PADERBORN UNIVERSITY
The University for the Information Society

Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group System Security

Master's Thesis

Submitted to the System Security Research Group
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Evaluation of TLS session tickets

Simon Nachtigall

Supervisors: Prof. Dr-Ing. Juraj Somorovsky - Paderborn University
Prof. Dr. Kenneth Paterson - ETH Zürich
Sven Niclas Hebrok - Paderborn University
Marcel Maehren - Ruhr University Bochum
Robert Merget - Ruhr University Bochum

Paderborn, August 5, 2021

Abstract

To establish a secure TLS connection between client and server, both parties have to perform a handshake where they establish a common secret. However, performing the secret establishment every time is very costly and increases the latency. For that reason, TLS offers session resumption mechanisms that allow both parties to reuse a previously established secret. A widely used resumption mechanism is *session tickets*. After the secret has been established, the server issues a session ticket containing the secret to the client. In the session resumption, the client sends the ticket back to the server. It is essential for the confidentiality of the sessions that the server encrypts the ticket with a Server Ticket Encryption Key (STEK). In 2020, Fiona Klute found a vulnerability in GnuTLS where the STEK was initialized with all-zeros allowing an attacker to decrypt recorded sessions retrospectively [11]. This motivates us to evaluate the TLS session ticket ecosystem in more detail. Therefore, we present different vulnerabilities that might appear in session ticket handling of webservers. Next, we implement test suites for the presented vulnerabilities and evaluate them for the Tranco Top Million hosts in a large-scale scan. Finally, we present the results for the different evaluated vulnerabilities and other interesting findings in our thesis. We discovered that similarly to GnuTLS several thousand domains hosted by AWS used an all-zero STEK to encrypt their session tickets.

Keywords: TLS, Session Ticket, Session Resumption

Official Declaration

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Paderborn, 5.8.2021	Simon Nachtigall
Date	Simon Nachtigall

Contents

1 Introduction

TLS (Transport Layer Security) [20] is one of the most used cryptographic protocols to ensure secure communication on the Internet. One big application area of TLS is to guarantee the security of web applications in HTTPS. The three main security goals of TLS are confidentiality, authenticity and integrity of the exchanged data. In order to set up a secure connection, both communication parties need to establish a common secret in a so-called *handshake*. In TLS, there are two different types of handshakes: First, the communication parties can perform a full handshake, in which they establish a new secret between both parties. Second, they can do a session resumption, which means that both parties reuse the common secret they have established in a previous session.

The main motivation for using session resumption is to reduce latency in a TLS connection since a full handshake needs time-expensive public key cryptographic operations. In practice, session resumption consumes less than half of the time of a full handshake. Moreover, the CPU time consumed on the client is only about 4% compared to a full handshake, which is especially useful for mobile users with limited battery power [19]. In order to perform a session resumption, both communication partners need to remember the session state including the established secret from a previous session.

The session ticket mechanism is one way to achieve a session resumption without storing any session state on the server-side. It is defined in the RFC 5077 as an extension of the TLS protocol [23]. In 2018 78% of the Alexa Top Million of TLS-enabled websites supported the session ticket mechanism [25]. If the client wants to perform a session resumption in the future, it asks the server for an encrypted session ticket containing the session state of the server. The client stores the ticket of the server along with its session state. When the client then reconnects to the server, it sends the received session ticket back to the server, which decrypts the ticket to get the session state. Now both sides are in possession of the session state again to resume the session. For the security of the resumed session, it is crucial that the server encrypts the session ticket with a Server Ticket Encryption Key (STEK). Otherwise, if attackers are able to read the contents of a session ticket, they could decrypt confidential data transmitted during the session.

The main disadvantage of the session ticket mechanism is that it provides no *forward secrecy* for TLS connections [10]. *Forward secrecy* essentially means that prior sessions remain secret even if an attacker corrupts one party and gets in possession of the long-term secret as for example the STEK. If session tickets are used, then a compromise of the long-term STEK will lead to a compromise of prior sessions. In order to achieve the properties of *forward secrecy* while using session tickets, the RFC 5077 recommends to rotate the STEK at least every 24 hours. This means that forward secrecy is not fully achieved, but the impact of a compromise is reduced. If an attacker is able to compromise the STEK, then only sessions using session tickets protected by the STEK in this 24 hours interval are affected. Besides, the security impact of a STEK compromise depends on the used TLS version. Sessions in the new TLS version 1.3 are significantly less affected than in the older TLS version 1.2. In TLS 1.2, session tickets contain the secret that is used to protect the initial session, where the ticket is issued, and the resumed session, where the ticket is redeemed. If an attacker compromises the STEK, then it can decrypt all corresponding initial and resumed sessions. In TLS 1.3, session tickets typically contain the secret that is only used to protect the early application data, data send in the very first roundtrip of the session resumption. If the STEK is compromised, then an attacker can only compromise the session's early application data.

In a certain GnuTLS version, Fiona Klute found a bug, which affected the security of the session resumption mechanism [6, 11]. The server used an all-zero STEK in the initial key rotation interval, allowing an attacker to decrypt the session tickets and get in possession of the included session secrets. With them, the attacker can decrypt the application data of corresponding sessions. This bug indicates that there might be additional security weaknesses in the area of session tickets. This motivates us to evaluate the session ticket handling of different TLS libraries and servers in the wild. We want to evaluate in a large-scale scan if other implementations are vulnerable to different kinds of attacks.

1.1 Current State of Research

The effects of TLS session tickets on forward security have been studied in several articles [10, 19, 27]. Besides, several large-scale evaluations of TLS session resumption mechanisms including TLS session tickets were performed in the past. In the following, we describe three different large-scale evaluations of TLS session tickets.

In 2016, Springall et al. have evaluated how performance enhancement mechanisms as session tickets weaken the forward secrecy of TLS in practice [24]. Therefore, they analyzed how often the Alexa Top Million websites rotated their STEK to achieve the properties of forward secrecy. Only 41% of the websites rotated the STEK every day as recommended in the RFC 5077. They showed that about 20% from the Alexa Top 100 reused the STEK for at least 30 days. In addition to that, they evaluated how many servers share their STEK. If the STEK of one server is compromised, then sessions of all servers sharing the same STEK would be affected. The authors conclude that performance enhancement mechanisms as session tickets reduce the overall forward secrecy property of TLS connections significantly in practice. Additionally, the authors provide an overview of how different TLS-implementations select their STEK identifiers which are included in the session tickets. They also described that Microsoft webservers encode their session tickets as ASN.1 objects.

In 2018, Sky et al. evaluated how users can be tracked across the web via TLS session resumption mechanisms as session tickets [26]. For this, they evaluated the TLS server configurations of the Alexa Top Million hosts. They found out that 65% of all users can be tracked permanently via TLS session resumption mechanisms.

Valsorda found in 2016 a vulnerability in the session ticket mechanism of the F5 TLS stack that allowed an attacker to extract 31 bytes of uninitialized memory at a time [9]. Due to the similarity to the Heartbleed bug [8], the vulnerability was named Ticketbleed. Valsorda performed a large-scale scan of the Alexa Top Million list to evaluate how many webservers were vulnerable.

1.2 Thesis Goals

Our main goal is to evaluate the session ticket handling of different TLS libraries and TLS servers in the wild. The recently found bug in GnuTLS [11] indicates that there might be further vulnerabilities in this area. The evaluation process consists of several steps:

- We do a source code analysis of nine TLS libraries regarding their session ticket handling. Additionally, we evaluate especially the key rotation mechanisms of the three libraries OpenSSL, GoTLS[1] and MbedTLS in more detail. Our goal is that we get a better understanding of how webservers in the wild handle their session tickets.

- We propose several possible vulnerabilities that can appear in the session ticket handling of webservers in the wild.

- We implement test suites for the proposed vulnerabilities into TLS-Scanner in order to evaluate if a TLS server is vulnerable to our proposals.

