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

HTTP, known as Hypertext Transfer Protocol cookies are generally used on the web to enhance efficient communication between client & server. It contains private & sensitive data about users. In the research paper, they find a proposal that - a novel cookie hijacking attack named “rotten cookie” which deactivates cookie flags even if they’re protected by Transport Layer Security.

HTTP cookie is a small piece of data which is sent to the user's web browser via a server. In our paper, the authors focused on “HttpOnly flag” - a cookie with a secure flag which ensures a cookie can’t be accessed by client-side APIs i.e., JavaScript. It protects the cookies against cookie theft via eavesdropping over HTTP. If the Secure flag isn’t properly set in the Set-Cookie header, a man-in-the middle attacker disguised as a legitimate HTTP server can obtain private cookies through HTTP. Although HTTP cookies are protected through a secure HTTPS connection with the additional mechanisms. Another security breach has been discovered in which an attacker can make a cookie set without a cookie flag. In a cookie cutter attack, either cookies are encrypted by TLS or not, miscommunication between TLS & HTTP allows the attacker to remove a cookie flag simply by closing the connection. A “rotten cookie attack” enables the attacker to invalidate cookie flags even if they’re encrypted by TLS by exploiting insecure mechanisms within HTTP & the faulty implementation of AES-GCM. AES-GCM is currently the most widely used cipher for symmetric authentic cited encryption in TLS. In fact, if the same nonce is used again at least once in the encryption process, the attacker can extract the authentication key for the session & invalidate the cookie flag, It’s a common scenario in practice on many HTTPS servers. Then the authors perform a rotten cookie attack against 5 major web browsers & 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.

Here they find HTTPS responses in various browsers can be used as a trigger point to break the security of cookies by deactivating the cookie flags & describe the attack scenario based on cases that may lead to security failure & also demonstrate the practicality of the attack by examining Quantcast's top 56,000 websites with major web browsers. They conduct an in-depth assessment of the 5 major browsers & 10 popular web applications. They proposed an approach to mitigate their attack scenario & explain how major browsers should handle HTTP responses correctly. They provide the information necessary to understand the attack scenario. To understand how the faulty implementation of AES-GCM can weaken the integrity of messages, known as a forbidden attack, they briefly explain how the AES-GCM cipher works in TLS, which is the most widely used protocol for secure channels. TLS can be classified into 2 protocols: handshake & record protocol.

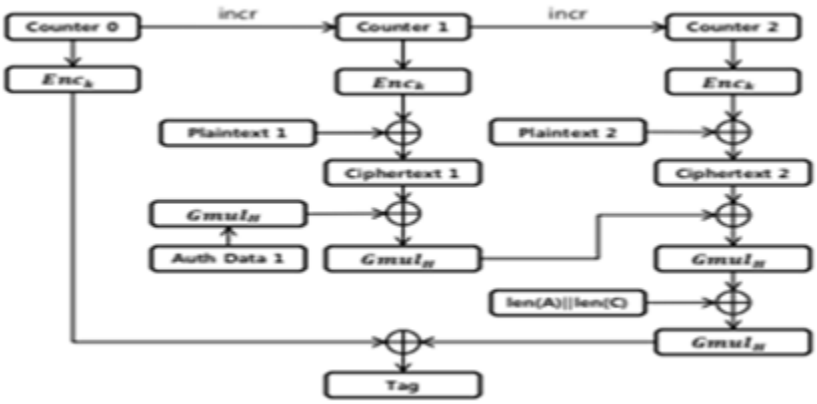
To negotiate the cipher suite that’s used in the record protocol & to provide a mechanism for mutual authentication between client & server, the handshake protocol should be run before sending/receiving encrypted data through the record protocol. ServerHello message sends by server contains a fresh server nonce & handshake parameter including the TLS version If the client verifies the server's certificates successfully, then it randomly generates a pre-master-secret, encrypts with the public key of the server, & sends it to the server in a clientele change message. If a server sends a duplicate 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.

