TLS Record Protocol: Security Analysis and Defense-in-depth Countermeasures for HTTPS

Olivier Levillain
ANSSI
olivier.levillain@ssi.gouv.fr

Baptiste Gourdin
Sekoia
baptiste.gourdin@sekoia.fr

Hervé Debar
Télécom SudParis
herve.debar@telecom-sudparis.eu

ABSTRACT

TLS and its main application HTTPS are an essential part of internet security. Since 2011, several attacks against the TLS Record protocol have been presented. To remediate these flaws, countermeasures have been proposed. They were usually specific to a particular attack, and were sometimes in contradiction with one another. All the proofs of concept targeted HTTPS and relied on the repetition of some secret element inside the TLS tunnel. In the HTTPS context, such secrets are pervasive, be they authentication cookies or anti-CSRF tokens. We present a comprehensive state of the art of attacks on the Record protocol and the associated proposed countermeasures. In parallel to the community efforts to find reliable long term solutions, we propose masking mechanisms to avoid the repetition of sensitive elements, at the transport or application level. We also assess the feasibility and efficiency of such defense-in-depth mechanisms. The recent POODLE vulnerability confirmed that our proposals could thwart unknown attacks, since they would have blocked it.

1. INTRODUCTION

SSL (Secure Sockets Layer) is a cryptographic protocol designed by Netscape in 1995 to protect the confidentiality and integrity of HTTP connections. Since 2001, the protocol has been maintained by the IETF (Internet Engineering Task Force) and renamed TLS (Transport Layer Security). Designed in 2008, the current version of the protocol is TLS 1.2 [9]. The original objective of SSL/TLS was to secure online-shopping and banking web sites. With the deployment of web services using the so-called Web 2.0, its use has broadened drastically.

TLS typical use consists of two consecutive phases: the *Handshake protocol* negotiates the cryptographic algorithms and keys, and authenticates the server to the client using certificates; the second phase, the *Record protocol*, protects the confidentiality and integrity of subsequent messages, called records, carrying the application data. In this article, we only consider attacks on TLS Record layer, letting aside handshake-related subtleties such as session resumption, renegotiation or client authentication.

Depending on the handshake outcome, the records can be protected using one of the three following cryptographic schemes: *MAC-then-encrypt with a streamcipher*, available since SSL inception (as of today, only RC4 is standardized); *MAC-then-encrypt with a blockcipher using CBC mode*, available since SSL inception; *AEAD* (Authenticated Encryption with Additional Data), available from TLS 1.2 only (e.g. AES using GCM mode). Optionally, the plaintext can be compressed before cryptographic transformations occurs. Fig. 1 describes the three possible workflows.

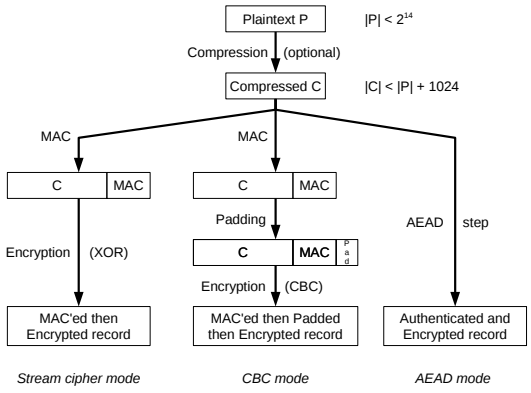

Figure 1: TLS Record protocol (streamcipher mode, CBC mode and AEAD mode).

Since 2011, several researchers have presented attacks affecting the Record protocol confidentiality. Each time, to prove the applicability of their findings, they implemented attacks against HTTPS. Typical HTTP secrets are cookies and anti-CSRF tokens. If stolen, they enable attacks like message replay, session highjacking or web site compromising. Fundamental to state maintenance in web applications, they are usually transmitted several times both within sessions and across different sessions. As a matter of fact, each proof of concept relied on one of these secret elements repeatedly transmitted inside TLS connections. Thus, a way to thwart attacks exploiting this kind of repetitions, existing or still unknown, is to mask the secrets so the attacks

ASIA CCS'15, April 14–17, 2015, Singapore..

ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714592.

become inefficient in practice. In this article, we focus on cookie protection. HTTP Basic/Digest Authentication, and server-side attacks are discussed in the appendices.

Sec. 2 describes several recent attacks, and the proposed countermeasures. Sec. 3 presents our attacker model and the masking principle. Two proofs of concept have been implemented to assess the applicability and efficiency of our proposals. Sec. 4 describes those implementations, whereas Sec. 5 analyses the effectiveness of the proposed mechanisms, as well as their impact on performance.

Our contribution is threefold: first, we propose a comprehensive analysis of recent attacks on TLS Record layer with their countermeasures; then we present a new defense-in-depth approach to the problem, orthogonal to the current efforts towards long-term solutions, and illustrate them with two proofs of concepts; finally, we analyse those implementations, security- and performance-wise.

2. ATTACKS ON THE RECORD PROTOCOL

This section describes five recent attacks published between 2011 and 2014, targeting a session cookie sent by the client.

2.1 BEAST: implicit IV in CBC mode

In 1995, Rogaway described an adaptive chosen plaintext attack against the CBC mode with a predictable IV [30]. In 2002 [21], it was noticed that TLS used predictable IVs, since the last encrypted block of a record is used as the next record IV. However, this attack was deemed impractical at the time, or at least challenging citeBard04–tls-cpa,Bard06–tls-cpa, since it was an adaptive chosen plaintext attack. The situation changed in 2011 when Duong and Rizzo presented BEAST (Browser Exploit Against SSL/TLS) [10], a proof of concept of the vulnerability. Fig. 2 (left-side) details the "Encryption (CBC)" step presented in Fig. 1, when implicit IV is used.

Using the notations of Fig. 2, the attacker can "guess" that the value of P_1 is equal to some $P^\star$ and check the validity of her guess. Indeed, after the two records have been sent, she knows C_5 will be the next record IV. Thus she sends $P_6 = P^\star \oplus C_5 \oplus C_0$ as the next plaintext block, and observes $C_6 = E(P_6 \oplus C_5) = E(P^\star \oplus C_0)$. If the guess was correct (i.e. P_1 is $P^\star$), $C_6 = C_1$, which is observable. The corresponding encryption step is given on the right side of the figure.

Furthermore, to avoid having to guess a whole block, the proof of concept cleverly aligns the blocks so that the block to guess only contains one unknown byte. For instance, if the attacker knows P_1 is `":SESSION_TOKEN=?"` where `?` is unknown, she only needs at most 256 attempts (128 on average) to recover the byte (or even less, if the targeted byte belongs to a constrained charset).

Finally, to send the chosen plaintext data in the same TLS session as the target web application, the Same Origin Policy (SOP[1]) [4] must be bypassed. Due to the complexity of the web ecosystem, vulnerabilities allowing SOP violations are regularly found on standard browsers and web applications[2].

[1]SOP is a fundamental security property implemented in web browsers, intended to prevent scripts loaded from a given site from communicating freely with another site

[2]Early BEAST implementations used WebSockets. Authors had to use a Java bug when WebSockets were randomised.

