(In-)Security of Cookies in HTTPS: Cookie Theft by Removing Cookie Flags

Hyunsoo Kwon, Hyunjae Nam, Sangtae Lee, Changhee Hahn, and Junbeom Hur, *Member, IEEE*

***Abstract*—HyperText Transfer Protocol (HTTP) cookies are widely used on the web to enhance communication efficiency between a client and a server by storing stateful information. However, cookies may contain private and sensitive information about users. Thus, in order to guarantee the security of cookies, most web browsers and servers support not only Transport Layer Security (TLS) but also other mechanisms such as HTTP Strict Transport Security and cookie flags. However, a recent study has shown that it is possible to circumvent cookie flags in HTTPS by exploiting a vulnerability in HTTP software that allows message truncation.**

In this paper, we propose a novel cookie hijacking attack called $rotten\ cookie$ which deactivates cookie flags even if they are protected by TLS by exploiting a weakness in HTTP in terms of integrity checks. According to our investigation, all major browsers ignore uninterpretable sections of the header of HTTP response messages and accept incorrect formats without any rejection. We demonstrate that, when combined with TLS or application vulnerabilities, this form of attack can obtain private cookies by removing cookie flags. Thus, the attacker can impersonate a legitimate user in the eyes of the server when cookies are used as an authentication token. We prove the practicality of our attack by demonstrating that our attack can lead five major web browsers to accept a cookie without any cookie flags. We thus present a mitigation strategy for the transport layer to preserve cookie security against our attack.

***Index Terms*—Cookie theft attack, SSL/TLS, HyperText Transfer Protocol.**

I. Introduction

An HTTP cookie is a small piece of data sent from a server and stored by a user's browser to remember stateful information or record the user's browsing activity. In the modern web environment, HTTP cookies efficiently handle HTTP requests and responses by providing a diverse range of functions. In many cases, cookies contain private and unique user information (e.g., authentication tokens). A recent study [1] investigated HTTP servers world-wide to determine whether their cookies could leak private user information when hijacked. They found that most popular web servers, such as Google, YouTube and Amazon, leak private user information including user names, email addresses, and account functionality. Furthermore, some web applications use cookies as an authentication token, which allows a server to authenticate a user without his account ID or password. Users can thus easily log onto web pages using authentication cookies previously stored by their browser. If an adversary obtains a cookie, he can thus attempt to impersonate the user. Therefore, private cookies should be kept secret and transmitted over secure channels during the exchange between a client and a server.

The Transport Layer Security (TLS) [2] protocol is the most widely deployed security mechanism for HTTP messages and private cookie protection. However, HTTPS (HTTP over TLS) cannot fully guarantee the security of the information in a cookie because cookie mechanisms are independent of the TLS protocol. Specifically, web servers selectively activate TLS when mutual authentication is required or data confidentiality needs to be guaranteed because supporting TLS when providing web services to users is optional. Therefore, it is necessary to consider additional mechanisms to fully protect the information in a cookie. Typically, HTTP Strict Transport Security (HSTS) [3] and cookie flags are the most well-known methods for this purpose. In this paper, we focus on security concerns regarding cookie flags.

A cookie flag is a security mechanism that protects the data in a cookie. The server can define special properties for a cookie by appending flags in the `Set-Cookie` header option when delivering a cookie to a browser. There are two types of cookie flag appended to the `Set-Cookie` header: `HttpOnly` and `Secure` flags.

An `HttpOnly` flag ensures that a cookie cannot be accessed by client-side APIs such as JavaScript. This restriction eliminates the threat of cookie theft via cross-site scripting (XSS) [4]. A cookie with a `Secure` flag can only be transmitted over an encrypted connection such as TLS when the client sends it to the server later. It protects cookies against cookie theft via eavesdropping over HTTP. If the `Secure` flag is not properly set in the `Set-Cookie` header, a man-in-the-middle (MITM) attacker disguised as a legitimate HTTP server can obtain private cookies via HTTP.

Although HTTP cookies are protected via a secure HTTPS connection with the additional mechanisms explained above, another security breach has been discovered in which an attacker can make a cookie set without a cookie flag. In a cookie-cutter attack [5], even if cookies are encrypted by TLS, miscommunication between TLS and HTTP allows the attacker to remove a cookie flag simply by closing the connection. This is a serious vulnerability for most HTTPS software because it does not enforce proper TLS termination, allowing the attacker to truncate a message at any TLS-fragment boundary. Because the browser accepts truncated HTTP response messages, the attacker can force the cookie to be set without a cookie flag even if it is encrypted. Thus, a security threat may exist even when a cookie in HTTP is protected by another independent TLS protocol. In other words, HTTPS itself lacks a security

H. Kwon, H. Nam, S. Lee, C. Hahn, and J. Hur are with the Department of Computer Science and Engineering, Korea University, Seoul, 156-756, Rep. of Korea e-mail: hs_kwon, niceotor13, tkdxo0624, hahn850514, jbhur @korea.ac.kr

H. Kwon, H. Nam, and S. Lee contributed equally to this work mechanism to protect cookies.

Unfortunately, there is no specific method for checking the integrity of messages in HTTP for two main reasons. First, HTTP needs to maintain its flexibility and scalability. Therefore, when a browser receives an HTTP response message with uninterpretable sections, it discards those sections and accepts the rest. Second, HTTP depends on the transport layer (that is, TLS) for its security without any security mechanism for its own (application) layer.

However, the security of TLS is questionable. At present, TLS suffers from a number of vulnerabilities such as HeartBleed [6], SLOTH [7], DROWN [8], POODLE [9], FREAK [10], BEAST [11], Lucky Thirteen [12], and Logjam [13]. Weak cryptographic primitives, faulty implementation, and side-channels all threaten the security of TLS. Although some of these vulnerabilities have been removed from the latest specification, most servers still support them in practice to ensure backwards compatibility [13]–[15]. In addition, the practical threat of zero-day vulnerabilities, faulty implementation, and various side-channels are always a danger. For example, an attacker can forge a valid ciphertext if the server has faulty implementation in a TLS Advanced Encryption Standard in Galois/Counter Mode (AES-GCM) cipher [16].

In this paper, we propose a novel cookie theft attack scenario called $rotten\ cookie$. A $rotten\ cookie$ attack enables the attacker to invalidate cookie flags even if they are encrypted by TLS by exploiting insecure mechanisms within HTTP and the faulty implementation of AES-GCM [17]. AES-GCM is currently the most widely used cipher for symmetric authenticated encryption in TLS. However, if the same nonce is used again at least once in the encryption process, the attacker can extract the authentication key for the session and invalidate the cookie flag, which is a common scenario in practice on many HTTPS servers [16].

In order to demonstrate the practicality of our attack scenario, we first examine GCM nonce usage patterns in Quantcast's top 56,000 web sites, finding that 141 servers are vulnerable in terms of nonce management. We then perform a $rotten\ cookie$ attack against five major web browsers (Chrome, Internet Explorer, Edge, Firefox, and Safari) and show that all of these browsers accept the cookie without any flags, meaning that the attacker can readily obtain private cookies from the target illegally. In addition, we examine the web browsers and popular web applications to assess how they handle the cookie flags of private cookies. Specifically, we conduct an in-depth assessment of the five major web browsers and ten popular web applications (Google, YouTube, Amazon, Facebook, Yahoo, Walmart, LinkedIn, Twitter, Netflix, and Instagram). We find that most of the web browsers accept truncated or incorrectly formatted response messages, which can then be used to deactivate cookie flags. We also found that only Amazon set both `HttpOnly` and `Secure` flags on their private cookies at the same time.

We also considered another form of cookie theft attack based on duplicate connections, which exploits the absence of HTTP integrity verification. When the zero round trip time resumption (0-RTT) protocol is run, nonce duplication should be checked rigorously. Otherwise, different sessions may be established with the same session key through a replay attack, which can be further used to conduct a $rotten\ cookie$ attack. We also present a practical mitigation approach for the transport layer to prevent the reuse of the nonce with AES-GCM in TLS.

A. Our contributions

- We show that HTTPS responses in various browsers can be used as a trigger point to break the security of cookies by deactivating the cookie flags. We describe our attack scenario based on cases that may lead to security failure (e.g., the faulty implementation of TLS or applications) and demonstrate the practicality of the attack by examining Quantcast's top 56,000 websites with major web browsers.
- We conduct an in-depth assessment of the five major browsers and ten popular web applications. We explore whether the browsers correctly identify the format of HTTP response messages and how they handle the header options. We also investigate whether popular web applications set cookie flags securely.
- We propose a practical mitigation approach to improve the security of cookie mechanisms against our attack scenario in the transport layer.