- We scan the internet using TLS-Crawler in order to evaluate how many servers are vulnerable to our implemented proposals.

[1]https://pkg.go.dev/crypto/tls

1.3 Structure of the Thesis

2. Background (more detailed overview of how session resumption and session tickets work in TLS 1.2 and TLS 1.3)
3. Library Analysis (Source code analysis of different TLS-libraries)
4. Proposal of Possible Security Vulnerabilities in session ticket handling
5. Implementation of Test Suites for Proposed Vulnerabilities (used tools, detailed description of implementation)
6. Large-Scale Scanning of Internet (used tools, evaluation results)
7. Conclusion and Future Work

2 Background

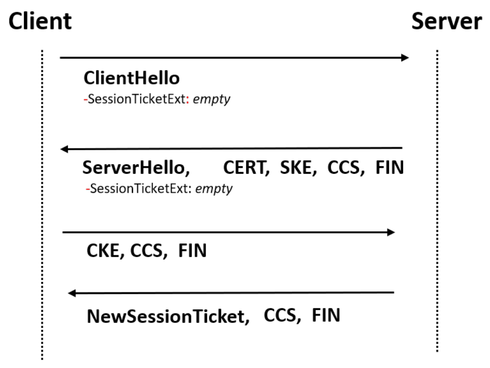
In this chapter, we give a detailed background on session resumption mechanisms and session tickets. We study in more detail the session ticket handling in TLS 1.2 and 1.3 and evaluate the differences between the two versions. Finally, we give a background to the padding oracle attack which we later need in the thesis.

2.1 Session Resumption Mechanisms in TLS

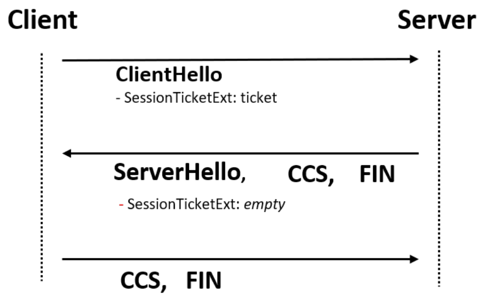
In the RFC 5077 [23] two main mechanisms for session resumption in TLS are defined.

- **Session caches**: In this mechanism, the client and server store all the cryptographic parameters of the previous session inside their session cache. When the client resumes the session, it only needs to send the session identifier to the server. Then, the server only looks up the cryptographic parameters of the corresponding session. With the previously established secret, the client and server are now able to resume the session. The main disadvantage of this mechanism is that the server needs to store for every connection all cryptographic parameters, which becomes for servers with many connections infeasible.

- **Session tickets**: The *session ticket* mechanism allows the server to outsource the storage of cryptographic parameters to the client. In the initial session, the server sends a session ticket with all necessary cryptographic parameters to the client and then discards all these parameters. The client stores the received session ticket and for itself all necessary cryptographic parameters for the session resumption. In the session resumption, the client sends the session ticket back to the server, so that both parties are in possession of the secret again and can securely resume the session. The session tickets are not sent in plain. The server encrypts it with a symmetric key called Server Ticket Encryption Key (STEK). It is crucial for the security of the resumed session that only the server is able to read the contents of the session ticket.



(a) Ticket issuance in TLS 1.2



(b) Session resumption with session ticket in TLS 1.2

Figure 2.1: Two handshake types in TLS 1.2

2.1.1 Session Tickets in TLS 1.2

Session resumption with session tickets was introduced in TLS 1.2 with RFC 5077 [23]. The RFC describes how the tickets can be issued and redeemed for session resumption. Therefore, two different handshake flows were introduced:

- **Ticket issuance:** The ticket issuance happens in a normal full TLS handshake. If a client wants to perform session resumption in future connections, it sends its empty session ticket extension after the ClientHello to the server (see Figure 2.1(a)). If the server supports session tickets, it answers with an empty session ticket extension. Then, the server sends the *NewSessionTicket* message (see Listing 2.2), which contains the issued session ticket, to the client. The *NewSessionTicket* message is sent before the ChangeCipherSpec message which means that the message is not encrypted. The client only needs to store the received session ticket until the next session resumption.

- **Session resumption:** When a client wants to resume the session, it sends its session ticket after the ClientHello to the server (see Figure 2.1(b)). If the server accepts the session ticket, it answers with an empty session ticket extension. Then, a secure connection is established and both can exchange securly application data.

```
struct {
  uint32 ticket_lifetime_hint;
  opaque ticket<0..2^16-1>;
} ticket;
```

Listing 2.2: TLS 1.2 *NewSessionTicket* message from RFC 5077

Recommended Ticket Structure according to RFC The usage and construction of session tickets in TLS 1.2 are described in the fourth chapter of the RFC 5077 [23]. The RFC recommends to structure the session ticket as following (see Listing 2.3):

```
struct {
  opaque key_name[16];
  opaque iv[16];
  opaque encrypted_state<0..2^16-1>;
  opaque mac[32];
} ticket;
```

Listing 2.3: recommended Session ticket format in RFC 5077

The session ticket contains four different values:

- `key_name`: a 16-Byte STEK identifier so that the server recognizes the key it needs to decrypt the encrypted state.
- `iv`: 16-byte initialization vector which is used in the encryption of the encrypted_state.
- `encrypted_state`: The session state contains all necessary cryptographic parameters of the server in order to resume the session. This state is encrypted with the STEK and IV to an `encrypted_state` since only the server is authorized to read the contents of the session state. The RFC 5077 mentions in the description of the ticket structure that the size of the encrypted state (2-Byte) shall also be included inside the session ticket.
- `mac`: session tickets are also integrity protected. The Message Authentication Code (MAC) is calculated over all three previously named fields.

It is recommended to use AES-128 CBC as the encryption algorithm and HMAC-SHA-256 as the authentication algorithm. Thus, the server needs to recognize two keys for both algorithms in the decryption process.

Additionally, the RFC recommends a structure for the session state.

```
struct {
  ProtocolVersion protocol_version;
  CipherSuite cipher_suite;
  CompressionMethod compression_method;
  opaque master_secret[48];
  ClientIdentity client_identity;
  uint32 timestamp;
} StatePlaintext;
```

Listing 2.4: recommended Session ticket format in RFC 5077

The session state contains all necessary cryptographic parameters of the original session including the selected `protocol_version` and `cipher_suite` The `master_-secret` (48 bytes long) is used to derive all necessary key material for the resumed session. Furthermore, the server uses the `timestamp` of the state to recognize expired tickets. However, this structure is only a recommendation. The client does not need to understand or modify the encrypted session state, so every server implementation can decide by itself how to structure the session state.