Hypotheses and prerequisites

- The connexion uses CBC mode with an implicit IV;
- The ciphertext is observable;
- The plaintext is partially controlled, adaptively;
- Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Switch to TLS 1.1. This is the long term solution, since TLS 1.1 introduces an explicit (and unpredictable) IV for each record. Yet, TLS 1.1 is still not widely deployed, as shown by different studies [28, 19]. What is worse, to accommodate broken implementations, up-to-date browsers use a fallback strategy when a TLS connection fails, and retry using older versions (TLS 1.0 or SSLv3).

Use TLS 1.2 AEAD suites. AES-GCM or AES-CCM are not vulnerable to this attack, but they are only available with TLS 1.2. As for TLS 1.1, this protocol version is not widely deployed, and may be subject to a downgrade attack.

Use RC4 to avoid CBC mode. With TLS 1.0 and earlier versions, this is an efficient way to counter *this* attack, and it can be deployed easily and reliably (but Sec. 2.4 shows this is not an overall acceptable solution).

Randomize the IV by splitting the record. By splitting the records to send in two records, the first one containing the first byte of the original record, and the second one the remaining data, the attack still works, but only on the first byte. This so-called "$1/n - 1$ split" is efficient and implemented in major browsers[3].

Fix SOP violations and XSS bugs. To mount the attack, forged requests must be sent to the target, either by bypassing the SOP or by exploiting a Cross Site Scripting (XSS) vulnerability. It is obviously desirable to fix all these bugs, but the ever-evolving web ecosystem makes this goal difficult to reach.

2.2 CRIME & TIME: client-side compression

In 2012, Duong and Rizzo published another attack against TLS named CRIME (Compression Ratio Info-leak Made Easy) [29]. Again, their objective was to recover a secret cookie. The attack is based on the compression step in the Record protocol and assumes the attacker is able to choose part of the cleartext, e.g. the URL path. The following year, another research team presented the TIME (Timing Info-leak Made Easy) attack [32], a variant of the CRIME attack, relying on a different feedback method.

Let's assume the secret cookie, `SESSION_ID`, is an hexadecimal string, and that the attacker can trigger successive HTTPS connections while controlling part of the cleartext (typically the URL path). This does not violate the Same Origin Policy, and the resulting HTTP requests will contain both the forged URL (`www.target.com/SESSION_ID=X` in our example, `X` being an hexadecimal character) and the cookies corresponding to the target. When these requests are compressed, the redundancy will be maximum when the attacker has guessed the secret correctly, which should re-

[3]Initially, a "$0/n$ split" had been implemented in OpenSSL, but it proved to break some implementations, despite empty ApplicationData records being licit.

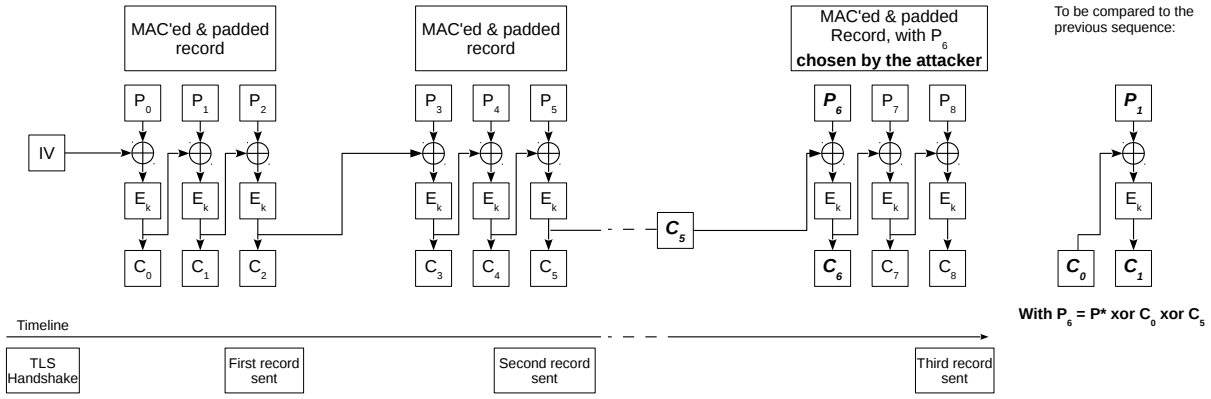

Figure 2: [Left] CBC with implicit IV in SSL/TLS before TLS 1.1: IV is generated during the handshake, then all records are encrypted as a continuous CBC flow. [Right] To check whether $P_1 = P^\star$, the attacker encrypts a record starting with plaintext block P_6, and compares the output C_6 to C_1, previously observed.

sult in a better compression. To make this phenomenon observable, the attacker needs the record to get smaller[4].

CRIME and TIME propose two different methods to observe the impact of compression on the plaintext. CRIME simply relies on the encrypted packet sizes, assuming the attacker is able to capture the victim's traffic. TIME uses a different feedback method: the variation of transmission time between a correct guess (where compression is more efficient) and an incorrect one[5]. To amplify the effect of a compressed plaintext being one byte shorter, the idea behind TIME is to forge a plaintext such that the encrypted packet just crosses the TCP window size and requires a TCP ACK from the server before sending the remaining byte. This way, when the attacker guesses the correct character, the compression kicks in and the encrypted data contains one byte less, which does not require to wait a Round Trip Time for the ACK. This difference in timing is observable from the client-side script launching the requests.

Hypotheses and prerequisites

- TLS compression is activated;
- The ciphertext length is observable, e.g. via packet sizes or timing leaks;
- Plaintext can be loosely controlled by the attacker;
- Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Disable TLS compression. This blocks the attack completely, and has no significant impact on performance.

Randomize the packet length. Several proposals were made to add random-length padding to application messages (adding random bytes or slicing the HTTP content in chunks). Unless the added data is significant enough (which is equivalent in practice to disabling compression), these proposals essentially force the attacker to collect more data, but do not fundamentally invalidate the attack.

Restrictions on cross-site requests. If cross-site requests were forbidden (or at least excluded sensitive information like authentication cookies), the attacker would first need to exploit an XSS to mount this attack, but many web applications would also break.

2.3 Lucky 13: CBC padding oracle

In TLS, when a blockcipher is used with CBC, the plaintext is MAC'ed then padded and encrypted, which allows for attacks exploiting padding oracles, first introduced by Vaudenay in 2002 [33]. As soon as an attacker can distinguish between a MAC error and a CBC padding error, be it through an out-of-band message or a timing difference, she can gain information about the plaintext.

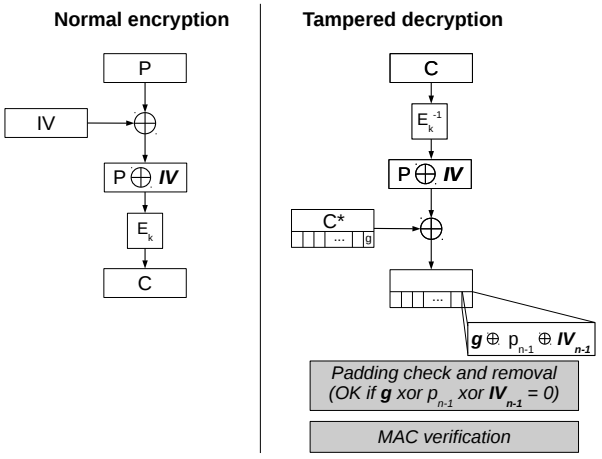

Figure 3: CBC encryption and decryption, in the light of a padding oracle exploitation. Blocks C, $C^\star$ (in particular its last byte g) and IV are known.