B. Organization

We briefly introduce TLS, including the session key derivation process and the AES-GCM cipher suite, in Section II. In Section III, we explain the weaknesses in HTTPS software (e.g., web browsers) that can be exploited. In Section IV, we define the attack model and the proposed attack scenario based on the faulty implementation of AES-GCM, in which the attacker can remove cookie flags. We also propose an approach to mitigate our attack scenario and explain how major browsers should handle HTTP responses correctly. In Section V, we discuss the results of an investigation into major browsers and popular web applications. In Section VI, we conclude the paper.

II. Background & Related Work

In this section, we provide the information necessary to understand our attack scenario. To understand how the faulty implementation of AES-GCM can weaken the integrity of messages, known as a forbidden attack [18], we briefly explain how the AES-GCM cipher works in TLS 1.2.

A. Transport Layer Security

TLS [2] is the most widely used protocol for secure channels. It provides mutual authentication between the client and the server and guarantees the confidentiality and integrity of messages by supporting various cipher suites. TLS can be divided into two protocols: a handshake protocol and a record protocol[1].

[1]Recently, even though TLS 1.3 [19] has been standardized by the IETF to remove the vulnerabilities in TLS 1.2 [2] and support the 0-RTT handshake protocol, TLS 1.2 is still the most widely used version in practice. Thus, we focus on TLS 1.2 in this paper.

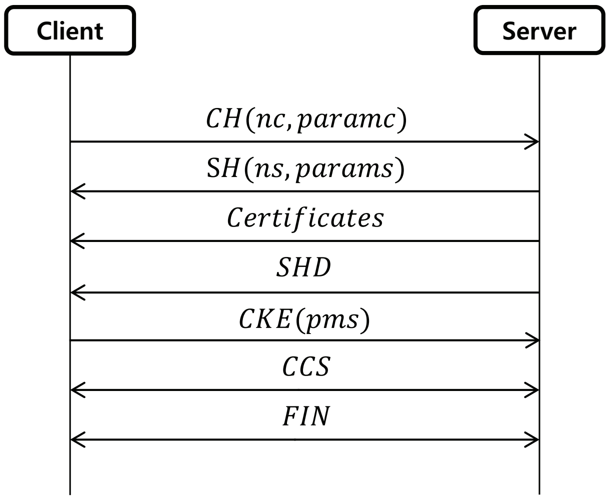


Fig. 1. TLS handshake protocol

1) TLS handshake protocol: To negotiate the cipher suite that is used in the record protocol and to provide a mechanism for mutual authentication between the client and the server, the handshake protocol should be run before sending or receiving encrypted data through the record protocol. For simplicity, we consider one-way authentication, where only server authentication is required, which is the de facto authentication procedure in practice. Fig. 1 depicts the general TLS handshake protocol with the RSA key exchange protocol[2].

First, a client sends a ClientHello (CH) message containing a random client nonce (nc) and handshake parameters ($paramc$) including the TLS version, a supported list of cipher suites, and TLS extensions. Upon receiving these messages from the client, the server sends back a ServerHello (SH) message, its certificates, and a ServerHelloDone (SHD) message. SH contains a fresh server nonce (ns) and handshake parameters ($params$) including the TLS version, the server's chosen cipher suite, and selected extensions. If the client verifies the server's certificates successfully, it randomly generates a pre-master-secret (pms), encrypts it with the public key of the server, and sends it to the server in a ClientKeyExchange (CKE) message. The server then decrypts it with its private key. After the pms is exchanged, they compute common session keys independently using the shared pms in a deterministic way. The client then signals a change of keys with a ChangeCipherSpec (CCS) message followed by a Finished (FIN) message. The client and the server check the integrity of the exchanged parameters and derive the same valid session keys after the FIN message. They can then communicate with each other by sending data encrypted under the derived session keys in the record protocol.

After the client and the server share the same valid pms, they generate the master-secret (ms) and session key block independently as follows: ms = PRF (pms , "master secret", nc , ns), session key block = PRF (ms, "key expansion", nc , ns), where PRF is a pseudo-random function. The session key block is then divided into six elements, client-write-mac-key, server-write-mac-key, client-write-key, server-write-key, client-write-IV, and server-write-IV, which are used in the record protocol to encrypt data or generate a valid message authentication code (MAC). Nonce values are used to prevent replay attacks by deriving different session keys for each session. Therefore, it is important to guarantee the uniqueness and freshness of the random nonce. If a server sends a duplicated nonce to a client, the attacker can open a duplicated connection using the same session keys as the target connection by replaying its handshake messages.

2) TLS record protocol: After the handshake protocol and session key derivation, the record protocol is followed, where data can be exchanged in an encrypted form using the derived session keys. In the record protocol, plaintext data blocks are translated into plaintext fragments, which in turn are translated into ciphertext fragments using encryption and MAC functions. In this procedure, TLS specifications support three cipher-modes (chosen in the handshake protocol): StreamCipher, BlockCipher, and AEADCipher.

We focus on the use of cipher AEAD (which stands for authenticated encryption with associated data), which can be exploited by forbidden attacks [18] (explained in Section 2.2). Unlike the other cipher modes, AEADCipher does not use MAC-keys but only IVs (client-write-IV and server-write-IV) derived from the session key block in the handshake protocol to determine the integrity of the fragments. AEADCipher mode computes the additional authenticated data, ciphertext, and plaintext respectively as follows:

- additional authenticated data = seq-num||plaintext.type|| plaintext.version||plaintext.length
- ciphertext := AEAD-Encrypt(write-key, nonce, plaintext, additional authenticated data)
- plaintext := AEAD-Decrypt(write-key, nonce, ciphertext, additional authenticated data)

AEAD-Encrypt and AEAD-Decrypt are the encryption and decryption functions in AEADCipher mode, respectively. IVs are included in the nonce. The nonce is then used to give the authenticity of the data fragments. Details of this procedure are described in the next section with the AES-GCM cipher suite.

B. AES-GCM and Forbidden Attacks

In this section, we briefly describe how AES-GCM works in TLS1.2 and introduce forbidden attacks [18]. In a forbidden attack, the attacker can compute the authentication key for a session if the same nonce is reused.

1) AES-GCM: AES-GCM is a mode of operation for block ciphers which provides AEAD. It is a standardized cipher suite in TLS1.2. To provide both data encryption and authentication, AES-GCM consists of two phases: encryption and authentication. Fig. 2 depicts the general process with two plaintext blocks and one additional authenticated data block. Like a

[2]Even though RSA-based key exchange in TLS cannot support forward security, unlike a Diffie-Hellman-based key exchange, forward security is not a concern in our cookie theft attack. Therefore, we introduce the RSA-based handshake TLS protocol for simplicity without a loss of generality in terms of the cookie theft attack.

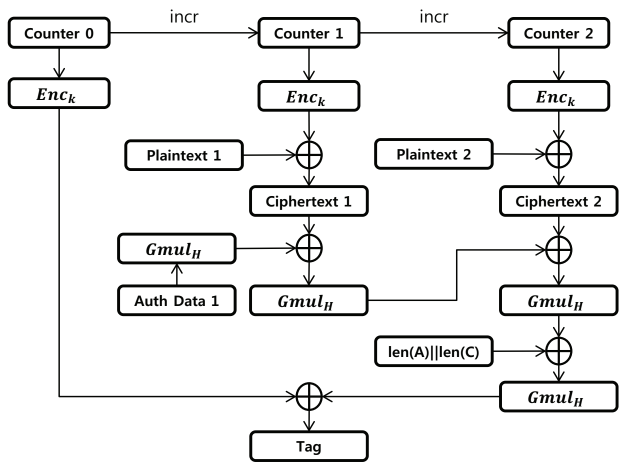


Fig. 2. Description of AES-GCM

typical counter mode, the blocks are numbered sequentially and encrypted with a block cipher AES. The result of this encryption is XORed with the plaintext block to produce a ciphertext block. It is essential that different IVs are used for each block.

In TLS with AES-GCM, the inputs for the encryption phase are a symmetric write key, plaintext, and IV. IV is used to form the counter block (J). IV is composed of two parts: implicit IV and explicit IV. Implicit IV, which is 4bytes in size, is derived during the TLS handshake protocol, which includes client-write-IV and server-write-IV. Explicit IV is an 8-byte nonce and transmitted explicitly in each fragment as a counter. In order to guarantee the uniqueness of the nonce, the counter value is increased by 1 in each new ciphertext block. The AES-GCM encryption process for the generation of one ciphertext fragment consisting of n plaintext blocks is as follows:

1) Generate a 96-bit long IV (implicit + explicit components).
2) Generate 128-bit long counter blocks J_i, where $J_i = IV \| \text{cnt}$, and $\text{cnt} = (i+1) \bmod 2^{32}$, for $i \in \{0, \dots n\}$.
3) Generate i-th ciphertext block $C_i = Enc_k(J_i) \oplus P_i$, where $Enc_k(\cdot)$ is AES encryption with symmetric key k and P_i is the i-th plaintext block.
4) Output ciphertext $C = C_1 \| C_2 \| C_3 \| \dots \| C_n$.