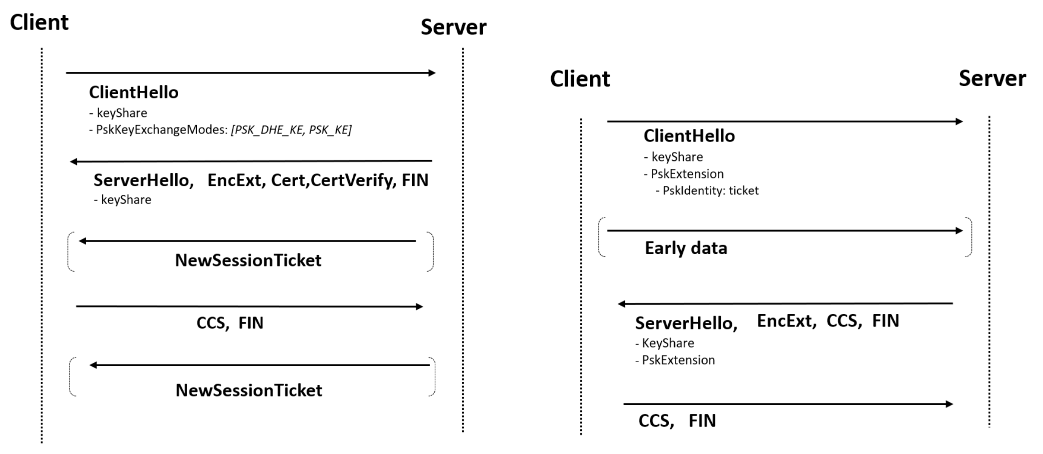
Forward Secrecy in TLS 1.2 The implementation of session resumption with session tickets in TLS has some important consequences for the security of TLS sessions[10]. Since the session tickets contain the master secret for deriving the session keys, it is crucial that the STEK is only accessible for the server. Otherwise, if the attackers could retrieve the STEK, they could effectively break all connections using session tickets. They could also decrypt connections established with perfect forward secure cipher suites. In case of the compromise of the STEK, this has two important consequences for sessions in TLS 1.2:

1. Resumed sessions only use the master secret in the session ticket to derive the session keys. There is no additional key exchange performed in order to update the session keys as in TLS 1.3. Therefore, resumed sessions do not offer forward secrecy against compromise of the STEK, which means an attacker is able to read the whole conversation of the resumed session.
2. In the initial session the master secret is used to derive the session keys. The session ticket for session resumption is issued with the same master secret. Thus, initial sessions are not considered as forward secure against the compromise of STEK. An attacker could decrypt messages of the resumed session and additionally of the initial session with a compromised STEK. A straightforward solution for this problem could be to issue the session ticket with the hashed master secret so that the attacker is not able to reconstruct the session keys of the initial session. However, TLS 1.2 does not use this mechanism, it was only introduced in TLS 1.3

As described before, a compromise of the STEK has fatal consequences for the security of the TLS sessions. In order to mitigate the effects of a STEK compromise and achieve the properties of forward secrecy, the RFC recommends to rotate the STEK every 24 hours.

2.1.2 Session Tickets in TLS 1.3

The newest TLS protocol version 1.3 [20] is faster and more secure than the predecessor TLS 1.2 [17]. Session resumption with session tickets is still supported in TLS 1.3, but there are some differences in comparison to TLS 1.2. Especially, sessions in TLS 1.3 are much less affected by a compromise of the STEK.



(a) Full handshake with ticket issuance (b) 0-RTT session resumption with session ticket

Figure 2.5: Two handshake types in TLS 1.3

First, we have a look at how session tickets are handled in TLS 1.3:

- **Ticket issuance:** The ticket issuance in TLS 1.3 (see Figure 2.5(a)) is quite similar to TLS 1.2. The main difference is that in TLS 1.3 the session tickets are handled by the PSK extension. The client sends initially his supported *PskKeyExchangeModes* to the server. There are two types of *PskKeyExchange-Modes*: 1. *PSK_DHE_KEY*: Client and server perform an additional Diffie Hellman key exchange after resuming the session. 2. *PSK_KE*: Similar to TLS 1.2 no additional key exchange is performed in the session resumption. The server can send the *NewSessionTicket* message (see Listing 2.6) with the issued ticket either directly after the server *Finished* message (only allowed if client authentication is disabled) or after receiving the final client *Finished* message. The session ticket contains a *pre-shared key*, which is derived from the session's master secret.

- **Session resumption:** In the session resumption, the client includes the session ticket in the Pre-shared Key(PSK) extension of the ClientHello. Depending on the the selected *PskKeyExchangeMode* the client also can include a *keyShare*. Then, there are two possibilities for the client in the session resumption: First, the client can send early data on the fly to the server which is called 0-RTT session resumption. The early data is encrypted with the Pre-shared Key inside the session ticket. The server can decrypt the payload with the received session ticket. Second, the client performs the resumption handshake without sending any application data which is called 1-RTT session resumption. The session ticket is only used to provide authenticity of the server so that the certificate verification can be skipped.

```
struct {
  uint32 ticket_lifetime;
  uint32 ticket_age_add;
  opaque ticket_nonce<0..255>;
  opaque ticket<1..2^16-1>;
  Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

Listing 2.6: TLS 1.3 *NewSessionTicket* message from RFC 8446

Forward Secrecy in TLS 1.3 There are three main security differences in comparison to TLS 1.2

1. Session tickets do not contain the master secret of the initial session, they contain the hashed master secret called *pre-shared key*. An attacker is not able to reconstruct the initial master secret so that a compromise of the STEK does not affect the initial sessions anymore.

2. The *pre-shared key* inside the session ticket is only used to protect the initial message (early data) of the client. After that, the client and server can perform an additional key exchange to update the session keys. Therefore, only the initial message is not considered as forward secure against a compromise of the STEK. After updating the session keys, all sent messages are considered forward secure. Another problem is that an attacker might be able to perform a replay attack with the early data of the client. Therefore, servers have to implement countermeasures as described in Chapter 8 [20].

3. In contrast to TLS 1.2, the NewSessionTicket message is sent after the ChangeCipherSpec message which means that the message is encrypted. An attacker is only able to read the contents of the issued session ticket when the client resumes the session.

Recommended ticket structure In the RFC 8044 of TLS 1.3, there is no ticket structure recommended. It only says that the session ticket has to be self-encrypted and self-authenticated by the server. However, in practice most TLS-implementations follow the recommendations for session tickets of the RFC 5077.

2.2 Padding Oracle Attack

In this section, we describe the main concept of a padding oracle attack. Before doing that, we first give a short cryptographic background.

Block cipher A *block cipher* is a method to encrypt a block of data with a symmetric key. The encryption algorithm

$$C = Enc_k(m), \qquad k \in \{0,1\}^l, m \in \{0,1\}^b, C \in \{0,1\}^b$$

encrypts a message m of block size b with a symmetric key k and outputs a ciphertext C of size b.

The decryption algorithm

$$m = Dec_k(C)$$

decrypts a ciphertext C with key k and outputs the original message m. *AES* [3] is the most used block cipher and allows to encrypt data blocks of size $b = 128$ bits. Block ciphers only allow us to encrypt data with a fixed block size. To encrypt data of arbitrary length we have to process the block cipher using mode of operation and additionally pad the data.

Mode of Operation A Mode of Operation uses a block cipher to encrypt data of arbitrary length. There are two different types of modes of operation.

The first type of mode of operation is *Confidentiality only* modes. These modes only provide confidentiality. Cipher block chaining (CBC) is a popular example. If we additionally want to provide authenticity and integrity, we have to calculate a Message Authentication Code (MAC). There are three different ways how we can combine the encryption with the MAC:

1. Encrypt-then-MAC: First, the message m is encrypted and the MAC is calculated over the encrypted message: $C = Enc(m)||MAC(Enc(m))$.
2. MAC-then-Encrypt: The MAC is calculated over the original message m. Then, the message m with the appended MAC is encrypted:
   $C = Enc(m||MAC(m))$
3. Encrypt-and-MAC: The MAC is calculated over the original message m and appended to the encryption of m: $C = Enc(m)||MAC(m)$.

The second type is *Authenticated encryption with additional data* (AEAD) modes. AEAD modes provide confidentiality, authenticity, and integrity in combination. Thus, no additional HMAC algorithm is needed. Galois/Counter (GCM) is a popular AEAD mode.