[4]To this aim, some parameters need to be tuned, like the block alignment (with CBC encryption) and a way to reset the compression dictionary (the main compression algorithm in TLS, Defalte, is stateful).

[5]TIME authors also described an attack to recover server-sent anti-CSRF tokens. Appendix B presents details on server-side compression attacks and countermeasures.

When decrypting a CBC-encrypted record, the recovered plaintext should end with a valid padding: p bytes all valued $p-1$ (for example, blocks ending with 00, 0101, 020202, etc. are correctly padded). Let $P = p_0 p_1 \cdots p_{n-1}$ be a plaintext block, and $C = c_0 c_1 \cdots c_{n-1}$ be the corresponding ciphertext (see Fig. 3, on the left side). To guess the value of p_{n-1}, the attacker can send a fake ciphertext containing two blocks: $C^\star C$, with C being the ciphertext block to recover and $C^\star$ a random block, ending with $c^\star_{n-1} = g$ (the decryption of the second block is described in Fig. 3, on the right side). If the guessed byte g is indeed equal to $p_{n-1} \oplus IV_{n-1}$, the output of E_k^{-1} will end with a null byte, the padding will be correct, and this will lead to a MAC error (since the attacker can not create a valid MAC). Otherwise, if the guess is incorrect, the padding will be incorrect, with overwhelming probability[6].

If the attacker can distinguish between MAC errors and CBC padding errors, she can use the resulting padding oracle to guess the content of a block, one byte at a time. Indeed, once the attacker has identified the last byte p_{n-1}, she can try and find whether $p_{n-2} = g$ with $C^\star$ ending this time with $(g \oplus 01)|(p_{n-1} \oplus 01)$, and so on.

The initial specifications of SSL/TLS stated that both error cases (invalid MAC and padding error) should lead to different alert messages. However, this was not directly useful from the attacker point of view, since the alert was encrypted. Another way to differentiate the two error cases is to measure the time needed to reject the invalid packet. When no MAC is performed, the answer is returned faster. That is why TLS 1.1 [8] contains a note stating that *implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct.*

Moreover, such attacks were initially considered impractical against TLS since modified records would eventually trigger a MAC error, be rejected, and cause the whole session to close. In 2012, Paterson et al. studied the applicability of this attack to DTLS [23]. Datagram TLS (DTLS) [27] is a cryptographic protocol similar to TLS relying on UDP instead of TCP; since UDP is not a reliable transport layer, datagrams may be lost or corrupted. Furthermore, DTLS does not close a session when a MAC error is encountered (nor does it emit a warning). The authors identified a timing attack that made it possible to distinguish a padding error from a MAC error (to amplify the timing info-leak, several identical consecutive packets are sent on the wire).

In case of a padding error, the standard implementation of TLS CBC decryption assumes a fixed-length pad, which, according to the implementation note quoted earlier, *leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable.* It was established in 2013 that this small info-leak was in fact exploitable in TLS to obtain a padding oracle [2]. Moreover, when the target is a constant secret string repeatedly sent in the encrypted channel, it does not matter that the TLS session is closed and that the keying material changes accross sessions. The proof of concept was named Lucky 13, after the size of the pseudo-header MAC'ed with the message.

[6]To be precise, there is a 2^{-16} probability to get a plaintext ending with 01 01, a 2^{-24} to get 02 02 02, and so on. To eliminate those false positives, the attacker can simply repeat the operation with a different random string in $C^\star$.

Hypotheses and prerequisites

- The connexion uses CBC with a timing info-leak;
- The attacker is able to intercept and modify packets;
- Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Add random delays to CBC-mode decryption step. This would only increase the complexity of the attack (requiring more samples to compute the mean value).

Implement constant-time MAC-then-Encrypt. Such implementations counter the attack, but the code needed to obtain efficient record processing in effectively constant time is complex (the OpenSSL patch is almost 300 lines long).

Use RC4 to avoid CBC mode. As for BEAST, it thwarts *this* attack but is inconsistent with other measures.

Use TLS 1.2 AEAD suites. As for BEAST, this works but is hard to deploy reliably.

Switch to Encrypt-then-MAC. This would solve the problem, but the specification [16] is still young. Moreover, it relies on an extension, which would lead to the same deployment and reliability issues as TLS 1.2.

2.4 RC4 biases

RC4 is a stream cipher designed by Rivest in 1987. It is very simple to implement and has very good performance in software. It has thus been widely adopted in protocols (WiFi encryption protocols WEP and WPA, or TLS for example). Since 1995, several statistical biases have been identified in the first bytes of an RC4 keystream. These flaws eventually led to very efficient attacks against WEP [31].

As these attacks rely on initial biases of the keystream, it was proposed to drop the first n bytes of the keystream, but later findings show the existence of additional statistical biases, even after the initial bytes [12]. In 2013, two research teams presented practical attacks against the encryption of the same fixed sequence of plaintext using large numbers of different keys [18, 1], which apply to HTTPS cookies.

Here is a short description of the most efficient attacks on RC4, presented in the article by AlFardan et al. Their single-byte bias attack relies on the fact that the first 256 bytes of the keystream are strongly biased. The researchers generated a lot of RC4 keystreams to observe the actual distribution of each of the 256 first bytes. Using an empirical reference of 2^{45} keystreams, it is possible to recover the first 256 bytes of a plaintext, as soon as it is encrypted a sufficient number of times; this number varies from 2^{24} to 2^{32} as a function of the byte position in the 256 bytes keystream. Using the reference distribution and the encrypted distribution, the idea is to find the most probable byte value, by measuring the distance between the reference distribution and each of the candidate keystreams.

However, the attack is hard to implement, since it requires a lot of different TLS connections, and only works for data sent in the first few bytes. To overcome these limitations, the researchers also used long-term biases described by Fluhrer and McGrew [12] on consecutive bytes to perform a practical attack requiring more keystream, but which could work in a pipelined HTTPS stream (i.e. using different messages within the same TLS connection). This double-byte bias attack is more practical than the single-byte bias one, and a proof of concept was developed to recover an HTTP cookie.

Hypotheses and prerequisites

- TLS uses RC4 to encrypt data;
- The attacker is able to observe encrypted packets;
- Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Use CBC mode to avoid RC4. This is an efficient way to counter *this* attack, and it can be deployed easily and reliably. Obviously, this recommendation is in contradiction with the *Use RC4* recommendation (from Sec. 2.1 and 2.3).

Use TLS 1.2 AEAD suites. This works but leads to deployment and reliability issues.

Use another streamcipher. ChaCha20 [5] is currently under examination by the IETF as an alternative stream cipher. Such a change could be easier to deploy than a protocol version switch, as the ciphersuite negotiation is usually better supported, but it will still require some time[7].

Throw away the first bytes of the keystream. This behavior could be specified in TLS (with a new ciphersuite or extension) or HTTP (by padding the beginning of messages), but we know exploitable long-term RC4 biases exist.

Randomize the packet length. As for compression attacks, random padding would only increase the complexity.