After the ciphertext blocks are created, the Galois Field (GF) [20] multiplication function combines the authenticated data to produce an authentication tag in GF(2^{128}) defined by the irreducible polynomial $f = 1 + \alpha + \alpha^2 + \alpha^7 + \alpha^{128}$. The AES-GCM authentication process for a ciphertext fragment with m blocks of additional authenticated data is described as follows:

1) Generate authentication hash key $H = Enc_k(0^{128})$.
2) Starting with $X_0 = 0$, compute GF multiplications over the additional authenticated data (A) consisting of m blocks (note that last block is padded with zeros to achieve a 128-bit length):
$X_i = Gmul_H(X_{i-1} \oplus A_i)$, for $i \in \{1, \dots m\}$,
where $Gmul_H(\cdot)$ is the GF multiplication function and A_i is the i-th authenticated data block.
3) Compute GF multiplications over ciphertext blocks C_i consisting of n blocks:
$X_{i+m} = Gmul_H(X_{i+m-1} \oplus C_i)$, for $i \in \{1, \dots n\}$.
4) Compute the final multiplication using the bit-lengths of A and C represented by 64bits:
$X_{m+n+1} = Gmul_H(X_{m+n} \oplus (len(A) \| len(C)))$,
where len(x) is the function that returns the length of x in bits.
5) Output authentication tag $T = X_{m+n+1} \oplus Enc_k(J_0)$.

The final output of the AES-GCM cipher is ciphertext C concatenated with authentication tag T: $C \| T$

2) Forbidden Attack: In 2006, Joux [18] described for the first time an attack against the GCM cipher, referred to as a forbidden attack. Even though a forbidden attack is damaging to AES-GCM in theory, it is not considered a practical threat because of the implicit assumption that the nonce in the AES-GCM cipher will never be reused. However, in a real-world environment, this is not always the case because of faulty implementation [16].

The uniqueness of the nonce (which is the explicit IV in the GCM) is essential for GCM security. In this section, we briefly explain how a forbidden attacker can calculate an authentication key if the nonce is reused in AES-GCM. Authentication tag T can be computed using the following polynomial function g with authentication key H:

$$\begin{aligned} g(X) = {} & A_1 X^{m+n+1} + \dots + A_m X^{n+2} \\ & + C_1 X^{n+1} + \dots + C_n X^2 + LX + Enc_k(J_0), \end{aligned} \tag{1}$$

where L is the encoded block of $len(A)$ and $len(C)$. All values are known to the attacker except $Enc_k(J_0)$. Tag T will then be computed as $T = g(H)$. If two messages are encrypted with the same nonce, $Enc_k(J_0)$ will be the same. For simplicity, we only consider one additional authenticated data block and one ciphertext block. $g_1(X)$ and $g_2(X)$ are the authentication polynomials for each message.

$$\begin{aligned} g_1(X) &= A_1 X^3 + C_1 X^2 + L_1 X + Enc_k(J_0), \\ g_2(X) &= A_2 X^3 + C_2 X^2 + L_2 X + Enc_k(J_0). \end{aligned}$$

By adding the corresponding known tag T ($g_1(H) = T_1, g_2(H) = T_2$), we have

$$\begin{aligned} {g_1}'(X) &= A_1 X^3 + C_1 X^2 + L_1 X + Enc_k(J_0) + T_1, \\ {g_2}'(X) &= A_2 X^3 + C_2 X^2 + L_2 X + Enc_k(J_0) + T_2. \end{aligned}$$

Since ${g_1}'(H) = {g_2}'(H) = 0$, by adding these two polynomials, we can obtain

$$\begin{aligned} {g_{1+2}}'(X) = {} & (A_1 + A_2) X^3 + (C_1 + C_2) X^2 \\ & + (L_1 + L_2) X + T_1 + T_2, \end{aligned} \tag{2}$$

which is known to the attacker and satisfies ${g_{1+2}}'(H) = 0$. Because ${g_{1+2}}'(H) = 0$, the attacker can factor the polynomial to recover a short list of candidates for authentication key H, which is relatively small in practice. If the attacker successfully recovers authentication key H, he can carry out a forgery on any valid ciphertext by computing valid authentication tag T. The full details of the attack are described in [16] and [18].

3) Forbidden Attack on TLS: As discussed above, the uniqueness of the nonce is essential to GCM security. Therefore, in the TLS record layer, IV should never be reused. Unfortunately, however, practical systems sometimes fail to achieve this due to faulty implementation or management. For example, Böck et al. [16] found vulnerable HTTPS servers in the real world and exploited practical forbidden attacks to forge valid ciphertext in TLS 1.2.

III. Security of Web Browsers

In this section, we explain a security weakness of web browsers that can be potentially exploited by our cookie theft attack that invalidates cookie flags. An example of how `HttpOnly` and `Secure` flags are appended to the `Set-Cookie` option in the header of an HTTP response message is as follows:

```
HTTP/1.1 200 OK
Set-Cookie: Name=value;Secure;HttpOnly
Content-Length: 0
```

A. *Integrity of* `Set-Cookie` *field attributes*

Before describing our attack scenario, we describe how browsers handle HTTP response message headers. To support flexibility and scalability, most HTTP-based software does not check all of the header options and the integrity of the HTTP response message. We give an example of the vulnerability in how a browser reads HTTP messages over TLS (HTTPS):

```
HTTP/1.1 302 Redirect
Location:https://A.com/B
Set-Cookie: Name=value;S?cure
Content-Length: 0
```

Assume that there is one typo in the HTTP response message “S?cure” (which should be “Secure”), by wrong-configuration of the server and sent over TLS with encryption. Because “S?cure” is uninterpretable, the browser should reject all response messages to protect against any potential threats arising from the incorrect configuration of the server or message modification by an attacker. However, we found that almost every major web browser, including Chrome, Internet Explorer, Edge, Firefox, and Safari, ignores uninterpretable sections in the response message (e.g., “S?cure”) and accepts the rest without any rejection. Therefore, cookies can be set without a `Secure` flag, which can be exploited by cookie-theft adversaries.

B. *Format of HTTP response messages*

We also found that most major browsers accept incorrect HTTP response message formats without any rejection. An HTTP response message is composed of the following four components: a status line, multiple headers, an empty line, and the message body. The status line indicates the status of the entire response message. It presents the HTTP version, a numeric status code and an associated textual phrase such as OK (200), Bad Request (400), Not Found (404) or Forbidden (403). Multiple headers allow the server to give additional information about the response message. The empty line indicates the end of the header message (Double CRLF: \r\n\r\n). In order to prevent cookie-cutter attacks [5], browsers should at least check the end of the HTTP response’s header message (re: the empty line). Most major browsers have been updated recently to check if the end of header message is truncated or not; however, we observed that all of the major browsers accept the following HTTP message even if it contains multiple status-lines:

```
HTTP/1.1 200 OK
HTTP/1.1 302 Redirect
Location:https://A.com/B
Set-Cookie: Name=value;Secure
Content-Length: 0

[ message-body ]
```

This incorrect response message format should be rejected by browsers because it might have been generated by the incorrect configuration of the server or an illegally modified message from an attacker. However, we found that browsers accept this incorrect format and set the cookie regardless.

To summarize, we found that most current web browsers do not correctly check the integrity of header messages and the format of the HTTP response message. This indicates that the HTTP layer itself lacks security in terms of protecting messages. Thus, HTTPS protocol security fully depends on TLS. Specifically, cookie flags should be protected by the MAC mechanism in TLS, even if the cookie is managed in the application layer (such as HTTP) and is independent of the transport layer. This dependency of security on different layers is a potential threat to the security of cookies if we do not consider the above weaknesses of the browsers and the vulnerabilities of TLS. In Section IV, we describe details of an attack scenario that enables an attacker to steal cookies by utilizing the properties described above and the existing vulnerabilities (including faulty implementation) of TLS or the application. In Section V, we discuss the practical implications of the above problem in the real world and explore how most major browsers handle it.

IV. The Proposed Attacks

In this section, we describe how an attacker can ensure that cookies are set without flags even if they are encrypted under TLS by utilizing the weaknesses of the browser. After defining the attack model, we describe our *rotten cookie* attack scenario based on AES-GCM forbidden attacks and evaluate the practicality of our attack in the real world. Finally, we suggest a practical strategy to protect against our attack.