```
01
02 02
03 03 03
04 04 04 04
05 05 05 05 05
...
```

Listing 2.7: PCKS#7 padding

Padding When block ciphers are used as AES-CBC, then only messages that fit into multiple 16-byte blocks can be encrypted. Thus, we append a padding to guarantee that we can encrypt messages of arbitrary length. A widely used padding scheme is PCKS#7. In the PKCS#7 scheme "the value of each added byte is the number of bytes that are added" [13]. If for example, 2 bytes are missing, then 2 bytes with the "02" byte value are added (see Figure 2.7).

2.2.1 Padding Oracle Attack

In the following, we will explain the main principles of a padding oracle attack in CBC. First, we look in more detail how the CBC mode of operation works. Figure 2.8

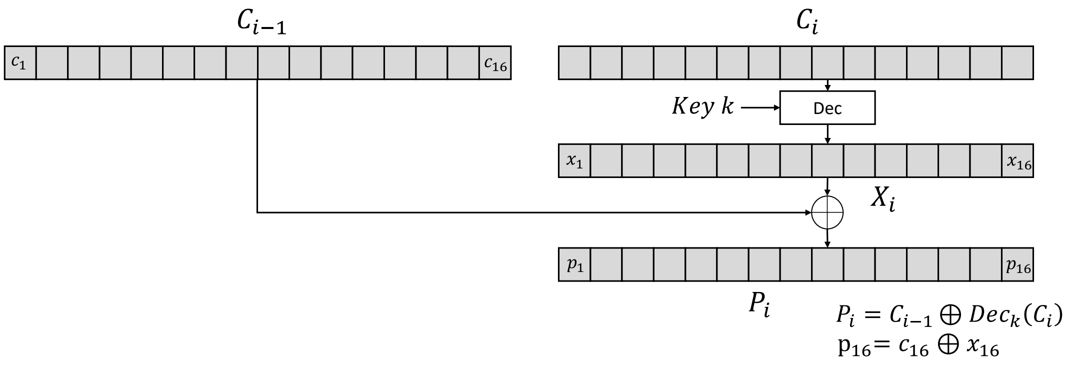


Figure 2.8: CBC decryption of block C_i of block size 16

shows the decryption of two ciphertext blocks. For ciphertext block C_i the following formula is applied to get the resulting plaintext block P_i:

$$P_i = Dec_k(C_i) \oplus C_{i-1} = X_i \oplus C_{i-1}.$$

The decryption formula for the last plaintext byte p_{16} of block C_i looks as following: $p_{16} = x_{16} \oplus c_{16}$. One property of CBC cipher suites is that they are malleable. This means that we can change the original ciphertext in a way that it decrypts to a related plaintext. It means the following for CBC cipher suites: If we modify last ciphertext byte $c'_{16} = c_{16} \oplus z$, the decrypted plaintext byte p'_{16} contains the same modification $p'_{16} = p_{16} \oplus z$. If the MAC-then-Encrypt scheme is used, then the malleability property can be used to perform padding oracle attacks [28]. An attacker flips arbitrary bits in the second last block C_{n-1} and sends the ciphertext to the server. Thereby, the attacker modifies the padding bytes included in the last ciphertext block. A server using MAC-then-Encrypt first decrypts the ciphertext and checks if the padding is valid. In one case, the attacker may have created an invalid padding. In this case, the server will detect it and normally respond with an alert message. In the other case, the attacker may have created a valid padding. Then, the server will remove padding to validate the MAC. The MAC validation will fail since the attacker has altered the ciphertext. In this case, the server may respond with an alert message. If an attacker can distinguish the responses for both cases, it can use the padding oracle to reconstruct the original plaintext. We will explain the main idea of the attack for the ciphertext block C_i. We test all 256 possible byte values for c'_{16}. For all 256 possibilities, we send block $C'_i - 1$ and C_i to the server and evaluate the responses. In one case $c'_{16} = c'_{validPad}$[1], we will create a valid 1-Byte padding at p'_{16}. In this case, the server responds differently than in the other 255 possibilities. We can reconstruct the original plaintext value p'_{16} with the following formula:

$$p_{16} = c'_{validPad} \oplus c_{16} \oplus 0x01$$

We can continue this procedure by creating a valid 2-Byte padding to recover p_{15} and so on until we have recovered all plaintext bytes.

The attack can be prevented by using the Encrypt-then-MAC scheme. In this scheme, the server first validates the MAC and then decrypts the ciphertext. If an attacker has flipped bytes in the ciphertext, the MAC validation will always fail and thus the server will respond uniformly.

[1]Note that there are edge cases, where we may create more than 1 valid padding.

3 Library Analysis

In this section, we evaluate how different TLS-libraries handle their session tickets. We perform a source code analysis and analyze the libraries for the following features:

1. Session Ticket Format
2. Authenticated Encryption
3. Key Rotation
4. Randomness
5. Replay Protection.

In total, we evaluate nine different TLS-libraries. However, we do not evaluate all TLS-libraries to the same extent for all features. For the first two features, we evaluate all nine TLS-libraries since we only need to analyze the code superficially. For the key rotation feature, we evaluate for all TLS-libraries if they implement a key rotation mechanism. Additionally, we explain the implemented key rotation mechanisms of three TLS-libraries OpenSSL[1], GoTLS (*crypto/tls* package of Go)[2] and MbedTLS[3] in more detail. The Randomness feature is only evaluated for the three TLS-libraries as well. The final Replay protection feature is only evaluated for OpenSSL since the majority of the evaluated libraries do not support early data. Our goal is to get a better understanding of how different webservers in the wild handle session tickets and if they fulfill the recommendations from the RFC 5077. Moreover, it may be possible that we find security vulnerabilities directly in the code.

3.1 Session Ticket Format

In this section, we evaluate the session ticket format of nine TLS-libraries and compare them to the recommended session ticket format in the RFC 5077 (see Listing 2.3). The RFC 5077 mentions in the description of the ticket structure that the size of the encrypted state (2-Byte) shall also be included inside the session ticket. However,

[1]https://www.openssl.org/
[2]https://pkg.go.dev/crypto/tls
[3]https://tls.mbed.org/

this is not explicitly shown in their ticket structure listing (see Listing 2.3). Thus, if webservers do not include the size, but all the other requirements are fulfilled, we say nevertheless, that the server fulfills the recommendation.

Table 3.1 shows the evaluation results. Four TLS-libraries OpenSSL, GoTLS, BoringSSL, and MatrixSSL follow the ticket format recommendation. However, they do not include the size of the encrypted state as the majority of the evaluated libraries. Only GnuTLS and MbedTLS include the size in their session ticket. The session ticket format of GnuTLS slightly differs from the recommendation. Only the MAC size is different because another MAC algorithm is used. The S2N ticket format also only differs in the MAC size. The MbedTLS ticket format does not fulfill the ticket format recommendations. The three fields *key_name*, *iv*, and *mac* all have a smaller size. The ticket format in RusTLS does not include any *key name* at all. Furthermore, the *iv* size is smaller than recommended. BotanSSL uses a totally different ticket format as it includes new fields such as the 8-byte *magic_constant* and the 16-byte *key_seed.* Likely, BotanSSL maintains their STEKs in a different way compared to the other libraries. However, we did not evaluate this in more detail.

3.2 Authenticated Encryption

In our source code analysis, we evaluate how the different TLS-libraries encrypt and authenticate their session tickets. First, we describe the used encryption algorithms. Then, we describe the used authentication algorithms to protect the session ticket. Finally, we will evaluate if the Encrypt-then-MAC scheme is used.