2.5 POODLE: another padding oracle

In October 2014, Möller, Duong and Kotowicz presented POODLE (Padding Oracle on Downgraded Legacy Encryption) [22], another padding oracle targeting SSLv3 CBC mode. The old SSL version indeed handles CBC padding in a specific way: when n bytes are needed to pad a plaintext, the last byte is set to $n-1$ (as in TLS, described in Sec. 2.3), but the other bytes can take any arbitrary value. An attacker can use this liberty to get a padding oracle.

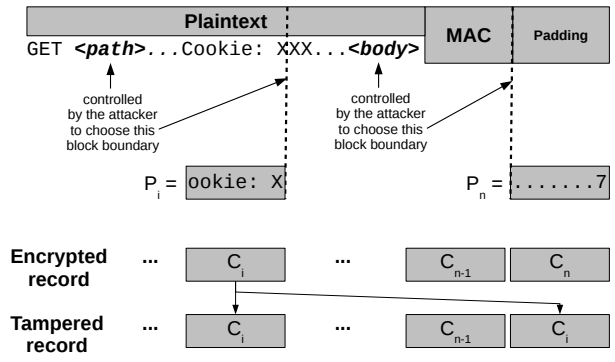

Figure 4: POODLE attack exploiting SSLv3 CBC Padding, assuming an 8-byte blockcipher.

Let's assume an attacker can trigger requests to the vulnerable site using SSLv3 and CBC mode. Since she may alter the path fragment in the URL, she may prepare the request in such a way that the cookie ends on a block frontier. Moreover, she may include a request body of arbitrary length after the headers, which allows her to get a plaintext message (once the MAC is appended) whose length is a multiple of the block size (see Fig 4). This way, a whole block would be added in the padding phase. Such a block has only one constraint: the last byte must be $n-1$ where n is the block length.

Once the request is sent by the browser, the attacker needs to modify the record on the wire. She must replace the all-padding block by the block where the last byte is to be guessed, as shown in Fig. 4. If the decryption of the last blocks leads to the correct value $(n-1)$, the rest of the block is ignored and the record is accepted by the server. It means the last byte of $E_k^{-1}(C_i) \oplus C_{n-1}$ is $n-1$, and that the last byte of P_i is $C_{n-1} \oplus C_{i-1} \oplus (n-1)$. If the padding does not end with $n-1$, the decryption will need to a MAC error and to the end of the connection. If the attacker retries, another key will be used and C_{n-1} will be randomized. Thus, each byte can be guessed with a 2^{-8} probability, which results in 256 requests needed to recover each byte of the secret value.

It is interesting to note that this attack relies on the browser using a fallback strategy, and on the server to accept the obsolete SSLv3 version of the protocol.

Hypotheses and prerequisites

- The connection uses SSLv3 with CBC mode;
- Plaintext can be loosely controlled by the attacker;
- The attacker is able to intercept and modify packets;
- Multiple connections containing the same secret can be triggered.

Proposed countermeasures

Use TLS 1.0. Since this powerful padding oracle is only present in SSLv3, forbidding this deprecated version is an efficient countermeasure. Moreover, only a small portion of the internet still relies on this version of the protocol, which makes this measure also practical.

Use RC4 to avoid CBC mode.

Use TLS 1.2 AEAD suites.

Switch to Encrypt-then-MAC. These three countermeasures work, with the same reservations as before.

Anti poodle record splitting. Opera and Google's developper Adam Langley proposed to split SSLv3 CBC records to counter the POODLE attack. The proposed splitting method is supposed to avoid whole blocks of padding. Yet, POODLE paved the way for new SSLv3 padding oracle attacks, which may not be blocked this way.

TLS_FALLBACK_SCSV. Möller and Langley proposed a mechanism to avoid browser fallbacks when a higher version is supported by both the client and the server, using a fake signaling ciphersuite. This would indeed block downgrade attacks and POODLE in particular, between up-to-date parties. Yet, legacy SSLv3 stacks would still be at risk.

2.6 Comparative analysis of these attacks

To mitigate these threats, many countermeasures have been proposed. Our analysis shows that some of them have no real effect on the attacks: throwing the first bytes of RC4 keystream, randomizing the packet length or adding random

[7]Moreover, the current draft actually specifies an AEAD suite, not a streamcipher one, which would require TLS 1.2.

Countermeasures	Dep.	Rel.	HTTP	Beast	L 13	RC4	*IME	POODLE
Structural changes to TLS								
Use TLS 1.0	+	+	+					+
Use TLS 1.1	−	−	+	+				+
Encrypt-then-MAC	− −	−	+		+			
Changes related to TLS ciphersuites or compression methods								
Use CBC mode	+	+	+			+		
Use RC4	+	+	+	+	+			+
Use a new stream cipher	−	+	+	+	+	+		+
Use AEAD (TLS 1.2)	− −	−	+	+	+	+		+
No TLS compression	+	+	+				+	
Changes related to TLS implementations								
$1/n-1$ split	−	+	+	+				
Constant-time CBC	−	+	+		+			
Anti poodle splitting	−	+	+					+
Other countermeasures in related work								
Single-use cookie	− −	+	−	+		+	+	+
One-Time Cookies [7]	− −	−	+	p	p	p	p	p
Countermeasures presented in this article								
TLS scrambling	− −	+	+	+	+	+	+	+
MCookies (server)	−	+	+	+		+	+	+
MCookies (client/server)	− −	+	+	+	+	+	+	+

Table 1: Summary of the proposed countermeasures.

delays to CBC-mode decryption. Others require significant changes to the architecture of web applications and would be hard to enforce: restricting cross-site requests or fixing all SOP/XSS bugs. The remaining countermeasures are listed in Table 1, and compared using different criteria:

- **Dep.** relates to the ease of **deployment** of the proposed solution. In particular, studies [24, 19] shed light on the problem of existing intolerance, e.g. new TLS versions raising important compatibility issues;
- **Reliability** (**Rel.**) corresponds to the assurance we have the countermeasure will not be easily bypassed between a client and a server both implementing the solution. The idea is to capture the possible down-negotiation and fall-back strategies (for example issues related to TLS version negotiation);
- **HTTP** assesses the compatibility of the measure with HTTP use-case. Current web applications have to e.g. accommodate with multi-tab browsing. Countermeasures should not break or limit such features;
- A set of columns state whether the countermeasure is efficient against each attack (**BEAST**, **L13**, **RC4**, ***IME** for compression attacks, and **POODLE**), "p" meaning the measure only partially blocks the attack.

The last lines of the table describe countermeasures proposed in related work (Sec. 6) and our proposals (Sec. 4).

3. ATTACKER MODEL AND THE MASKING PRINCIPLE

The legitimate actors we consider are: the user agent (e.g. Firefox), the HTTP(S) server (e.g. Apache) and the web application (a program written in PHP or Python for example). The web application may rely on a framework designed to abstract the inherent complexity of web development (e.g. Django or Zend). The attacker we consider is an active network attacker, able to read, modify or delete packets between the client and the server. We also assume, as for each of the attacks presented in Sec. 2, that a secret cookie is repeated across different TLS messages.

Given a TLS session, we assume the attacker is able to retrieve some information about κ consecutive bytes of the corresponding plaintext. Typically, $\kappa = 1$, and the attacker is able to check whether a cleartext byte is equal to a guessed value. Thus, by repeating the attack on constant plaintext bytes, she can recover this part of the plaintext.