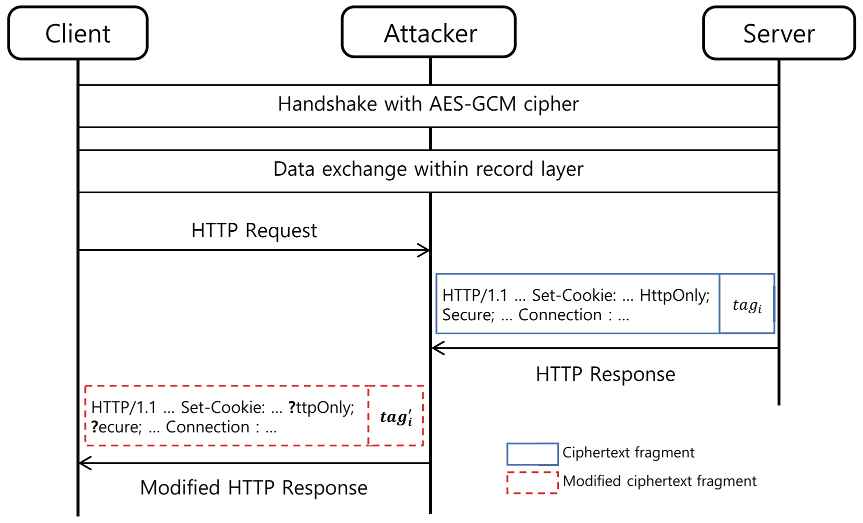

Client
Attacker
Server
Handshake with AES-GCM cipher
Data exchange within record layer
HTTP Request
HTTP/1.1 ... Set-Cookie: ... HttpOnly; Secure; ... Connection : ...
tag_i
HTTP Response
HTTP/1.1 ... Set-Cookie: ... ?ttpOnly; ?ecure; ... Connection : ...
tag_i'
Modified HTTP Response
Ciphertext fragment
Modified ciphertext fragment

Fig. 3. *Rotten cookie* scenario based on a forbidden attack

A. Attack Model

In our attack scenario, we suppose that an MITM attacker[3] has full control over the network traffic between the client and the server. In addition, the attacker can readily predict the location of the internal content of his interest in TLS fragments such as the header of the HTTP response message. For instance, the attacker can take the normal TLS content from the server before he attempts to carry out the attack and determine where the cookie flag starts or what cookie values it contains.

In our attack model, the attacker desires to ensure cookies are set without `Secure` or `HttpOnly` flags even if all of the messages are encrypted. If cookies are set without `Secure` or `HttpOnly` flags, the attacker is able to conduct an MITM attack over HTTP or an XSS attack and then obtain the target private cookies of the client.

B. Rotten Cookie Attack

The *rotten cookie* attack is based on a forbidden attack where the client and the server use the AES-GCM cipher suite in TLS. We assume that the server may re-use the same GCM nonce either due to faulty implementation or at random. The attacker can thus utilize this to conduct a forbidden attack to recover the authentication key. After locating a vulnerable server and a target client, the attacker performs an MITM attack, manipulating TLS fragments and attempting to steal private cookies from the victim by removing cookie flags. $\mathcal{C}$, $\mathcal{A}$ and $\mathcal{S}$ are the target client, the MITM attacker, and the vulnerable server, respectively. Fig. 3 depicts our attack scenario based on a forbidden attack, which proceeds as follows:

[3]These are reasonable and practical assumptions because most of the current network traffic goes through various intermediate devices such as access points, HW or SW routers, and so on, which might be easily compromised or deployed by adversaries.

1) If $\mathcal{C}$ initiates a TLS connection with $\mathcal{S}$, $\mathcal{A}$ observes the handshake protocol to determine whether the AES-GCM cipher is negotiated.
2) If the connection is set up with the AES-GCM cipher suite, $\mathcal{A}$ selects two or more TLS fragments where the same nonce is used.
3) $\mathcal{A}$ then builds a polynomial from the two fragments to calculate possible candidates for the authentication key.
4) By comparing this with other valid TLS fragment sets, $\mathcal{A}$ can finally compute a valid authentication key and conduct a cookie theft attack.
5) Because $\mathcal{A}$ can determine the location of the first byte of both `HttpOnly` and `Secure`, $\mathcal{A}$ replaces the first byte with any random value and generates a valid tag for the modified fragment including the original additional authenticated data with the previously computed authentication key.
6) $\mathcal{A}$ sends the modified fragment to $\mathcal{C}$. $\mathcal{C}$ then discards the uninterpretable flag in the response message and sets the cookie without the flags.
7) Finally, $\mathcal{A}$ is able to conduct an MITM attack over HTTP or an XSS attack and then obtain the target private cookie of $\mathcal{C}$.

1) Evaluation: As described above, an attacker is able to steal private cookies if the nonce is reused, even if they are encrypted securely. The forbidden attack may be considered a somewhat strong attack because the attacker has the ability to not only generate a valid tag on a ciphertext but also generate a valid ciphertext, which allows him to inject his own content to the clients by utilizing XOR malleability. However, we note that it is not a powerful adversary since the ability to inject content is not our attack model in terms of cookie theft. In other words, even if the attacker can inject malicious messages he manipulated, it would not help to steal private cookies because he does not possess a decryption key for the session anyway. Furthermore, in our attack scenario, the attacker

TABLE I
NUMBER OF SERVERS USING ABNORMAL NONCE AMONG QUANTCAST TOP 56000 SITES.

Status	Type	# of the server	Remarks
Secure	R-4CTR	236	0xFD B0 03 19 02 00 00 00 0xFD B0 03 19 03 00 00 00 0xFD B0 03 19 04 00 00 00 0xFD B0 03 19 05 00 00 00
	8CTR	59	0x01 00 00 00 00 00 00 00 0x02 00 00 00 00 00 00 00 0x03 00 00 00 00 00 00 00 0x04 00 00 00 00 00 00 00
Vulnerable	Random	98	0x6F EB AC 1D 7B 31 5F C7 0x01 D0 DA 29 1E F0 09 2A 0x74 62 E5 D3 4F 68 F2 81 0xE9 3D EA 19 C1 1D 82 E2
	Fixed	43	0x26 61 AF 70 7C 2F 5C 13 0x00 00 00 00 00 00 00 00 0x00 00 00 00 00 00 00 00 0x00 00 00 00 00 00 00 00

does not need to care about the plaintext content generated by slightly modified ciphertext such as "?ecure" in Fig. 3. Because the attacker's goal is to obtain an authentication key and generate only an associated valid tag (not a valid ciphertext) [4], our attack is more practical and easier to perform. Therefore, to protect the private cookie, the server should not repeat any nonce during a TLS connection. One of the promising ways to ensure this is using a counter. In counter mode, the nonce starts with 0 or a random value and increases by 1 with each message.

However, in the real world, we found that a non-negligible number of servers still generate duplicate nonces. Böck et al. [16] surveyed how many servers produce repeated nonces on World Wide Web HTTPS servers in 2016. In this study, we implemented a duplicated nonce detection tool based on GCM Nonce Checker supported by Böck et al. [21] and re-examined Quantcast's top 56,000 sites [22] to validate the practicality of our attack and the impact on the real-world web service environment. According to our investigation, we observed that 436 out of 56,000 websites (the other servers use normal counter mode or do not support an AES-GCM cipher) use one of the following four types of non-typical nonce patterns: the 4 left most bytes are random and the most significant byte of the 4 right-most bytes increases by 1 (Random||4-byte counter [R-4CTR] type), the most significant byte of the 8 bytes increases by 1 (8-byte counter [8CTR] type), the 8 bytes are random (Random type), and the 8 bytes are fixed (Fixed type). Among the four different types, R-4CTR and 8CTR types are secure against our attack because both types can be seen as variations of the counter mode. However, Random and Fixed types are vulnerable because Random type may allow duplicated nonces[5] and Fixed type always generates duplicated nonces. Thus, servers that employ either Random or Fixed type nonce patterns could be attacked under our *rotten cookie* attack due to the redundant nonce values of the AES-GCM cipher in TLS.

[4]Specifically, the cost of our attack is the time to factor the polynomial in Eq. (2).

[5]In fact, finding a duplicated nonce in the case of Random type seems not so much practical (e.g., due to the birthday paradox, nonce collision probabilities [16] are about 0.2% and 80% after 2^{28} and 2^{33} transmissions under the same encryption key, respectively). However, we classify them as vulnerable because there is still a non-negligible possibility of a nonce repeating with incorrect cryptographic implementation, such as a weak pseudorandom generator [23], [24], in practice. Therefore, it is reasonable to assume that preserving the ideal freshness and uniqueness of each random nonce sometimes fails in a practical environment.