After the handshake protocol & 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 & MAC functions. Decipher mode computes the additional authenticated data, ciphertext, & plaintext respectively as follows: additional authenticated data = seq-num plaintext. Length ciphertext: = AEAD-Encrypt (write-key, nonce, plaintext, additional authenticated data) plaintext: = AEAD-Decrypt (write-key, nonce, ciphertext, additional authenticated data) AEAD-Encrypt & AEAD-Decrypt are the encryption & decryption functions in Decipher mode, respectively.

Then, how AES-GCM works in TLS1.2 & introduces forbidden attacks. In a forbidden attack, the attacker can compute the authentication key for a session if the same nonce is re-used. AES-GCM is a mode of operation for block ciphers which provides AEAD. It’s a standardized cipher suite in TLS1.2. To provide both authentication & data encryption, AES-GCM consists of two phases: encryption & authentication.

The AES-GCM encryption process for the generation of one ciphertext fragment consisting of n plaintext blocks is as by firstly Generating a 96-bit long IV (implicit + explicit compo-nets). & then 128-bit long counter blocks Ji, where Ji =IV kcnt, & cnt = (i

+ 1) mod 232, for i ∈ {0, . . . n}. Next, it generates i-the ciphertext block Ci = Enck (Ji) ⊕ Pi, where Enck (·) is AES encryption with symmetric key k & Pi is the i-the plaintext block. Finally give the output ciphertext C = C1kC2kC3k . . . kCn.



After the ciphertext blocks are created, the Galois Field (GF) multiplication function combines the authenticated data to produce an authentication tag in GF (2128) defined by the irreducible polynomial f = 1+α+α 2 +α 7 +α128. The AES-GCM authentication process for a ciphertext fragment with blocks of additional authenticated data is described as follows:

1. Generate authentication hash key H = Enck (0128).
2. Starting with X0 = 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): Xi = GmulH (Xi−1 ⊕ Ai), for i ∈ {1, . . . m}, where GmulH (·) is the GF multiplication function & Ai is its authenticated data block.
3. Compute GF multiplications over ciphertext blocks Ci consisting of n blocks: Xi+m = GmulH (Xi+m−1 ⊕ Ci), for i ∈ {1, . . . n}. & final multiplication using the bit-lengths of A & C Xm+n+1 = GmulH(Xm+n _ (len(A)klen(C))).
4. Output authentication tag T = Xm+n+1 ⊕ Enck(J0). The final output of the AES-GCM cipher is ciphertext C concatenated with authentication tag T: CkT.

Then if the attack is a forbidden attack. Even. A forbidden attack is damaging to AES-GCM in theory, it isn't considered a practical threat because of the implicit assumption that the nonce in the AES-GCM cipher will never be reused. 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: g(X) = A1Xm+n+1 + · · · + AmXn+2 + C1Xn+1 + · · · + CnX2 + LX + Enck (J0). If the attacker successfully recovers authentication key H, he can carry out a forgery on any valid ciphertext by computing valid authentication tag T.

Then they move to another section, they explain a security weakness of web browsers that can be potentially exploited by our cookie theft attack that invalidates cookie flags.

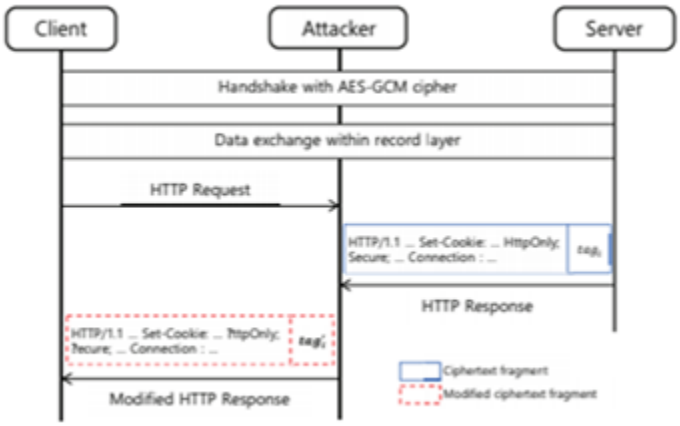
To support flexibility & scalability, most HTTP-based software does not check all of the header options & the integrity of the HTTP response message. they found that almost every major web browser, including Chrome, Firefox etc. ignores uninterpretable sections in the response message (for example: "S?cure") & accepts the rest without any rejection.