To draw a parallel with side-channel attacks [6, 15, 26], such attacks may be called first order attacks. To deal with them, a typical countermeasure is to mask the secret value: each time a secret s of κ bytes must be transmitted, a random value m (the mask) of the same length would be chosen and the pair $(m, m \oplus s)$ would be sent instead of s. This way, the value can trivially be recomputed by the other party, but the representation on the wire is different for every message. If the secret s to mask is longer than κ, s can be split in κ-byte words, masked by the same mask. Alternatively, it is possible to choose a longer mask to cover the secret entirely.

Masking all secrets using a fresh mask would force the attacker to mount a second order attack, that is find a way to simultaneously retrieve information about the mask and the masked value, to learn something about the secret.

Most of the attacks described in Sec. 2 are designed to recover the plaintext one byte at a time, which makes them first order attacks with $\kappa = 1$. Some attacks against RC4 also exploit statistical biases on two consecutive bytes, which also corresponds to our model (for $\kappa = 2$).

In practice, the BEAST, CRIME, TIME and Lucky 13 attacks could easily be extended to guess κ consecutive bytes at once (for example, for BEAST, this would mean aligning the boundary of the block to guess differently). However, the

complexity to recover κ bytes at once would be proportional to $2^{8\kappa}$ (instead of $\kappa \cdot 2^8$), which limits κ to small values in practice. To be conservative, we consider the maximum number of recoverable successive bytes κ to be 8.

Overall, the recent attacks against TLS Record layer can all be considered as first order attacks (with $\kappa \leq 8$). So, masking secret values using unique 8-byte random strings will mitigate these attacks. In the following sections we present two implementations of this concept applied to the transport and the application level.

Concerning RC4, it can be noted that other known biases exist and are related to distinct distant groups of keystream bytes. Thus, second order attacks against RC4 might be possible by exploiting such biases. We briefly discuss this case in the conclusion.

4. PROPOSED MECHANISMS

This section presents generic mechanisms to mitigate the impact of TLS security flaws, by leveraging the masking principle. The first one acts at the transport (TLS) layer, while the second one works at the application (HTTP) level.

4.1 TLS Scramble: Masking at the TLS level

The idea of masking application data at the transport level is not new in TLS. During the specification of WebSockets [11], a recent HTML5 feature, a randomization step was added to avoid confusion between WebSocket traffic and other protocols, that could be leveraged by an attacker. WebSocket randomizes client-to-server traffic using 4-byte long masks. An interesting side effect of this change was to block the early version of the BEAST attack, forcing Duong and Rizzo to rewrite their exploit using Java instead of WebSockets.

4.1.1 A fake compression method: Scramble

As shown in Fig. 1, record processing may optionally compress the plaintext before the cryptographic transformations. This step takes a plaintext record of at most 2^{14} bytes, and produces a compressed record that can be at most 1024 bytes longer than the plaintext.

To generalize the idea behind WebSockets masking, we define a *fake* compression algorithm, Scramble. Given a κ parameter (the mask length) and a plaintext P, the way Scramble *compresses* P is as follows:

- the `scramble_record` method generates a κ-byte random string m;
- m is repeated, and possibly truncated, to be as long as P. The result is a masking string M;
- the *compressed* record is $m|P \oplus M$, which is exactly κ-byte longer than P.

The `unscramble_record` operation is straightforward:

- on receiving a *compressed* string c, which should contain at least κ bytes, extract the first κ bytes of c as the mask m, and call X the remaining string;
- expand m to be as long as X to obtain M;
- the *uncompressed* value is $M \oplus X$.

4.1.2 Implementation in OpenSSL

To check the feasibility of this idea, we implemented the Scramble compression method in OpenSSL (v1.0.1) with 8-byte masks. The patch affects the `crypto/comp` directory. It adds `c_scramble.c`, a 75-line file describing the method, as well as trivial changes to `comp.h` and to the corresponding `Makefile`. The `scramble_record` function is given in Appendix C. To test the method with real connections, we also patched `apps/s_client.c` and `apps/s_server.c` to exchange data over the scrambled channel.

4.2 MCookies: Masking at the application level

Another way to tackle the problem is to mask secret values at the application level, which requires less bandwidth (only relevant elements would need to be masked) and avoids modifying TLS stacks. In this section, we propose a method to mask cookies at the HTTP level.

4.2.1 MCookies principle

Usually, HTTP cookies work as specified in Fig. 5: a server can define cookies to be stored by its client, then each time this client sends a request to the server, the cookies are added to the headers [3]. In some cases, they may also be read and modified by client-side scripts.

If we only consider cookies that are never read nor modified by client-side scripts, there is a simple way to break this repetition while modifying only the HTTP server, which is described in Fig. 6:

- **Cookie definition:** when the web application sets such a cookie (e.g. calling `set-cookie(SESSID,` V`)`), the HTTP layer rewrites the `Set-cookie` header to send `SESSID=`$M:M \oplus V$ instead of `SESSID=`V[8].
- **Cookie restitution (and redefinition):** for each request containing a `SESSID=`$X:Y$ cookie, the HTTP server transmits `SESSID=`$X \oplus Y$ (the unmasked cookie) to the web application. Then, three cases may arise:
  - the web application updates the cookie, which is covered by the Cookie definition step;
  - it can erase the cookie by setting an outdated expiration time, in which case the HTTP layer simply transmits the header as is;
  - otherwise (the cookie is left unchanged by the application), the HTTP server sets a new version of the cookie, M':$M' \oplus V$, that is the same initial value masked using a fresh random mask.

4.2.2 Discussion of MCookies feasibility

To select the senstive cookies to protect, a simple way would be to define a static list of cookie names, but a natural heuristic is to protect every cookie flagged both `httpOnly` and `secure`. Only considering `httpOnly` cookies guarantees that client-side script have no access to the cookie value, leaving the HTTP server free to change the cookie representation at will. However, from the web application point of view, the cookie value sent and received remains the same. Moreover, protecting a non-`secure` cookie is pointless as this one can be easily stolen with our attacker model in general[9].

[8]Since M and $M \oplus V$ are binary strings, Base64 is used.

[9]Even when cookies are sent without the `secure` attribute, security mechanisms like HSTS (HTTP Strict Transport Security [17]) can forbid cleartext communications.

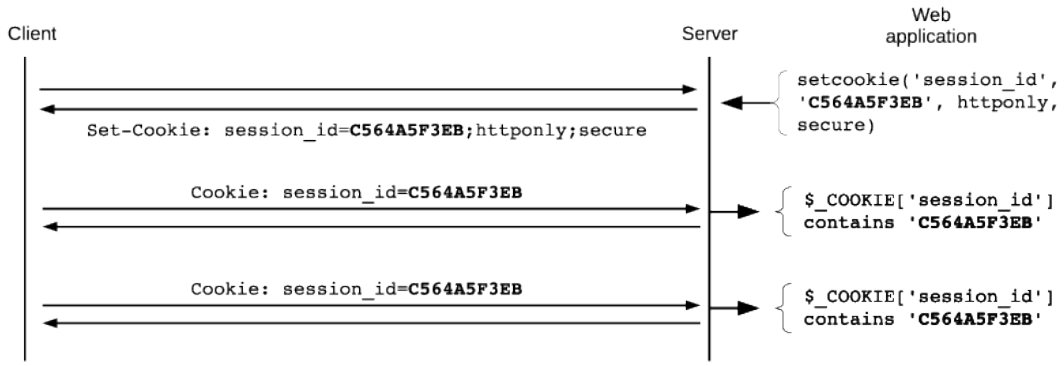

Figure 5: HTTP cookies: definition and restitution.