TABLE II
VERSION OF WEB BROWSERS

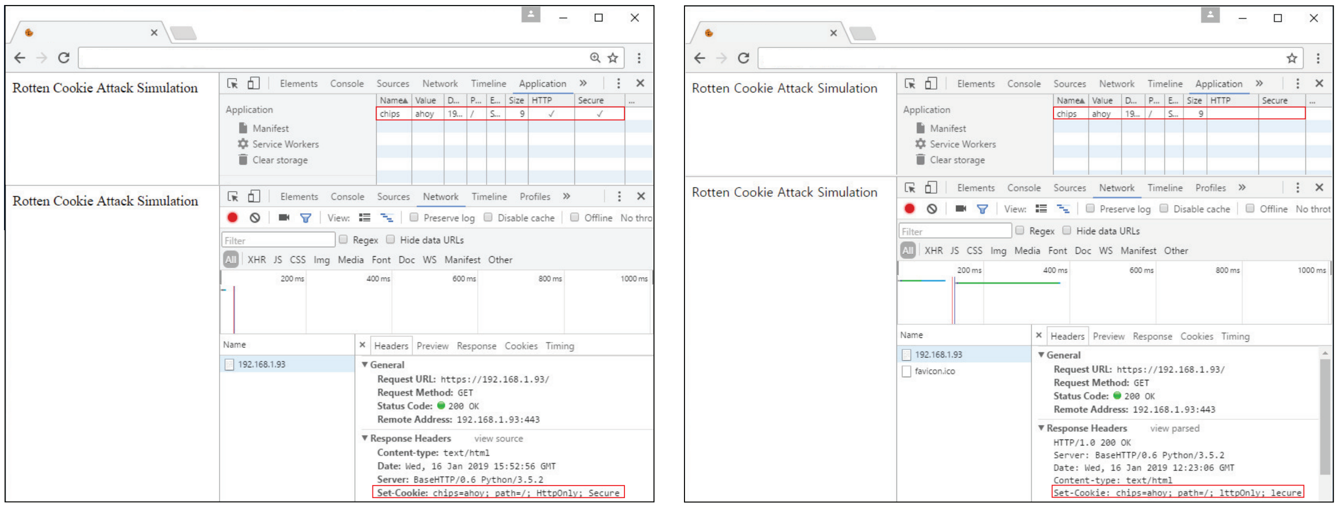
Web browser	Version
Chrome	55.0.28883.87
Internet Explorer	11.187.14393.0
Edge	38.14393.0.0
Firefox	50.0.1
Safari	10.0.3

As shown in Table I, we found that 141 servers are vulnerable (98 Random type servers and 43 Fixed type servers), of which numbers are non-negligible in practice. Even worse, the 43 Fixed type servers use the same nonce pattern which initializes with an 8-byte random number, and all subsequent nonces are set to 0.

Therefore, the proposed attack is a practical concern because numerous vulnerable HTTPS servers exist in the real world.

2) A practical attack on existing web browsers: To prove the possibility of the proposed attack in real-world HTTPS sessions, we implement an MITM attacker in Python with Scapy [25], which is an interactive packet manipulation tool, and OpenSSL 1.0.2.q [26]. In addition, to recover the authentication key from the record data, we leverage the factoring functionality given by [27].

We choose five target clients $\mathcal{C}$s, Chrome, Internet Explorer, Edge, Firefox, and Safari, which are the most popular web browsers in the real world. Table II shows the version information of the five web browsers we used in the attack. We make them communicate with the vulnerable server $\mathcal{S}$ of Random or Fixed type that generates a duplicate AES-GCM nonce. In our experiment, we adopted the latter to make it simpler to present the experimental results. When $\mathcal{S}$ transmits a cookie containing `Secure` and `HttpOnly` flags to the five web browsers through a TLS connection, we employ MITM attacker $\mathcal{A}$ to accomplish *rotten cookie* attack. The original cookie message sent by the server is as follows:



Rotten Cookie Attack Simulation

Name	Value	D...	P...	E...	Size	HTTP	Secure
chips	ahoy	19...	/	S...	9	✓	✓

```
General
  Request URL: https://192.168.1.93/
  Request Method: GET
  Status Code: 200 OK
  Remote Address: 192.168.1.93:443
Response Headers
  Content-type: text/html
  Date: Wed, 16 Jan 2019 15:52:56 GMT
  Server: BaseHTTP/0.6 Python/3.5.2
  Set-Cookie: chips=ahoy; path=/; HttpOnly; Secure
```

(a) Original cookie status

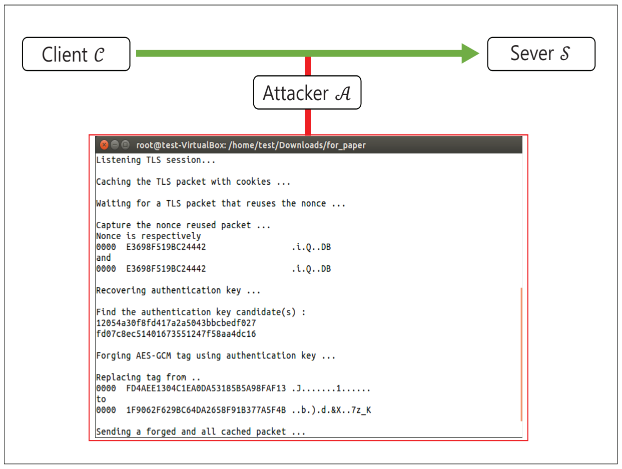
Rotten Cookie Attack Simulation

Name	Value	D...	P...	S...	Size	HTTP	Secure
chips	ahoy	19...	/	S...	9		

```
General
  Request URL: https://192.168.1.93/
  Request Method: GET
  Status Code: 200 OK
  Remote Address: 192.168.1.93:443
Response Headers
  HTTP/1.0 200 OK
  Server: BaseHTTP/0.6 Python/3.5.2
  Date: Wed, 16 Jan 2019 12:23:06 GMT
  Content-type: text/html
  Set-Cookie: chips=ahoy; path=/; lttpOnly; lecure
```

(b) Modified cookie status under a *rotten cookie* attack

Fig. 4. Cookie status before and after a *rotten cookie* attack against Chrome.



```
Listening TLS session...

Caching the TLS packet with cookies ...

Waiting for a TLS packet that reuses the nonce ...

Capture the nonce reused packet ...
Nonce is respectively
0000  E3698F519BC24442                 .i.Q..DB
and
0000  E3698F519BC24442                 .i.Q..DB

Recovering authentication key ...

Find the authentication key candidate(s) :
12054a30f8fd417a2a5043bbcbedf027
fd07c8ec51401673551247f58aa4dc16

Forging AES-GCM tag using authentication key ...

Replacing tag from ..
0000  FD4AEE1304C1EA0DA53185B5A98FAF13 .J.......1......
to
0000  1F9062F629BC64DA2658F91B377A5F4B ..b.).d.&X..7z_K

Sending a forged and all cached packet ...
```

Fig. 5. Manipulation of the packet and recovery of the authentication key

```
  HTTP/1.1 200 OK
  Server: BaseHTTP/0.6 Python/3.5.2
  Path: /
  Set-Cookie: chips=ahoy; path=/;
HttpOnly; Secure
```

Before the attack starts, if $\mathcal{C}$ receives an original cookie message from $\mathcal{S}$, the cookie would be set with `Secure` and `HttpOnly` flags as shown in Fig. 4(a). We can also observe how the cookie would be parsed at the bottom of Fig. 4(a). Our *rotten cookie* attack then progresses as follows[6].

1) When the AES-GCM cipher is negotiated under a TLS connection, as shown in Fig. 5, $\mathcal{A}$ holds all packets sent from $\mathcal{S}$ to $\mathcal{C}$ until at least two record data including the cookie value are received.
2) $\mathcal{A}$ then extracts additional authenticated data, the ciphertext, and the tag from the record fragments, which is encrypted under the same session key (that is, the explicit nonce of the AES-GCM cipher is redundant), and builds a polynomial from the derived data.
3) This polynomial is then factorized to restore candidates for the authentication key. If only one candidate for the authentication key is obtained, $\mathcal{A}$ replaces the first byte of the cookie flags with an arbitrary character using an XOR operation and utilizes the recovered authentication key to generate a validate tag for the altered content. In this case, we attempted to change the first letters 'H' and 'S' both to 'l' for `HttpOnly` and `Secure` by performing the following operations:

$$data[i] = data[i] \oplus \text{'H'} \oplus \text{'l'},$$
$$data[j] = data[j] \oplus \text{'S'} \oplus \text{'l'}, \qquad (3)$$

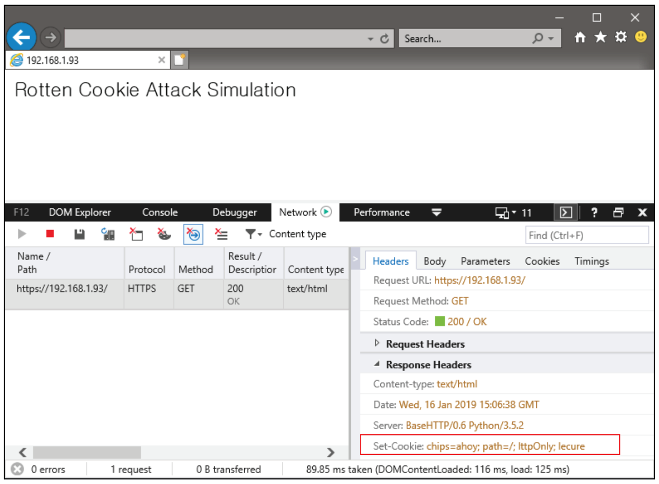
where $data$ is the encrypted HTTP content including the cookie value, $\oplus$ is the XOR operation, and i and j indicate the location of the first character of `HttpOnly` and `Secure`, respectively.
4) After receiving the manipulated cookie, $\mathcal{C}$ accepts the cookie values except for the uninterpretable sections, as shown in Fig. 4(b). We can also see that the HTTPS fragment modified by the XOR operation was parsed normally with "lttpOnly" and "lecure", and Chrome ignored the cookie flag values.