Encryption algorithm In the RFC 5077, it is recommended to use *AES-128-CBC* as the encryption algorithm. Table 3.2 shows the used encryption algorithms for the evaluated TLS-libraries. In total, five different encryption algorithms are used across all TLS-libraries. BoringSSL is the only library that uses exactly the recommended encryption algorithm with the suggested key size. GnuTLS, MatrixSSL and OpenSSL use also *AES-CBC* but with a larger key (256 bits). BotanSSL, MbedTLS, and S2N use the AES-GCM (Galois/Counter Mode) encryption algorithm. Moreover, MbedTLS supports AES-CCM (Counter with CBC-MAC). RusTLS and GoTLS are the only libraries using *ChaCha20Poly1305* and *AES-128-CTR.*

Authentication algorithm In the following, we will describe the used authentication algorithms (see Table 3.2). The RFC 5077 recommends the use of HMAC-SHA-256. Note that encryption algorithms that have the AEAD property do not have to use an additional authentication algorithm since they offer encryption and authentication in combination. In total, four TLS-libraries use an AEAD encryption

Table 3.1: Ticket format of evaluated TLS-libraries. If a ticket format is not conform with RFC 5077, then we mark the differences with red color.

TLS-library	Session Ticket Format	RFC 5077 conform?
- OpenSSL - GoTLS - MatrixSSL - BoringSSL	key_name[16] iv[16] enc_state<...> mac[32]	✓
- GnuTLS	key_name[16] iv[16] enc_state_length[2] enc_state<...> mac[20]	(✓)
- S2N	key_name[16] iv[16] enc_state<...> mac[16]	(✓)
- MbedTLS	key_name[4] iv[12] enc_state_length[2] enc_state<...> mac[16]	✗
- RustTLS	iv[12] enc_state<...> mac[16]	✗
- BotanSSL	magic_constant[8] key_name[4] key_seed[16] iv[16] enc_state<...> mac[16]	✗

algorithm. Thus, we only present the authentication algorithms for all five libraries that are using a Non-AEAD encryption algorithm. Four of them, do use the recommended HMAC algorithm. Only GnuTLS uses a different authentication algorithm with HMAC-SHA-1.

Encrypt-then-MAC For Non-AEAD ciphers, we evaluate if the Encrypt-then-MAC scheme is used. The RFC 5077 mentions to calculate the MAC over the *key name*, *iv* and *encrypted state* which we interpret as the Encrypt-then-MAC scheme. If an implementation uses the MAC-then-Encrypt or Encrypt-and-MAC scheme, then the implementation may be vulnerable to different attacks as padding ora-

Table 3.2: Encryption and authentication algorithms of evaluated TLS-libraries. Note that encryption algorithms with AEAD property(see column 4), also automatically provide authentication. Thus, they do not need to additionally implement an authentication algorithm.

TLS-library	Encryption Algorithm	Authentication Algorithm	AEAD	EtM[a]
botanSSL	AES-256-GCM		✓	
boringSSL	AES-128-CBC	HMAC-SHA256		✓
GnuTLS	AES-256-CBC	HMAC-SHA1		✓
GoTLS	AES-128-CTR	HMAC-SHA256		✓
matrixSSL	AES-256-CBC	HMAC-SHA256		✓
mbedTLS	AES-GCM (128/256) AES-CCM (128/256)		✓	
openSSL	AES-256-CBC	HMAC-SHA256		✓
RustTLS	ChaCha20Poly1305		✓	
S2N	AES-256-GCM		✓	

[a] EtM: Encrypt-then-MAC

cle attacks [28]. Our evaluation has shown that all five libraries that use Non-AEAD algorithms use the recommended Encrypt-then-MAC scheme (see Table 3.2).

3.3 Key Rotation

In a source code analysis, we evaluate for all nine TLS-libraries if they implement a key rotation mechanism. The RFC 5077 recommends to rotate the STEKs regularly to reduce the impact of a STEK compromise.

In Table 3.3, we present results of our evaluation. In total, six TLS-libraries implement a key rotation mechanism. OpenSSL, MatrixSSL, and BotanSSL do not implement any key rotation mechanism at all. If no key rotation mechanism is implemented, then the STEK is used for the whole server lifetime.

In the following, we look more closely at the implemented key rotations in GoTLS and MbedTLS. After that, we evaluate how OpenSSL allows the implementation of

Table 3.3: Evaluation results of TLS-libraries for key rotation mechanism.

	Implements Key Rotation
botanSSL	✗
boringSSL	✓
GnuTLS	✓
GoTLS	✓
matrixSSL	✗
mbedTLS	✓
openSSL	✗
RustTLS	✓
S2N	✓

a customized key rotation mechanism. We introduce two terms in the context of key rotation:

1. *ticketKeyRotation*: This is the interval until the server rotates its STEK. After the interval passed, the server creates a new STEK and encrypts all newly issued tickets with that key.

2. *ticketKeyLifetime*: This is the interval until the server discards the STEK. Until this interval passes, all session tickets that were issued with the STEK, can be redeemed by the clients.

MbedTLS always stores two STEKs at the same time in a key array (see Figure 3.4). The *ticketKeyRotation* interval is per default set to 24 hours. The *ticketKeyLifetime* interval always is twice as long as the *ticketKeyRotation*. That means that every 24 hours the oldest STEK is discarded and a new STEK is created.

GoTLS offers two possibilities to maintain the STEK: 1. We can manually generate and maintain STEKs. At the server start, we have to input a list of STEKs. The first key in the list is used for encryption and all others can only be used for the decryption of redeemed tickets. However, to rotate the keys, the list has to be manually updated. 2. We use the auto-generated and maintained STEKs in GoTLS. Then, GoTLS generates and rotates automatically the STEKs as we can see in Figure 3.5. Per default the *ticketKeyRotation* is 24 hours and the *ticketKeyLifetime* is 7 days. Thus, the number of keys stored in parallel can grow up to 7.

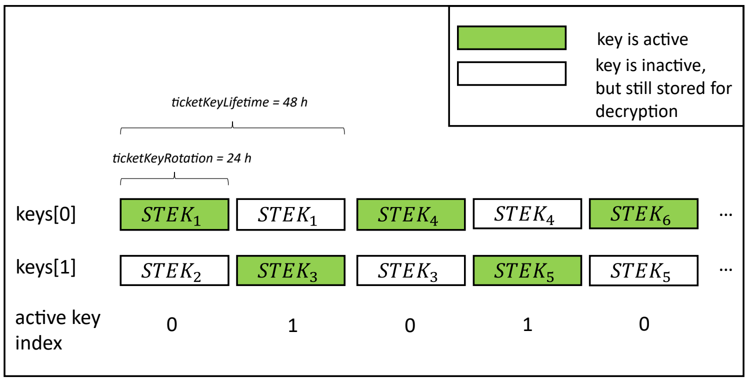


Figure 3.4: Key rotation in MbedTLS

Additionally, we evaluate if the key rotation in MbedTLS and GoTLS is working correctly. Therefore, we set the *ticketKeyRotation* interval to 5 seconds and output the newly generated STEKs for 100 key rotations. As the keys are only rotated when a client connects to the server, we periodically connect to the server with the OpenSSL client. Then, we check if the newly generated STEKs are all different and that no STEK is initialized with zeros. In the test results, we could not find any abnormalities.

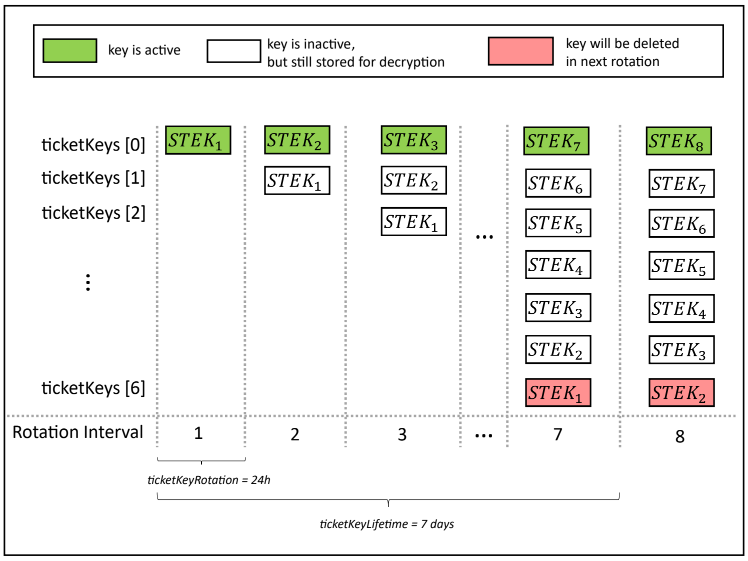


Figure 3.5: Key rotation in GoTLS

Customizing Ticket Handling in OpenSSL As described before, OpenSSL does not implement any key rotation. However, OpenSSL allows customizing the session ticket mechanism with call-back functions[4]. Webservers using OpenSSL can use this feature to implement a key rotation mechanism. We make a source code analysis of the three most three popular webservers Apache, Nginx, and OpenLiteSpeed that use OpenSSL and evaluate if they implement a key rotation. All three webservers offer the option to input a session ticket key file at the server start. This ticket key file must contain 48 random bytes: 16 bytes for the *key name*, 16 bytes for the *encryption key* and 16 bytes for the *HMAC key* (see Listing 3.6).