Figure 6: Definition and restitution of MCookies

Yet, rewriting the cookies on every request has negative consequences. First, it adds extra-bandwidth in server messages. Second, the original cookie attributes (`Expires`, `Max-Age`, `Domain`, `Path`) are lost in the process. These attributes would need to be specified at each redefinition to keep application cookies constistency. To fix this problem, the attributes from the original `Set-cookie` header are encoded inside the masked cookie: a sensitive cookie V with attributes A would be transmitted to the client as $M : M \oplus V : A$. It would thus be possible to remember the correct attributes for each request.

By fixing the attribute problem, we amplified the bandwidth overhead. However, we can use smaller representations for several attributes, since each request carries information that can help rebuild the `Domain` and `Path` attributes. For domains, we can keep only the number of subdomains in the domain and the presence of a starting dot. For example, the `.example.com` domain would become `.2`, whereas `sub.example.com` would become `3`. Indeed, the exact domain can be rebuilt using the `Host` header in the request. Therefore, we only need a single byte to encode domains, using the sign bit to store the presence of the starting dot, leaving 7 bits for the node count. A similar transformation can be applied to path attributes. Finally, the expiration attribute can be converted into a 8-byte timestamp.

Another drawback of MCookies is that they cannot prevent active network attacks (such as Lucky 13). Indeed, when the record packets are modified by the attacker, they are seen as corrupted records by the server TLS stack, and discarded. Since the HTTP server never receives the corresponding request, it cannot answer with a freshly masked cookie. To counter active attacks, the browser could monitor failed successive HTTPS connections for each origin and, after a given number of broken connections, erase the cookies associated to this domain. Drawbacks of this countermeasure are twofold:

- the client has to be modified to maintain this counter;
- setting a correct threshold is hard: the trigger should be effective against real attacks, but a low threshold would easily break HTTPS sessions on poor quality network connections.

To overcome MCookies limitations, we extend MCookies with a new HTTP header: `Masked-Cookie`. This extension requires a change in the browser that is now put in charge of masking cookies.

4.2.3 *Masked-Cookie headers*

In addition to the MCookies mechanism, we introduce a new `Set-Cookie` attribute, `masked` to signal to the client the presence of masking. Then, a compliant client would, for each request, send this cookie in a new header, `Masked-Cookie`, with the value here masked by the client : `Masked-`

Figure 7: Use of `Masked-Cookie` headers: masking is done by the client.

`Cookie: SESSID=`$M' : M' \oplus V$. Fig. 7 describes the protocol use of this new header.

By moving the cookie masking process to the client side, active attacks are no more effective. Faced with a compliant client, a server only needs to define MCookies once, so the extra bandwidth cost is essentially removed. On the other hand, a standard client will ignore the `masked` attribute, and the server will fall back on the previous behavior.

4.2.4 Apache Implementation

Implementing MCookies can easily be done as a filter module for HTTP servers. We chose Apache for our proof of concept, since it is open-source, and currently the most deployed HTTP server (its market share is estimated at 45 % by Netcraft[10] and at 65 % by W3Techs[11]).

Apache exposes a powerful module system with hooks allowing to interact with the request and response processing. In order to mask the cookies sent by the server during the emission of the `Set-Cookie` header, we hook the response process using an `output_filter`, defined by `mod_filter`. To this aim, we call the `ap_register_output_filter` and `ap_add_output_filter_handle` functions. Then, to unmask the cookies received via `Cookies` headers from the client, we hook the request handling. The `input_filter` hook happening too late in the process, we use an earlier control point, when headers are parsed, using `ap_hook_header_parser`. The last step is to send a new representation of the cookie parsed in the request, along with the response; this step is easy to implement since request cookies are available from the `output_filter` hooks.

The overall code to implement the MCookies (including the Base64 code to encode the masked value safely) is around 500 lines of C. It handles both the MCookie rewriting, without compression, and the `Masked-Cookie` extension.

4.2.5 Masked-Cookies for Chromium

Chromium is currently the most popular web browser. As it is open-source and modular, we decided to patch this web browser to prove the feasibility of `Masked-Cookies` headers in a real world context. The overall C++ patch for Chromium (version 31) only counts 241 lines. It adds the `masked` attribute to the internal cookie representation, the `CanonicalCookie` class, as a new attribute. Masking and unmasking are implemented in the `CanonicalCookie::Create` and `CookieMonster::BuildCookieLine` methods.

[10] `http://www.netcraft.com`
[11] `http://w3techs.com`

5. ANALYSIS OF MASKING MECHANISMS

5.1 Security analysis

5.1.1 TLS Scramble method

Specified as a new TLS compression method, TLS Scramble would require deployment efforts. However, once deployed, TLS compression negotiation would be reliable, since it is similar to ciphersuite negotiation, which is known to work between all TLS stacks.

TLS Scramble disables real TLS compression, trivialy defeating CRIME and TIME attacks. This technique is also efficient against BEAST, Lucky 13, RC4-biases and POODLE attacks, since they only recover the secret one byte at a time, and that Scramble makes this byte a moving target.

It is however important to notice that TLS Scrambling masks the entire messages, and not only the secret values, which means this method does not meet the principle described in section 3 *per se*. In particular, some attacks relying on mixing secret values and attacker's guess might still work, e.g. compression attacks in the application layer.

5.1.2 MCookies

MCookies randomize only the cookie values sent by the client, which fits exactly our masking principle. As long as secrets are masked, first-order attacks will be defeated by MCookies: this is the case for passive network attacks like BEAST, client-side compression attacks and RC4-biases.

To be effective, MCookies need HTTP requests to reach the server and the corresponding answers (containing a freshly masked cookie) to get back to the client. Active network attackers may be able to block such answers. Even if they do not, Lucky 13 can not be blocked by MCookies, since each tampered request will lead to a server-side TLS error, thus breaking the connection. For POODLE, a lucky guess will lead to a valid record, and the server will be able to send a new cookie representation, which may still be blocked by the attacker. So MCookies are inefficient against active attacks.

Apart from thwarting most of the attacks, MCookies have the advantage of requiring only a small modification of the HTTP server, leaving the browser and the web application untouched, which makes it a reliable solution to deploy.

5.1.3 MCookies with `Masked-Cookies` headers

As MCookies, this extended mechanism fits the masking principle. Moreover, since sensitive cookies are masked by the browser, every request will be masked differently, even in the presence of an active attacker: all the studied attacks are covered by the countermeasure.

Moreover, this mechanism is backward compatible: it is possible with the same HTTPS server to handle old and new clients, taking advantage of the new `Masked-Cookies` headers when available, but still defeating passive network attacks with older clients.