In this way, we conduct a *rotten cookie* attack on Internet Explorer, Edge, Firefox, and Safari, and the resulting screenshots are presented in Figs. 6(a), 6(b), 6(c) and 6(d), respectively. As we expected, all of the five major web browsers accept the cookie without a `Secure` flag and `HttpOnly` flag, which demonstrates the practicality of our attack in the real-world web environment.

C. Further Consideration

A *rotten cookie* attack exploits a vulnerability in TLS that incapacitates the cookie flags. As an extension of the attack, we further considered exploiting duplicated connections in

[6]Since the attack is identical and independent of the web browser, we describe how our attack works with Chrome without a loss of generality.



192.168.1.93

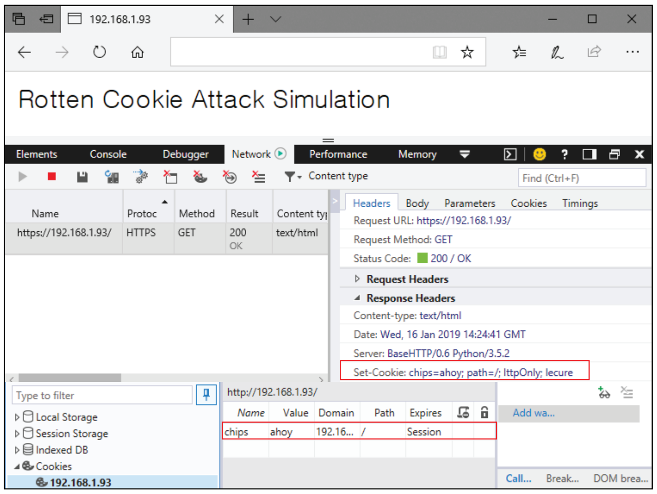
Rotten Cookie Attack Simulation

Name / Path	Protocol	Method	Result / Descriptior	Content type
https://192.168.1.93/	HTTPS	GET	200 OK	text/html

Request URL: https://192.168.1.93/
Request Method: GET
Status Code: 🟩 200 / OK
Request Headers
Response Headers
Content-type: text/html
Date: Wed, 16 Jan 2019 15:06:38 GMT
Server: BaseHTTP/0.6 Python/3.5.2
Set-Cookie: chips=ahoy; path=/; IttpOnly; lecure

0 errors | 1 request | 0 B transferred | 89.85 ms taken (DOMContentLoaded: 116 ms, load: 125 ms)

(a) Cookie status in Internet Explorer



192.168.1.93

Rotten Cookie Attack Simulation

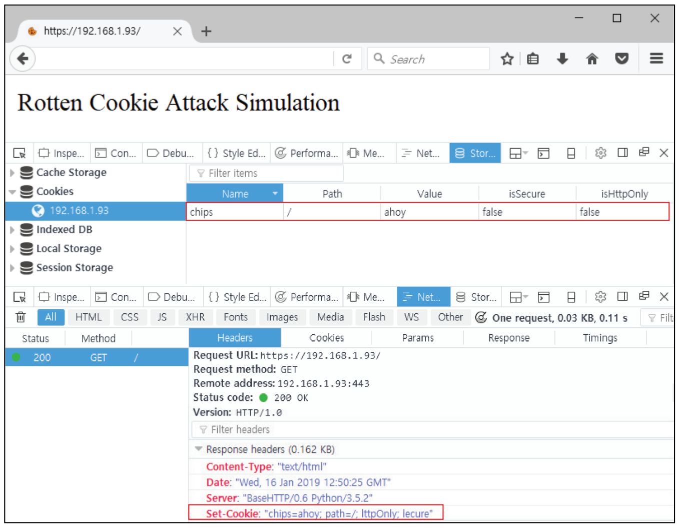
Name	Protoc	Method	Result	Content typ
https://192.168.1.93/	HTTPS	GET	200 OK	text/html

Request URL: https://192.168.1.93/
Request Method: GET
Status Code: 🟩 200 / OK
Request Headers
Response Headers
Content-type: text/html
Date: Wed, 16 Jan 2019 14:24:41 GMT
Server: BaseHTTP/0.6 Python/3.5.2
Set-Cookie: chips=ahoy; path=/; IttpOnly; lecure

http://192.168.1.93/

Name	Value	Domain	Path	Expires
chips	ahoy	192.16...	/	Session

(b) Cookie status in Edge



https://192.168.1.93/

Rotten Cookie Attack Simulation

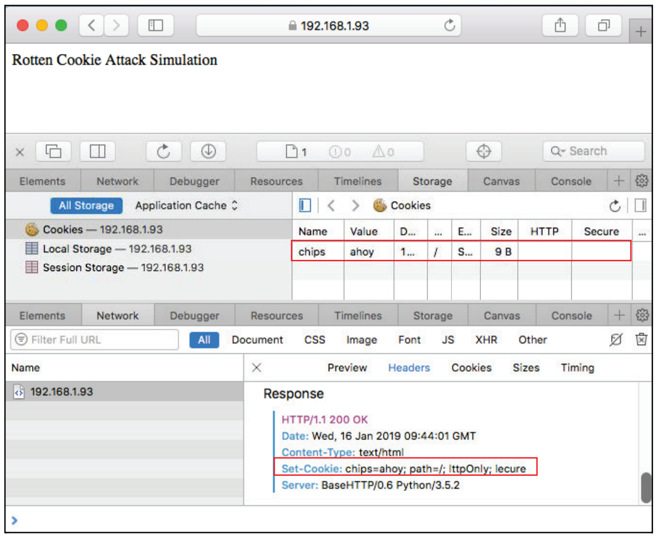
Name	Path	Value	isSecure	isHttpOnly
chips	/	ahoy	false	false

One request, 0.03 KB, 0.11 s

Status	Method	
200	GET	/

Request URL: https://192.168.1.93/
Request method: GET
Remote address: 192.168.1.93:443
Status code: ● 200 OK
Version: HTTP/1.0
Response headers (0.162 KB)
Content-Type: "text/html"
Date: "Wed, 16 Jan 2019 12:50:25 GMT"
Server: "BaseHTTP/0.6 Python/3.5.2"
Set-Cookie: "chips=ahoy; path=/; IttpOnly; lecure"

(c) Cookie status in Firefox



192.168.1.93

Rotten Cookie Attack Simulation

Name	Value	D...	...	E...	Size	HTTP	Secure	...
chips	ahoy	1...	/	S...	9 B			

Response
HTTP/1.1 200 OK
Date: Wed, 16 Jan 2019 09:44:01 GMT
Content-Type: text/html
Set-Cookie: chips=ahoy; path=/; IttpOnly; lecure
Server: BaseHTTP/0.6 Python/3.5.2

(d) Cookie status in Safari

Fig. 6. The result of a *rotten cookie* attack against IE, Edge, Firefox, and Safari.

which two different sessions share the same session key, of which case might be caused by a vulnerability in the security protocol.

To prevent cookie-cutter attacks, most browsers have been recently updated to check whether the HTTP header is truncated or not by forcing acceptance of the string “\r\n\r\n”, which indicates the end of the header message (CVE-2013-2853). However, this may not completely protect private cookies. Assume that the header of the response message is truncated right after the ‘e’ of the `Secure` flag and the next response messages are concatenated and sent together from the server. The browser may then receive the following message:

```
HTTP/1.1 302 Redirect
Location:https://A.com/B
Set-Cookie: Name=value;SeHTTP/1.1 200 OK
Date: Sun, 10 Oct 2010 23:26:07 GMT
Content-Length: 0
```

In this case, the browser recognizes two different response messages as a single message because it receives only one end string “\r\n\r\n”. Therefore, the cookie is set without a `Secure` flag. This seems unrealistic because TLS protects messages. However, we show that it is possible theoretically if we can establish a duplicated connection in the 0-RTT protocol. Once a duplicated connection is established between the client and the server, the attacker is not required to consider what type of cipher is used in the record protocol. We assume that the server sometimes fails to check the freshness of the server nonce in the 0-RTT protocol and may accept a duplication because of faulty implementation (e.g., a strike register for QUIC [28], [29]). In addition, the attacker can control the length of the content of the message, so he may choose a specific truncation point for TLS fragments.