Then they found that major browsers accept incorrect HTTP response message formats without any rejection. In order to prevent cookie-cutter attacks, browsers should at least check the end of the HTTP response's header message. Most major browsers have been updated recently to check if the end of header message is truncated or not; however, they observed that all of the major browsers accept the following HTTP message even if it contains multiple status-lines.

So, they found that most current web browsers do not correctly check the integrity of header messages & the format of the HTTP response message. 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 & is independent of the transport layer. This

dependency of security on different layers is a potential threat to the security of cookies if they do not consider the above weaknesses of the browsers & the vulnerabilities of TLS.

This section is based on 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, the rotten cookie scenario is given following figure:



The rotten cookie attack is based on a forbidden attack where the client & the server use the AES-GCM cipher suite in TLS. Let the server may re-use the same GCM nonce either due to faulty implementation or at random. After locating a vulnerable server & a target client, the attacker performs an MITM attack, manipulating TLS fragments & attempting to steal private cookies from the victim by removing cookie flags. C, A & S are the target client, the MITM attacker, & the vulnerable server, respectively. Above figure depicts our attack scenario based on a forbidden attack, which proceeds as follows:

1. If C initiates a TLS connection with S, 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, A selects more than 1 TLS fragments where the same nonce is used.
3. 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, A can finally compute a valid authentication key & conduct a cookie theft attack.
5. Because A can determine the location of the first byte of both HTTP Only & Secure, A replaces the first byte with any random value & generates a valid tag for the modified fragment including the original additional authenticated data with the previously computed authentication key.
6. A sends the modified fragment to C. C then discards the uninterpretable flag in the response message & sets the cookie without the flags.
7. Finally, A is able to conduct an MITM attack over HTTP / XSS attack & then obtain the target private cookie of C.

As described above, an attacker is able to steal private cookies if the nonce is reused, even if they are encrypted securely.

In the real world, they found that a non-negligible number of servers still generate duplicate nonce. Research among the 4 different types R-4CTR & 8CTR types are secure against our attack because both types can be seen as variations of the counter mode.

To prove the possibility of the proposed attack in real-world HTTPS sessions, they implement an MITM attacker in Python with Scapy. In addition, to recover the authentication key from the record data, they leverage the factoring functionality given by. They choose five target clients Cs, Chrome, Internet Explorer, Edge, Firefox, & Safari, which are the most popular web browsers in the real world. When S transmits a cookie containing Secure & HttpOnly flags to the five web browsers through a TLS connection, they employ MITM attacker A to accomplish a rotten cookie attack.

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

Before the attack starts, if C receives an original cookie message from S, the cookie would be set with Secure & HttpOnly flags as shown in Figure. They can also observe how the cookie would be parsed at the bottom of Figure. Our rotten cookie attack then progresses as follows:

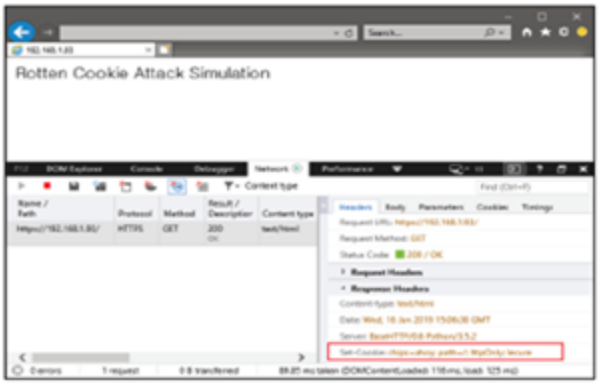
1. When the AES-GCM cipher is negotiated under a TLS connection, as shown in Fig.5, A holds all packets sent from S to C until at least two record data including the cookie value are received.
2. A then extracts additional authenticated data, the ciphertext, & the tag from the record fragments, which is encrypted under the same session key, & 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, A replaces the first byte of the cookie flags with an arbitrary character using an XOR operation & utilizes the recovered authentication key to generate a validate tag for the altered content. Where data is the encrypted HTTP content including the cookie value, is the XOR operation, & i & j indicate the location of the first character of HttpOnly & Secure, respectively.
4. After receiving the manipulated cookie, C accepts the cookie values except for the uninterpretable sections. They can also see that the HTTPS fragment modified by the XOR operation was parsed normally with "LttpOnly" & "Lecure", & Chrome ignored the cookie flag values. In this way, they conduct a rotten cookie attack on Internet Explorer, Edge, Firefox, & Safari.