5.2 Performance analysis

5.2.1 TLS Scrambling overhead analysis

Masking plaintext data at the TLS level is easy to implement, and is completely transparent from the application layer point of view. Yet, it presents two major drawbacks:

- each and every record has to be κ-byte longer, even for short messages;
- deploying a new compression method in TLS would be hard, since deploying new ciphersuites (an easier operation) can take years.

Performance-wise, the CPU overhead is negligible, and the network bandwidth overhead is less than 1 % in bytes.

5.2.2 MCookies network overhead analysis

At application level, the masking of cookie values increases the HTTP requests and responses size by adding the mask and the attributes to the initial cookie value and by sending new headers. In order to quantify this overhead, we built a script that simulates MCookies on a real internet navigation traffic. It computes the overheads according to whether the web browser used supports `Masked-Cookies` headers or not.

Traffic type	**Raw traffic volume**	**Extra bandwidth**		
		w/o UA support		with UA support
		naive	compr.	
Sensitive	24 MB	+20.1 %	+14.9 %	+10.8 %
Overall	122 MB	+4.1 %	+3.0 %	+2.2 %

Table 2: Network overhead evalulation

We instrumented a web browser (Chromium) to analyse HTTP requests and responses from HTTPS secured traffic obtained during the following one day scenario. The user logs in Google, Facebook, Twitter, Dropbox, an RSS aggregator, and some other web sites; he keeps tabs opened on multiple pages for each services, uses them during the day and browses other websites as well. 8,185 HTTP requests (122 MB) were retrieved. Among these, 4,823 requests (24 MB) actually set or sent sensitives cookies.

With this HTTP trace, the script processes sequentially each request and simulates the overheads induced by installing the MCookie module on every reached host. We identify every sensitive cookie by looking at the `httpOnly` and `secure` flags. Table 2 describes the extra bandwidth in different situations: without User-Agent support (that is using MCookies), either with naïve or compressed encoding, and with User-Agent support (the `Masked-Cookies` headers). The results show a significant overhead on requests containing sensitive cookies. However, when looking at the big picture, the overall HTTPS traffic, this overhead turns out to be rather small. Finally, compared to the entire web communications, both HTTP and HTTPS, the cost induced by cookie masking proves to be negligible.

5.2.3 Apache module system overhead analysis

To evaluate the efficiency of MCookies and assess their scalability, we performed a web server benchmark with and without the MCookies module enabled. In order to evaluate the module overhead, we ran the benchmarking tool on a single HTML page. Furthermore, we also benchmarked the module on a Wordpress website for a more realistic scenario. Each request sent embedded three different sentitive cookies.

The host used for this evaluation was an Intel Xeon X5650 with 6Go of RAM running a Debian system with an Apache (v2.4.7) web server, a Mysql (v5.5) database server and hosting a Wordpress (v3.8.1) web site.

	Vanilla server	**MCookies enabled**	
		w/o UA support	with UA support
Static page	384	318 (-17 %)	382
Wordpress page	221	212 (-4 %)	220

Table 3: Performance results (transactions/second)

We used Siege[12] in benchmark mode to assess the number of transactions the web server is capable to process per second with the three scenarios. The results, as described in Table 3, show a small decrease of 4 % of the Wordpress web server capacity when dealing with User-Agents without the support of this mechanism, whereas the overhead is negligible otherwise. However, for a static page served to a User-Agent with no `Masked-Cookie` header support, performance are much more degraded, but this is a worst-case scenario, since static web sites rarely produce sensitive cookies.

6. RELATED WORK

To avoid repeating the same cookie across different TLS messages, a natural idea would be to change its value for every new connection, or at least to limit the cookie lifetime. In fact, PHP proposes a way to handle session identifiers this way with the `session_regenerate_id` function, usually called after a user logged in, to decorrelate the old and the new sessions and avoid session fixation attacks[13]. Short-lived cookies (which could even be pushed to single-use cookies) should thwart all passive attacks, but choosing the right lifetime is not easy. In fact, mitigating attacks would require a very short lifetime, which could easily lead to out-of-sync cookies when dealing with parallel HTTP connections. Modern web sites heavily use JavaScript asynchronous requests, and session regenerations are known to provoke requests concurrency errors[14]. This is why MCookies are designed to always have the same *intended* value.

One-Time Cookies [7] are another solution to protect plaintext HTTP cookies against replay, by having the client bind

[12] `http://www.joedog.org/siege-home/`

[13] `http://www.acros.si/papers/session_fixation.pdf`

[14] Our 1-day HTTPS capture actually contains such concurrent requests, that would be problematic.

the cookie with the request sent. This mechanism uses cryptographic mechanisms (symmetric encryption and HMAC) and borrows the idea of Kerberos tickets and proposes an elegant solution requiring no server-side state. Applied to HTTPS and our attacker model, One-Time Cookies (OTC) do not counter attacks, since repeating the same exact request would lead to the same OTC; however, the value retrieved could only be replayed for the request in question, which would limit the scope of the attacks. Moreover, as OTC rely on specific HTTP headers, their implementation requires browser and server/web application modifications. With regard to the Record protocol attacks exposed here, `Masked-Cookie` headers are much simpler to implement (no cryptographic primitives are needed).

Both solutions (single-use cookies and One-Time Cookies) are described in Table 1. Other alternative cookie protocols have been proposed, such as [14, 20], but they share OTC advantage (unique cookies bound to the request data) and drawbacks (heavy changes needed on both end points).

7. CONCLUSION

We have studied recent attacks on TLS Record protocol, and thoroughly analyzed the proposed countermeasures. In practice, the countermeasures implemented in most of the software are specific to each attack: $1/n-1$ split for BEAST, constant-time CBC decryption for Lucky 13, deprecation of RC4, disabling TLS compression for CRIME and TIME, and deprecation of SSLv3 for POODLE.

In parallel, we showed that all the attacks relied on the common assumption that a secret would be repeatedly sent in different TLS sessions. We suggested a common model to describe these attacks. We also proposed to reuse the concept of masking, borrowed from the side-channel community, to mitigate the attacks. Such a technique can be implemented as a complementary measure, a defense-in-depth strategy. We described different ways this countermeasure could be implemented, and wrote two proofs of concept to check its feasibility to protect cookies.

At the TLS level, our Scramble compression method builds on the idea of WebSockets masking, which actually blocked the BEAST attack. At the HTTP level, our MCookies extend the idea of single-use cookies, without requiring complex changes in web protocols and applications. Masking allows for a defense-in-depth strategy, giving developers and integrators more time to solve the crisis. It would have been effective against the presented attacks, and might be against yet unknown ones. The recent POODLE attack did in fact meet all the criteria, and was published after we implemented our proposals. The fact it would have been blocked by our proofs of concept actually validates our work.

We would however make it clear that masking is designed to be a defense-in-depth measure *in addition* to specific countermeasures, not instead of them. When a cryptographic algorithm or scheme shows significant weaknesses, they should be phased out and correctly patched. In the particular case of RC4, we now know a lot of statistical biases, some of which can lead to efficient first order attacks, but realistic second order attacks could be the next attack against TLS Record layer. All the masking proposals could be easily extended to use two (or three) masks instead of one. Yet we consider RC4 is a good example of a streamcipher that should have been phased out a long time ago, since many RC4 practical and theoretical flaws have been known for a decade.

8. REFERENCES

[1] N. J. AlFardan, D. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the security of RC4 in TLS and WPA. In *USENIX Security*, 2013.