The attack scenario is then described as follows:

1) If $\mathcal{C}$ initiates a 0-RTT handshake at a point in time, $\mathcal{A}$ records all of the handshake messages and initiates a 0-RTT handshake with $\mathcal{S}$ independently.
2) If $\mathcal{S}$ accepts any duplicated server nonce, $\mathcal{A}$ is able to

establish another duplicated connection which shares the same session keys. Even if both $\mathcal{C}$ and $\mathcal{A}$ send the same request message to $\mathcal{S}$, the length of the entire response message from $\mathcal{S}$ could differ because the server may provide different web services at different times.

3) In the original connection between $\mathcal{C}$ and $\mathcal{S}$, $\mathcal{A}$ manipulates the length of the response messages from $\mathcal{S}$ and creates a truncation point just before `HttpOnly` or `Secure` in the fragment containing the target cookie. $\mathcal{A}$ then replaces the next fragment with another fragment from his duplicated connection which has the same sequence number and a different message.
4) $\mathcal{C}$ accepts an HTTP response message which includes a double status line without rejecting it, and sets the cookie without flags.
5) Finally, $\mathcal{A}$ is able to conduct an MITM attack over HTTP or an XSS attack and then obtain the target private cookie from $\mathcal{C}$.

In TLS, a Snap-Start extension was employed to support the 0-RTT protocol, but unfortunately, it was removed from the draft in 2010 and is no longer supported by TLS. Thus, it is not practically implemented and deployed in the real world. Instead, Googles QUIC protocol [30], which inherits the concept of the 0-RTT protocol in the TLS Snap-Start extension, is deployed in practice. Therefore, we evaluate the QUIC protocol in terms of a cookie theft attack on a duplicated connection.

In the QUIC protocol, the server verifies if the nonce contained in the 0-RTT request is fresh by adopting a distinct mechanism that stores all nonces in the special purpose memory, called the strike-register, and rejects the duplicate. We note that the strike-register is used in the application layer and is not included in the QUIC protocol itself. However, the strike-register plays an important role for QUIC protocol security because it is used to prevent replay attacks (or nonce reuse attacks) [31], [32]. According to recent Google reports [28], [29], unfortunately, the strike register has a security flaw in that it cannot fundamentally prevent a replay attack in the QUIC protocol. Therefore, Google removed it from the latest version of the QUIC [7]. In order to check whether the QUIC server accepts duplicated nonces or not, we captured several 0-RTT request packets exchanged between a client (Google Chrome browser) and Google servers. We then replayed the packets that we captured to the servers. After 50 trials of an experiment, we observed that all of the 24 unique IP addresses for Google servers received the duplicated 0-RTT request message and replied with normal encrypted messages without rejecting repeated nonces. To proceed to the next step, similar to a cookie cutter attack [5], $\mathcal{A}$ needs to drop the truncated fragment that contains the cookie flags. In the QUIC protocol, the server sends a packet containing multiple frames and each frame can contain the flow control data or actual application data (i.e. HTTP message). If the packet containing both stream frames and flow control frames is dropped, the QUIC session will no longer be maintained between $\mathcal{C}$ and $\mathcal{S}$.

[7]We found that there have not yet been any alternative mechanisms proposed to replace the strike-register by Google.

TABLE III
SECURITY ANALYSIS OF CURRENT WEB BROWSERS

	Chrome	Firefox	Safari	IE	Edge
Malformed `Set-Cookie` attribute	Yes	Yes	Yes	Yes	Yes
Check empty line	Yes	Yes	Yes	No	Yes
Accept multiple status lines	Yes	Yes	Yes	Yes	Yes

D. Mitigation

In a real web environment, it may be impossible to define a well-formed HTTP message structure and force browsers to reject an ill-formed structure because the HTTP protocol is fundamentally designed to have a free-formed message structure. Furthermore, it may also be difficult for browsers to support all of the diverse header options and check them correctly. Thus, HTTP protocol shifts the responsibility for security, such as the guarantee of confidentiality and integrity, to the transport layer.

One root cause that enables our *rotten cookie* attack to break down transport layer security is the reuse of nonces in AES-GCM, which is a well-known vulnerability in TLS [33], [34]. If there are no servers which generate either a random or fixed nonce, our attack cannot be delivered in the real world.

Therefore, we recommend that the initial vector used for tag generation should be implemented with a counter mode, which eliminates nonce duplication. Generating a nonce in a more secure way is one of the independently ongoing research topic [35], [36], which can be employed as an effective mitigation in the transport layer.

V. INVESTIGATION OF COOKIE FLAGS IN WEB BROWSERS AND SERVERS

In this section, we conduct an in-depth assessment of how major browsers check the integrity and format of HTTP messages. We examine the five main web browsers (Chrome, Internet Explorer, Edge, Firefox, and Safari), with the versions described in Table II. We also investigate how popular websites set cookie flags securely. We select ten popular domains from among Alexa's top 15 websites as shown in Table IV.

Table III shows our investigation results. 'Yes' indicates the browser checks the corresponding content correctly; 'No' indicates it does not. First, we investigate whether they accept the `Set-Cookie` field when it has uninterpretable cookie flags because it can be utilized as the fundamental condition of our attack. Second, we investigate whether the browsers are secure against cookie-cutter attacks by forcing them to accept the string "\r\n\r\n" message over HTTPS. Finally, we investigate whether they accept incorrectly formatted HTTP messages created by concatenating two HTTP response messages.

A. Integrity of `Set-Cookie` Field Attributes

We found that all of the browsers do not verify the integrity of the `Set-Cookie` header message and accept cookies even if they included sections that are uninterpretable. This is useful because it enables HTTP to support various forms of functionality in a more flexible way, even when uninterpretable content exists in HTTP format.

TABLE IV
THE USE OF COOKIE FLAGS ON POPULAR WEBSITES

	Private cookie			Ratio	Public cookie	Total
	`HttpOnly`	`Secure`	Both			
google.com	2	1	1	0.25	5	9
youtube.com	4	1	1	0.17	2	8
amazon.com	0	0	3	1	9	12
facebook.com	0	3	6	0.7	0	9
yahoo.com	1	3	1	0.2	7	12
walmart.com	1	3	4	0.5	17	25
linkedin.com	0	4	2	0.3	9	15
twitter.com	0	2	5	0.7	9	16
netflix.com	1	0	1	0.5	5	7
instagram.com	0	5	5	0.5	1	11
Total	60			0.48	64	124

Private cookie : Cookie set with `Secure` or `HttpOnly` flag

Ratio : # of cookies set with both flags / # of private cookies

Public cookie: Cookie without `Secure` or `HttpOnly` flag

B. Format of HTTP Response Messages

As discussed in Section III, it is important for the browsers to check the entire format of an HTTP response message. We investigate whether the browsers accept HTTP response messages that include multiple status lines and how they handle missing empty lines over HTTPS. We found that all browsers accept multiple status lines, and strictly check for the empty line except Internet Explorer. Therefore, we recommend that all browsers, especially Internet Explorer, should check the entire format of the HTTP response message including both the status line and the end string of the header message (i.e., the empty line) correctly.

C. Cookie Flags on Popular Websites

We consider a private cookie to be a cookie that is set with a `Secure` or `HttpOnly` flag. We note that these cookie flags are clearly different security mechanisms against different attacks. For example, an `HttpOnly` flag is used to prevent XSS attacks, whereas a `Secure` flag is typically used to prevent HTTP MITM attacks.

In terms of `HttpOnly` flag, according to previous studies [37]–[41], it is not widely deployed in practice for two reasons. First, server administrators tend to believe that their servers might be secure against XSS attack even if the cookies are set without an `HttpOnly` flag so long as they strictly screen the malicious script code on their websites. Second, in order to support flexible functionality for non-HTTP APIs (such as JavaScript), the server unavoidably allows the browser to access the cookies in a legitimate manner. However, as shown in Table IV, we found that 38 out of 60 private cookies (about 63%) contained an `HttpOnly` flag, which indicates that `HttpOnly` flags are still used in practice in more than a half of the private cookies of Alexa's top 10 websites.