```
Apache:
key_name[16] || hmac_key [16] || aes_key [16]

OpenLiteSpeed:
key_name[16] || aes_key [16] || hmac_key [16]

Nginx:
key_name[16] || aes_key [16] || hmac_key [16]
```

Listing 3.6: 48-byte session ticket file structure in Apache, OpenLiteSpeed and Nginx

If this option is selected, then all webservers use *AES-128-CBC* to encrypt their session tickets. Nginx does also support an 80 byte session ticket key file. Then, 32 bytes are used to initialize the *encryption key* and 32 bytes for the *HMAC key* (see Listing 3.7). In this case, *AES-256-CBC* is used as the encryption algorithm.

```
Nginx:
key_name[16] || hmac_key [32] || aes_key [32]
```

Listing 3.7: 80-byte session ticket file structure in Nginx

Interestingly, the order inside the session ticket key file is not equal for all three webservers. If a session ticket key file is used, then the keys are not automatically rotated. In Apache, a rotation of the STEK is only possible when the key file is updated and the server is restarted. This can be done by using a cron job. In Nginx, to rotate the key you can reload the server configuration periodically with a cron job. Richard Fussenegger implemented this in his master thesis [21]. According to Tim Taubert, both approaches do not come close to a real solution [27]. In OpenLiteSpeed, it suffices to update the session ticket key file periodically.

If no session ticket key file is provided, then all three webserver implementations automatically generate their own STEK. Apache and Nginx use the in OpenSSL generated STEK. However, they do not implement any key rotation mechanism so that the STEK is used for the whole server lifetime (see Table 3.8). In OpenLiteSpeed, at server start, three STEKs are generated. Moreover, the STEKs are

[4]https://www.openssl.org/docs/man1.0.2/man3/SSL_CTX_set_tlsext_ticket_key_cb.html

Table 3.8: Evaluation results for key rotation mechanisms of popular webservers using OpenSSL

Webserver	Implements Key Rotation
Apache	✗
Nginx	✗
OpenLiteSpeed	✓

rotated automatically. The rotation interval is configurable and is set per default to 60 hours.

3.4 Randomness

In the following section, we will evaluate the randomness generation of the three TLS-libraries OpenSSL, GoTLS, and MbedTLS in more detail. First, we will describe what randomness is used for. In the context of session tickets, we need randomness in two procedures: 1. Initializing a new STEK. This includes a random *key name*. 2. Generating IV for a new session ticket. We evaluate which RNG function is used for every library:

- MbedTLS: At server start, you can configure the RNG function for session tickets. The example server included in MbedTLS uses the *mbedtls_ctr_drbg_-random* (ctr_drbg.c) function as an RNG for session tickets.
- GoTLS: If no explicit RNG function is configured, GoTLS uses the *crypto-/rand*[5] package as an RNG.
- OpenSSL: OpenSSL uses two different RNG functions for session tickets: The RNG function *RAND_priv_bytes_ex*[6] is used for generating the encryption and HMAC key at server start. The RNG function *RAND_bytes_ex* is used for generating the *key name* and the IVs. The difference between both RNG functions is that *RAND_priv_bytes_ex* is intended to be used for values that should remain private.

We made a small test to check if the RNG functions are working correctly.
Therefore, we generate 100000 16-byte random values for each RNG function. Then, we test the generated values for duplicates and the zero vector. In our test results, we can not see any abnormalities.

[5]https://pkg.go.dev/crypto/rand
[6]https://www.openssl.org/docs/manmaster/man3/RAND_priv_bytes_ex.html

3.5 Replay Protection

We only evaluated the replay protection for OpenSSL, because the majority of the other libraries do not support 0-RTT. In OpenSSL, you can enable replay protection for the 0-RTT data in TLS 1.3 with the `SSL_OP_NO_ANTI_REPLAY` flag. If the replay protection is enabled, then the server does not send a stateful session ticket that includes all cryptographic parameters. Instead, they send a stateless ticket that includes the corresponding session-id to the client. This is the concept of session caches as explained in Section 2.1. Thus, the server loses the advantage of using session tickets and has to store all cryptographic parameters of the sessions by itself. None of the proposed replay protection mechanisms in the RFC 8446 are implemented in OpenSSL.

4 Proposal of Possible Security Vulnerabilities

In this chapter, we propose and explain different security vulnerabilities that may exist in the session ticket handling of servers. We will implement test suites for each proposed vulnerability. In the following, we shortly describe our attack model and the differences in the impact between TLS 1.2 and 1.3:

We assume that an attacker has access to the network and can eavesdrop the traffic. Therefore, they can eavesdrop and store the sent session tickets. If an attacker gets in possession of the session secrets stored in the session ticket, they can do the following depending on the selected TLS version:

- TLS 1.2: An attacker can decrypt all sessions where the leaked session ticket was either issued or redeemed. Additionally, an attacker can perform a man-in-the-middle attack and break the authenticity of the server, when the client resumes the session.
- TLS 1.3: An attacker can decrypt the 0-RTT data of all sessions where the leaked session ticket was redeemed. Additionally, an attacker can perform a man-in-the-middle attack and break the authenticity of the server, when the client resumes the session.

Additionally, we assume that the attacker can actively modify messages or sent self-constructed messages by itself to the server.

Next, we describe the different proposed vulnerabilities and their impact:

Repeated Initialization Vectors (IV) An initialization vector is used right along the symmetric key to encrypt data to prevent repetition in the ciphertext. Ideally, for every encryption, a new IV has to be chosen randomly. However, generating randomness is an expensive computational operation for the server, so it can be possible that servers repeat their IVs.

Impact: An attacker might be able to observe block collisions. If the AES-GCM (Galois/Counter mode) cipher is used to encrypt the session ticket, then an attacker could additionally exploit it in two different ways [5]: 1. An attacker could learn the authentication key and forge session tickets. 2. If an IV (called *nonce* in GCM) is used twice with the same key, then an attacker learns the XOR of the two plaintexts by XORing the encrypted states of the received session tickets. It may be possible to learn the session secret included in the session ticket.

Unencrypted Session Tickets We evaluate whether servers issue session tickets with an unencrypted session state so that attackers can gain access to the session secrets.

Impact: A passive attacker can simply extract the session secrets from the session tickets.

Zero Key We evaluate if servers use an all-zero STEK to encrypt their session tickets like in GnuTLS [11]. There are two possible ways how a server can use a zero key:

Zero Encryption Key The server uses an all-zero key to encrypt the session ticket.

Impact: A passive attacker can decrypt all session tickets encrypted with an all-zero key and therefore gain access to the session secrets.

Zero HMAC-key The server uses an all-zero key as its HMAC-algorithm.

Impact: The impact is similar to the No-Mac check vulnerability. An attacker can alter the session ticket and simply append the recalculated MAC.

Ciphersuite Change The RFC 5077 recommends including the selected cipher suite of the *initial* session inside the session ticket. We interpret that as a recommendation to continue the session with the same cipher suite. However, this is not explicitly stated in the RFC. Thus, we evaluate if webservers in the wild allow resuming the session with a different cipher suite than in the *initial* session.

Impact: This is undefined behavior, but no attacks are known.

Version Change The RFC 5077 recommends including the selected TLS version of the *initial* session inside the session ticket. We interpret that as a recommendation to continue the session with the same TLS version. However, this is not explicitly stated in the RFC. We evaluate if servers allow resuming the session with a different TLS version than in the initial session.