[2] N. J. AlFardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE SSP*, 2013.

[3] A. Barth. HTTP State Management Mechanism. RFC 6265, 2011.

[4] A. Barth. The Web Origin Concept. RFC 6454, 2011.

[5] D. Bernstein. ChaCha, a variant of Salsa20. cr.yp.to/papers.html#chacha, 2008.

[6] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, 1999.

[7] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Trans. Internet Techn.*, 2012.

[8] T. Dierks and E. Rescorla. TLS Protocol Version 1.1. RFC 4346, 2006.

[9] T. Dierks and E. Rescorla. TLS Protocol Version 1.2. RFC 5246, 2008.

[10] T. Duong and J. Rizzo. BEAST: Surprising crypto attack against HTTPS. Ekoparty, 2011.

[11] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, 2011.

[12] S. Fluhrer and D. McGrew. Statistical Analysis of the Alleged RC4 Keystream Generator. In *FSE*, 2000.

[13] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, 1999.

[14] K. Fu, E. Sit, K. Smith, and N. Feamster. The Dos and Don'ts of Client Authentication on the Web. In *USENIX Security*, 2001.

[15] L. Goubin and J. Patarin. DES and Differential Power Analysis The "Duplication" Method. In *CHES*, 1999.

[16] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366, 2014.

[17] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797, 2012.

[18] T. Isobe, T. Ohigashi, Y. Waatanabe, and M. Morii. Full Plaintext Recovery Attack on Broadcast RC4. In *FSE*, 2013.

[19] O. Levillain, A. Ebalard, H. Debar, and B Morin. One Year of SSL Measurement. In *ACSAC*, 2012.

[20] A. X. Liu, J. M. Kovacs, C. Huang, and M. G. Gouda. A Secure Cookie Protocol. In *IEEE ICCCN*, 2005.

[21] B. Möller. Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures, 2002-2004.

[22] B. Möller, T. Duong, and K. Kotowicz. Google Security Advisory: This POODLE Bites: Exploiting The SSL 3.0 Fallback, 2014.

[23] K. G. Paterson and N. J. AlFardan. Plaintext Recovery Attacks Against DTLS. In *NDSS*, 2012.

[24] Y. N. Pettersen. Renego patched servers: A long-term interoperability time bomb brewing. My Opera blog: Implementer's notes, 2010.

[25] A. Prado, N. Harris, and Y. Gluck. SSL, Gone in 30 seconds - A BREACH beyond CRIME. Black Hat USA, 2013.
[26] E. Prouff and M. Rivain. Masking against side-channel attacks: a formal security proof. In *Eurocrypt*, 2013.
[27] E. Rescorla and N. Modadugu. DTLS Version 1.2. RFC 6347, 2012.
[28] I. Ristic. Internet SSL Survey, Talk at BlackHat 2010. Black Hat USA, 2010.
[29] J. Rizzo and T. Duong. The CRIME attack. Ekoparty, 2012.
[30] P. Rogaway. IETF Draft: Problems with proposed IP Cryptography, 1995.
[31] A. Stubblefield, J. Ioannidis, and A. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. In *NDSS*, 2002.
[32] A. Shulman T. Be'ery. A Perfect CRIME? TIME Will Tell. Black Hat EU, 2013.
[33] S. Vaudenay. Security Flaws Induced by CBC Padding Applications to SSL, IPsec, WTLS. In *Eurocrypt*, 2002.

APPENDIX

A. HTTP BASIC/DIGEST AUTH

In this paper, we discussed cookie protection. Other methods exist to authenticate the client at the HTTP layer: Basic and Digest Authentication [13]. In practice, they are rarely used, as the user interfaces do not allow for a clean integration with modern web applications (e.g. no logout feature).

Both headers are as vulnerable as cookies w.r.t. the described attacks. For Basic Authentication, it is not clear how they could be randomized: there can be no equivalent to MCookies, unless the mechanism is deeply changed. For Digest Authentication however, the standard already allows for possible randomizations resembling MCookies, producing a new server nonce value for each new request. Such policy would defeat all passive network attacks, since the client would produce a different header for each request.

B. SERVER-SIDE COMPRESSION

B.1 TIME and BREACH

In the TIME attack, in addition to targeting client-side compression, the researchers proposed to target server-side secret information repeatedly sent on the wire. For example, Cross-Site Request Forgeries (CSRF) are usually blocked by having the server insert a token in forms, which is later checked on form submission. Such anti-CSRF tokens are usually reused for a given user and a certain amount of time.

Server-side messages can be compressed using two different mechanisms. In addition to the aforementioned TLS compression, the server can also use HTTP compression. This HTTP compression is also a target of the attack, as the attacker can inject attacker-controlled data in the server answer (preferably close to the targeted token in the payload). This is the principle of the BREACH attack [25], an attack aiming at retrieving the anti-CSRF token sent by the server.

Hypotheses and prerequisites

- TLS or HTTP compression is activated;
- The ciphertext length is observable;
- The answer containing the target token is partially controlled by the attacker, e.g. using a reflected field;
- Multiple connections containing the secret can be triggered by the attacker.

Proposed countermeasures

Disable TLS and HTTP compression. As for client-side compression, this measure blocks the attack. In fact, TLS compression can be (and has been) disabled, but HTTP compression is essential to reduce bandwidth and disabling it would drastically increase the size of HTTP responses.

Randomize the packet length. See Sec. 2.2.

Structurally modify web applications to separate secrets from attacker-controlled content. If sensitive information and attacker-controlled content come from different servers, compression contexts are distinct and the attack does not hold. Yet, this requires significant changes.

Change the token value for each request (single-use token). It would block the attack but it also requires an important change in the way anti-CSRF tokens usually work.

B.2 MTokens: Making anti-CSRF tokens

To mitigate server-side attacks, the CSRF tokens (and similar objects) could easily be protected using a technique close to MCookies, i.e. by masking the token with a different value for each message. The *intended* value of the token would remain the same, avoiding out-of-sync problems, while randomizing the data that is sent over the network. Implementing this would require very very small changes to web applications (or even no change at all if web frameworks are modified): it would simply amount to replacing every call to the function producing the token (which we will call `write_csrftoken()`) with `mask(write_csrftoken())`, and each call to the function getting the token from the client form (`read_csrftoken()`) with `unmask(read_csrftoken())`.

Security analysis

MTokens require small modifications in web applications (or in frameworks), so they are easy to deploy, reliable and compatible with web applications, especially if the changes are made in the framework. They are effective against server-side first order attacks against anti-CSRF tokens in general.

C. TLS SCRAMBLING FUNCTION

The core of the Scramble compression method is the `scramble_record` function (and `unscramble_record`, its counterpart), which is implemented as follows

```
static int scramble_record(COMP_CTX *ctx,
  uchar *out, uint olen, uchar *in, uint ilen)
{
  uchar mask[MSIZE];

  if (olen < (ilen + MSIZE)) return -1;
  if (RAND()->bytes(mask, MSIZE) < 0) return -1;
  memcpy (out, &mask, MSIZE);

  out = out + MSIZE;
  for (int i = 0; i < ilen; i++)
    *(out++) = *(in++) ^ mask.bytes[i % MSIZE];
  return (ilen + MSIZE);
}
```