In terms of the `Secure` flag, in order for the browser to return the private cookies to the server safely, it should be included in the cookie. As shown in Table IV, our investigation indicates that 9 out of 60 private cookies (about 15%) did not contain a `Secure` flag (the corresponding websites are google.com, youtube.com, yahoo.com, walmart.com, and netflix.com). However, they all support HSTS, which enforces HTTPS connections. This would appear to make them secure, but we found that these five websites do not support HSTS with the $includeSubDomain$ option, whose purpose is preventing HSTS incapacitation attacks [42]. Even if HSTS without the $includeSubDomain$ protects host-only cookies which are not shared among sub-domains from such attacks, all cookies we captured for examining HSTS are marked with the Domain attribute which allows sub-domains to access them. Thus, those websites should provide either private cookies with a `Secure` flag or HSTS with the $includeSubDomain$ option to improve the security of the cookies.

Interestingly, only Amazon set both types of cookie flag in their cookies, and this may be the strictest management of cookie flags among Alexa's top 10 websites.

VI. CONCLUSION

When a server sets a private cookie with a flag, the `Set-Cookie` header option in the cookie flag message should be protected from any modification attack. It may be protected by TLS, but we have identified some vulnerabilities in TLS or the application layer that can invalidate cookie flag mechanisms even if it is securely encrypted. On the basis of our in-depth investigation, most major browsers do not strictly check the format of the response messages and ignore important header options. Thus, we found that most are vulnerable to our cookie theft attack when they are connected to a vulnerable server.

To protect against a $rotten\ cookie$ attack, we suggest that AES-GCM nonces should be used in counter mode to eliminate any possible nonce reuse.

ACKNOWLEDGMENT

This work was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2019-0-00533, Research on CPU vulnerability detection and validation). This work was also supported by the research fund of Signal Intelligence Research Center supervised by the Defense Acquisition Program Administration and Agency for Defense Development of Korea.

REFERENCES

[1] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: HTTP cookie hijacking and the exposure of private information," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 724–742.

[2] T. Dierks, "The transport layer security (TLS) protocol version 1.2," 2008.

[3] J. Hodges, C. Jackson, and A. Barth, "HTTP strict transport security (HSTS)," Tech. Rep., 2012.

[4] K. Spett, "Cross-site scripting," *SPI Labs*, vol. 1, pp. 1–20, 2005.

[5] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 98–113.

[6] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 475–488.

[7] K. Bhargavan and G. Leurent, "Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH," in *Network and Distributed System Security Symposium–NDSS 2016*, 2016.

[8] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni *et al.*, “DROWN: Breaking TLS using SSLv2.”

[9] B. Möller, T. Duong, and K. Kotowicz, “This POODLE bites: exploiting the SSL 3.0 fallback,” 2014.

[10] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, “A messy state of the union: Taming the composite state machines of TLS,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 535–552.

[11] T. Duong and J. Rizzo, “Here come theŁ ninjas,” 2011.

[12] N. J. Al Fardan and K. G. Paterson, “Lucky thirteen: Breaking the TLS and DTLS record protocols,” in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 526–540.

[13] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta *et al.*, “Imperfect forward secrecy: How diffie-hellman fails in practice,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 5–17.

[14] T. Jager, J. Schwenk, and J. Somorovsky, “On the security of TLS 1.3 and QUIC against weaknesses in PKCS# 1 v1. 5 encryption,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1185–1196.

[15] J. Jeong, H. Kwon, H. Shin, and J. Hur, “A practical analysis of TLS vulnerabilities in korea web environment,” in *International Workshop on Information Security Applications*. Springer, 2016, pp. 112–123.

[16] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, “Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS,” 2016.

[17] J. Salowey, A. Choudhury, and D. McGrew, “AES galois counter mode (GCM) cipher suites for TLS,” Tech. Rep., 2008.

[18] A. Joux, “Authentication failures in NIST version of GCM,” *NIST Comment*, p. 3, 2006.

[19] E. Rescorla, “The transport layer security (TLS) protocol version 1.3–draft-ietf-tls-tls13-05,” 2015.

[20] C. J. Benvenuto, “Galois field in cryptography,” *University of Washington*, 2012.

[21] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, “GCM nonce checker,” https://gcm.tlsfun.de.

[22] “Quantcast,” 2017.

[23] L. Bello, M. Bertacchini, and B. Hat, “Predictable prng in the vulnerable debian openssl package: the what and the how,” in *the 2nd DEF CON Hacking Conference*, 2008.

[24] J. Melià-Seguí, J. Garcia-Alfaro, and J. Herrera-Joancomartí, “A practical implementation attack on weak pseudorandom number generator designs for epc gen2 tags,” *Wireless personal communications*, vol. 59, no. 1, pp. 27–42, 2011.

[25] P. Biondi, “Scapy,” 2011.

[26] E. A. Young, T. J. Hudson, and R. Engelschall, “OpenSSL: The open source toolkit for SSL/TLS,” 2011.

[27] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, “nonce-disrespect git-hub,” https://github.com/nonce-disrespect/nonce-disrespect.

[28] E. Rescorla, “TLS 1.3,” https://www.ietf.org/proceedings/92/slides/slides-92-tls-3.pdf, 2015.

[29] A. Langley, “QUIC and TLS,” https://www.ietf.org/proceedings/92/slides/slides-92-saag-5.pdf, 2015.

[30] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, “QUIC: A UDP-based secure and reliable transport for HTTP/2,” *IETF, draft-tsvwg-quic-protocol-02*, 2016.

[31] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, “How secure and quick is QUIC? provable security and performance analyses,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 214–231.

[32] M. Fischlin and F. Günther, “Multi-stage key exchange and the case of google’s QUIC protocol,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1193–1204.

[33] N. Ferguson, “Authentication weaknesses in GCM,” 2005.

[34] S. Gueron and V. Krasnov, “The fragility of AES-GCM authentication algorithm,” in *Information Technology: New Generations (ITNG), 2014 11th International Conference on*. IEEE, 2014, pp. 333–337.

[35] M. Bellare and B. Tackmann, “The multi-user security of authenticated encryption: AES-GCM in TLS 1.3,” in *Annual Cryptology Conference*. Springer, 2016, pp. 247–276.

[36] S. Gueron and Y. Lindell, “GCM-SIV: Full nonce misuse-resistant authenticated encryption at under one cycle per byte,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 109–119.

[37] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, “Cookiext: Patching the browser against session hijacking attacks,” *Journal of Computer Security*, vol. 23, no. 4, pp. 509–537, 2015.

[38] S. Tang, N. Dautenhahn, and S. T. King, “Fortifying web-based applications automatically,” in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 615–626.

[39] Y. Mundada, N. Feamster, and B. Krishnamurthy, “Half-baked cookies: Hardening cookie-based authentication for the modern web,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 675–685.

[40] S. Calzavara, G. Tolomei, A. Casini, M. Bugliesi, and S. Orlando, “A supervised learning approach to protect client authentication on the web,” *ACM Transactions on the Web (TWEB)*, vol. 9, no. 3, p. 15, 2015.

[41] Y. Zhou and D. Evans, “Why arent HTTP-only cookies more widely deployed,” *Proceedings of 4th Web*, vol. 2, 2010.

[42] L. N. Egea, “OFFENSIVE: Exploiting changes on DNS server configuration,” in *Black Hat Asia*, 2014.

Hyunsoo Kwon received the B.S. degree from Chung-Ang University, Seoul, Korea, in 2014 and the M.S. degree from Korea University, Seoul, South Korea, in 2016 all in Computer Science. He is currently pursuing an Ph.D. degree in the Department of Computer Science and Engineering, College of Informatics, Korea University, Korea. His research interests include information security, network security, and cloud computing security.

Hyunjae Nam received the B.S. degree from Chung-ang University, Seoul, South Korea, in 2016, and the M.S. degree from Korea University, Seoul, South Korea, in 2018 all in Computer Science. He is currently pursuing an Ph.D. degree in the Department of Computer Science and Engineering, College of Informatics, Korea University, Korea. His research interests include network security and information security.

Sangtae Lee received the B.S. degree from Chung-ang University, Seoul, South Korea, in 2015. He is currently pursuing an M.S. degree in the Department of Computer Science and Engineering, College of Informatics, Korea University, Korea. His research interests include network security and information security.

Changhee Hahn received the B.S. and M.S. degrees from Chung-Ang University, Seoul, South Korea, in 2014 and 2016, respectively, all in Computer Science. He is currently pursuing an Ph.D. degree in the Department of Computer Science and Engineering, College of Informatics, Korea University, Korea. His research interests include information security, network security, and cloud computing security.

Junbeom Hur received the B.S. degree from Korea University, Seoul, South Korea, in 2001, and the M.S. and Ph.D. degrees from KAIST in 2005 and 2009, respectively, all in Computer Science. He was with the University of Illinois at Urbana-Champaign as a postdoctoral researcher from 2009 to 2011. He was with the School of Computer Science and Engineering at the Chung-Ang University, South Korea as an Assistant Professor from 2011 to 2015. He is currently an Associate Professor with the Department of Computer Science and Engineering at the Korea University, South Korea. His research interests include information security, cloud computing security, network security, and applied cryptography.