Then they conduct a rotten cookie attack on Internet Explorer, Edge, Firefox, & Safari, & the resulting screenshots are presented in below Figure's subsection a, b, c, & d respectively. As they expected, all of the 5 major web browsers accept the cookie without a Secure flag & Http Only flag, which demonstrates the practicality of our attack in the real-world web environment.

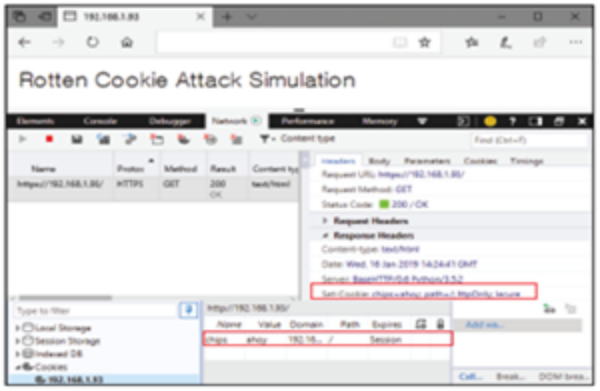
After seeing the proposed attack, they conduct an in-depth assessment of how major browsers check the integrity & format of HTTP messages. They examine the five main web browsers, also investigate how popular websites set cookie flags securely, then investigate whether the browsers are secure against cookie-cutter attacks by forcing them to accept the string "Rnrn" message over HTTPS. Finally, investigate whether they accept incorrectly formatted HTTP messages created by concatenating two HTTP response messages. All of the browsers do not verify the integrity of the Set-Cookie header message & accept cookies even if they include sections that're 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.

It's important for the browsers to check the entire format of an HTTP response message as well as investigate whether the browsers accept HTTP response messages that include multiple status lines & how they handle missing empty lines over HTTPS. They found that all browsers accept multiple status lines & strictly check for the empty line except "Internet Explorer".

Consider a private cookie to be a cookie which is set with a Secure or "HttpOnly" flag. Firstly, server administrators thought that their servers might be secure against XSS attacks even if the cookies are set without an HttpOnly flag so long as they strictly screen the malicious script code on their 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. In the research paper, from the table "The use of cookies on popular website's rank" shows the investigation indicates that 9 out of 60 private cookies didn't contain a secure flag.

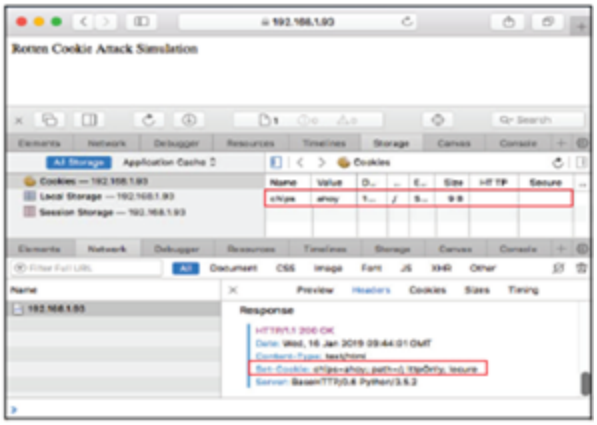


(a) Cookie status in Internet Explorer



(b) Cookie status in Edge

(c) Cookie status in Firefox



(d) Cookie status in Safari

So finally, 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. Thus, they 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, they suggest that AES-GCM nonces should be used in counter mode to eliminate any possible nonce reuse.