized state, these attacks do not represent a practical threat against the majority of implementations. However, optimization can be made and depending on the individual target, analysis of the structure, language can be made to reduce the attack time frame.

7.4 Counter Measures

A number of counter measures have been proposed, however, the majority such as discarding the initial keystream bytes of output as well as randomizing HTTP requests can be bypassed. If feasible limiting Session ID lifetimes and throttling client initiated re-negotiations and connections from individual IP addresses can be utilized to make the attack significantly more difficult. Long term, it is recommended to depreciate usage of RC4 and move to patched versions of CBC based algorithms or ideally Authenticated Encryption based algorithms.

7.5 Software supporting the mitigation

Organizations are working to phase out RC4 as it no longer offers any extra security margin.

- Google is focusing on implementing TLS 1.2 and AES-GCM in Chrome.
- Microsoft has modified their code so that RC4 is no longer enabled by default[58] for TLS in Windows 8.1 Preview.
- Opera has implemented a number of countermeasures[59] to modify browser behavior.

This white paper will be updated once further information about patches is made available.

8 Countermeasures in TLS 1.2

The design of TLS 1.2 has incorporated many of the countermeasures and mitigations of the attacks discussed in this paper. Although TLS 1.2 has had slow adoption by the general community, this is likely to change in the near future. The following highlights specific design changes in TLS 1.1 and 1.2:

- An implicit Initialization Vector (IV) is replaced with an explicit IV in TLS 1.1. TLS versions 1.1 (RFC 4346)[60] and 1.2 (RFC 5246)[61] are not vulnerable to the BEAST attack. Padding errors raise a bad_record_mac alert, not decryption_failed alert, so a session can be resumed after a premature closure.

[58] http://technet.microsoft.com/en-us/library/dn303404.aspx
[59] http://my.opera.com/securitygroup/blog/2013/03/20/on-the-precariousness-of-rc4
[60] http://www.ietf.org/rfc/rfc4346.txt
[61] http://tools.ietf.org/html/rfc5246

- Authenticated Encryption with Associated Data[62] (AEAD) support is present in TLS 1.2 and hence new authenticated encryption modes (CCM, GCM)[63] have been introduced for AES. These modes are proven to be more secure than CBC mode and ECB mode and can be used to prevent BEAST attack, Lucky 13 attack and can also replace RC4 encryption for better security.
- Weak encryption schemes like IDEA and DES[64] cipher suites are now deprecated and will not be part of any cipher suites when negotiating TLS 1.2.
- When TLS 1.2 is negotiated, the MD5/SHA-1 PRF [65] is replaced with a suite specified hash function. The hash used is SHA-256 for all TLS 1.2 suites, but in the future another hash function can be defined such as SHA-3. The digitally-signed element include a field defining the hash algorithm it uses, which is SHA-256 in TLS 1.2
- *Verify_data* length is no longer fixed length. This allows TLS 1.2 to define several new, SHA-256 based, cipher suites.

As TLS 1.2 builds on TLS 1.1, TLS 1.1 must be implemented before implementing TLS 1.2. This is the requirement for the clients who are currently running TLS 1.0 and trying to upgrade to TLS 1.2. There are at least five key areas to look for before implementation of TLS 1.2 :

1. Operating System Support
2. Programming Language Environment Support
3. Web Server Support
4. Browser Support
5. Communication Library Support

Modern operating systems like Windows 7, Windows Server 2008 R2, many Linux distributions[66] like RedHat, Debian, SUSE, Ubuntu and Mandriva come with support for TLS 1.2. Programming Language Environment like SunJSSE in the Java SE 7 release and .Net Framework 4.5 support TLS 1.2. Apache 2.0, IIS 7.5+ and nginx web servers have also enabled the support of TLS 1.2, as the underlying communication libraries like OpenSSL, GnuTLS, CYASSL, NSS, Microsoft SChannel etc. extended their support for the same.[67] The latest version of Internet Explorer, Google Chrome, Mozilla Firefox, Opera and Safari started supporting TLS 1.2 for the desktop operating systems.[68]

The *SSL/TLS Deployment Best Practices*[69] is a good reference to consult during implementation of the SSL/TLS. Some hardened configuration files[70] for SSL/TLS created by various researchers are also available for reference.

9 General Long Term Recommendations

At the time of writing the majority of servers and modern browsers do not support TLS 1.2, despite its availability for the last five years. For a number of years, questions have been asked about how and why to upgrade to TLS 1.2. This is a typical deadlock situation where there is no advantage of upgrading web servers to TLS 1.2 without the corresponding browser support. Indeed, browser vendors are accustomed to dealing, on a weekly basis, with vulnerabilities which are much more devastating than many of the ones discussed in this paper (typically entailing

[62] https://tools.ietf.org/html/rfc5288
[63] More information at - https://www.isecpartners.com/media/19276/introduction_to_authenticated_encryption.pdf
[64] http://tools.ietf.org/html/rfc5469
[65] pseudorandom function
[66] https://wiki.linuxfoundation.org/en/OpenSSL#OpenSSL_versions_in_distributions
[67] http://en.wikipedia.org/wiki/Comparison_of_TLS_implementations
[68] https://en.wikipedia.org/wiki/Transport_Layer_Security#Web_browsers
[69] https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices_1.1.pdf
[70] https://github.com/ioerror/duraconf/tree/master/configs

hostile hijacking of the whole client system). A somewhat theoretical exploit of the cryptography in SSL/TLS is unlikely to be seriously examined, let alone acted upon. On the other hand, there is little motivation for browsers to support TLS 1.2 if a large percentage of servers do not support TLS 1.2. Hence there has been no real incentive for a mass update of client and server TLS implementation.

This upgrade logjam has been an ongoing issue, and the ”crisis in crypto” should force improvements in this area. Every major update requires time and resources, and it is always advisable to start upgrading early before attack techniques become more efficient and once theoretical exploits become practical and feasible. Instead of looking for incentives for mass upgrades, the deployment of TLS 1.2 can be initialized in environments where both servers and clients are controlled (typically internal networks/services). It is time to phase out TLS version 1.1 and below and everyone should make an effort to implement or add support for TLS 1.2, and only then can the situation fundamentally change.