Impact: This is undefined behavior, but no attacks are known

No MAC check The MAC check may be skipped by the server or not executed correctly.

Impact: An attacker might be able to forge or alter session tickets to impersonate another user. Depending on the selected encryption algorithm, the reconstruction of the included session secret may be possible.

Padding oracle When using CBC operation mode, the implementation might be vulnerable to a padding oracle attack. Servers which are using the MAC-then-Encrypt scheme might be vulnerable to a padding oracle attack [28]. Additionally, servers which are not validating the MAC may also be vulnerable.

Impact: An attacker may reconstruct the session secret stored inside the session ticket.

Replay attack In TLS 1.3, the client can send early data to the server in the session resumption. An attacker can replay this early data to the server. Therefore, the RFC 8446 recommends that servers should implement a countermeasure against replay attacks. We evaluate if servers in the wild are vulnerable to replay attacks. In TLS 1.2, replay attacks are not possible since early data is not supported. Thus, we only evaluate this test suite for TLS 1.3.

Impact: An attacker may change the server's state by replaying the early data. For example, an attacker may be able to re-execute a valid financial transaction a second time.

5 Implementation

In this chapter, we will outline our implementation for the proposed security vulnerabilities in the TLS-Scanner. First, we will explain the used tools and then we will describe the implementation of the test suites more in detail. Finally, we will explain how we verified that our test suites are implemented correctly.

5.1 Used Tools

In order to implement our test suites and scan different TLS servers in the wild, we use existing tools from the TLS-Attacker project. The project is created and maintained by the Chair for Network and Data Security from the Ruhr-University Bochum and the Research Group Systems Security from the Paderborn University.

5.1.1 TLS-Attacker

TLS-Attacker[1] is a Java based framework that allows us to test the configuration and functionality of TLS libraries which includes TLS servers. The framework allows defining and executing custom TLS protocol flows like TLS handshakes. We can dynamically perform TLS handshakes and adapt our behavior depending on the server's response. We can use this feature to test the server configuration or to perform different kinds of attacks. For our implementation, we use the functionality of TLS-Attacker to receive and process session tickets issued by servers in the initial handshake. We can freely modify the issued session ticket before sending it back to the server and finally analyze the server's behavior. The *TLS-Core* module contains the main functionality of the TLS-Attacker framework.

5.1.2 TLS-Scanner

TLS-Scanner[2] evaluates the TLS server configurations for a specific host. The tool is implemented in Java and is based on the TLS-Attacker framework.

[1]https://github.com/tls-attacker/TLS-Attacker
[2]https://github.com/tls-attacker/TLS-Scanner

The main idea is that TLS-Scanner connects to a TLS server and performs different Probes. There are two different types of probes:

1. Probes that evaluate the server configuration: for example supported TLS versions, supported cipher suites, supported extensions
2. Probes that test if the server is vulnerable to different kinds of attacks: for example Bleichenbacher, Heartbleed, Padding Oracle

Finally, TLS-Scanner generates a report with the probe results of the scanned TLS server. We can easily add new probes to the TLS-Scanner. In our case, we implement a `SessionTicketProbe` to evaluate the server's behavior according to session tickets.

5.2 Implementation of SessionTicketProbe

In this section, we will explain in more detail our implementation of the `SessionTicketProbe` in the TLS-Scanner. The `SessionTicketProbe` evaluates the different test suites for session tickets and outputs the final result to the `SiteReport`.

5.2.1 SessionTicketProbe

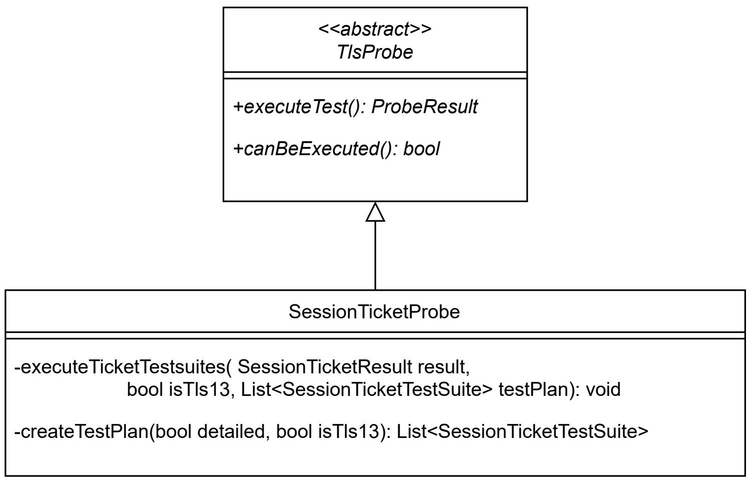

<<abstract>>
TlsProbe
+executeTest(): ProbeResult
+canBeExecuted(): bool
SessionTicketProbe
-executeTicketTestsuites(SessionTicketResult result, bool isTls13, List<SessionTicketTestSuite> testPlan): void
-createTestPlan(bool detailed, bool isTls13): List<SessionTicketTestSuite>

Figure 5.1: Class diagram of SessionTicketProbe

In TLS-Scanner, the `TlsProbe` implements the basic functionality for executing a specific probe against a host. We extend it with a new `SessionTicketProbe` where we execute the test suites for the presented vulnerabilities (see Figure 5.1). The method *canBeExecuted* defines the precondition for executing the current probe. In the case of the `SessionTicketProbe`, we have to wait until the Pre-Probes, evaluating the supported protocol versions and cipher suites, are completed. In the method *executeTest* we execute all test suites for the proposed vulnerabilities. We will describe the execution of the test suites later in detail.

For every presented vulnerability, we implement a test suite that evaluates and outputs if the scanned host is vulnerable. Therefore, we define the abstract class `SessionTicketTestsuite` (see Figure 5.2). Every test suite has to implement the *executeTicketTestsuite* function where the test suite is executed and the results are written to the `SessionTicketResult`. We categorize the test suites into two different types:

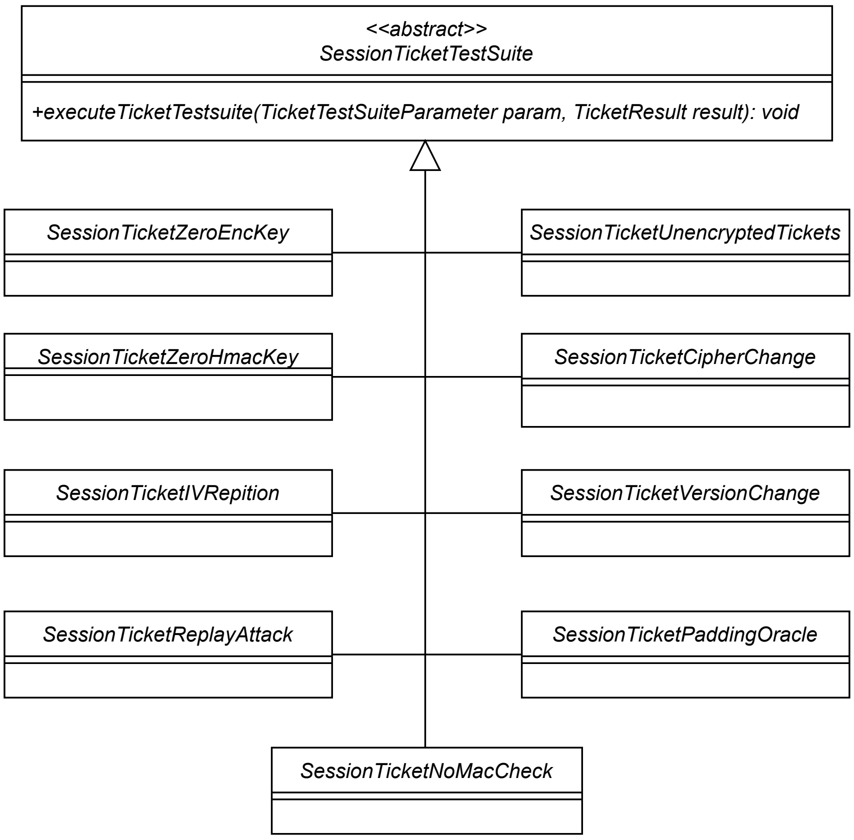

<<abstract>>
SessionTicketTestSuite
+executeTicketTestsuite(TicketTestSuiteParameter param, TicketResult result): void
SessionTicketZeroEncKey
SessionTicketUnencryptedTickets
SessionTicketZeroHmacKey
SessionTicketCipherChange
SessionTicketIVRepition
SessionTicketVersionChange
SessionTicketReplayAttack
SessionTicketPaddingOracle
SessionTicketNoMacCheck

Figure 5.2: class diagram of implemented session ticket test suites

1. *Passive* test suites: *IV Repetition*, *Zero Encryption Key*, *Zero HMAC-key* and *Unencrypted Ticket.* These test suites only evaluate the tickets issued by the server. They do not perform any session resumption. We can compare it to an passive attacker who sits in the network and eavesdrops the traffic.

2. *Active* test suites: *No Mac Check*, *Padding Oracle*, *Ciphersuite Change*, *Replay Attack* and *Version Change.* These test suites actively perform session

resumptions to find a vulnerability at server side.

We execute all test suites for TLS 1.2 and TLS 1.3 except two test suites: The *Replay Attack* test suite can only be executed in TLS 1.3 because TLS 1.2 does not support early data. The *Version Change* test suite evaluates vulnerabilities across different TLS versions and therefore does not belong to any TLS version. We will give a detailed description of the implemented test suites in the upcoming sections.

In TLS-Scanner, every probe outputs a `ProbeResult` which contains the results of the executed probe (see Figure 5.3). We implement a `SessionTicketResult` which stores all results of the executed session test suites. As we are scanning session tickets for TLS 1.2 and 1.3, we have to store the results for each version separately in two `TicketResult` objects *ticketResultTls12* and *ticketResultTls13*. In our thesis, we refrain from a more detailed description of the result implementation.

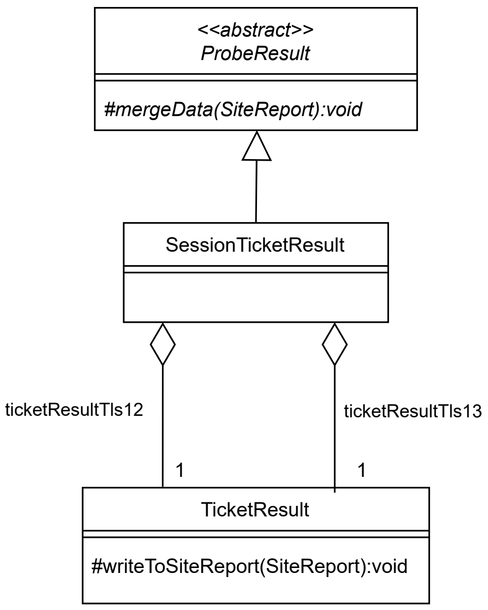


Figure 5.3: class diagram of implemented `SessionTicketResult`

Executing test suites for specific TLS version First, we describe how we evaluate test suites for a specific TLS version (either TLS 1.2 or TLS 1.3) with the *executeTicketTestsuites* function (see Listing 5.4). The function gets three input parameters: 1. The selected TLS version. 2. The `SessionTicketResult` *result*: The object where we output the test suite results. 3. *List<`SessionTicketTestSuite`> testPlan*: a list of test suites which shall be executed. Before executing the test suites, we first have to evaluate if the server supports session tickets for the selected

TLS version. Therefore, we implement the `SessionTicketSupportTest` class which evaluates the following:

```
private void executeTicketTestSuites(SessionTicketResult sessionTicketResult,
boolean isTls13, List<SessionTicketTestSuite> testPlan) {
  try {
    TicketResult ticketResult = sessionTicketResult.getTicketResult(isTls13);
    //test if server issues session ticket and resumes sessions with ticket
    SessionTicketSupportTest sessionTicketSupportTest = new
        SessionTicketSupportTest(handshakeHelper, getParallelExecutor());
    sessionTicketSupportTest.testSessionTicketSupport(isTls13,ticketResult);
    if(ticketResult.getIssuesTickets()!= TestResult.TRUE){
      return;
    }
    //generate all necessary data for test suites
    TicketTestSuiteParameter testSuiteParameter =
    new TicketTestSuiteParameter(isTls13, ticketResult.getKeyNameLength(),
    sessionTicketSupportTest.getStateList(), supportedSuites, supportedVersions);
    //Execute all test suites
    for(SessionTicketTestSuite testSuite :testPlan){
      testSuite.executeTicketTestsuite(testSuiteParameter, sessionTicketResult);
    }
  } catch (Exception e) {
    LOGGER.error("Anaylze session tickets failed");
  }
}
```

Listing 5.4: *executeTicketTestSuites* function in `SessionTicketProbe` executes test suites for selected TLS version

1. ISSUES_TICKET: We perform one full handshake and evaluate if the server issues a session ticket. If yes, we output that the server issues tickets.

2. RESUMES_WITH_TICKET: We perform a session resumption with the issued ticket. If the server accepts the session ticket and performs a resumption handshake, we output that the server allows resumption.

3. *stateList*: As default we execute 10 full handshakes and store all 10 sessions along with the cached session ticket in the *stateList*. Later, all passive test suites can evaluate the issued session tickets and do not have to perform handshakes on their own. We evaluate multiple session tickets because a host can use multiple load balancers. With 10 handshakes, we may hit different load balancers and therefore we can evaluate if one of them is vulnerable to our test suites. We execute the 10 handshakes in parallel to improve the performance of our scan.

4. Additional information: We output some additional information for later anaysis in the `SessionTicketResult`:

   - *keyNameLength*: We set the length of the key name field as following: The number of equal bytes from the starting position for two issued session tickets.

- *ticketList*: We store all issued session tickets for later analysis.

We only evaluate the test suites if the server supports session tickets for the selected TLS version. In the next step, we generate the `TicketTestSuiteParameter` object *testSuiteParameter*. This object contains all necessary data that the test suites need for execution. For example, the passive test suites need the *stateList* to evaluate the session tickets. Finally, we can execute all test suites sequentially. Every test suites stores the results in the *sessionTicketResult* object.

Execute Test In the following, we will describe in more detail how we implement the *executeTest* function in the `SessionTicketProbe`. This is the main function that is called to execute a probe. First, we create a new `SessionTicketResult` object *ticketResult* where we store the results of all test suites (see Listing 5.5). Then, we generate for both TLS versions 1.2 and 1.3 a list of test suites which we want to execute. Therefore, we implement the *createTestPlan* function. The function generates, dependent on the input parameter *PlanDetail*, a list with the following test suites:

```
public ProbeResult executeTest() {
  try {
    SessionTicketResult ticketResult = new SessionTicketResult();
    List<SessionTicketTestSuite> testPlanTls12 = createTestPlan(false/*TLS 1.2*/,
        PlanDetail.HIGH);
    List<SessionTicketTestSuite> testPlanTls13 = createTestPlan(true/*TLS 1.3*/,
        PlanDetail.NORMAL);
    // test tls 1.2 test suites
    if (supportedVersions.contains(ProtocolVersion.TLS12)) {
      executeTicketTestSuites(ticketResult, false, testPlanTls12);
    }
    // test tls 1.3 test suites
    if (supportedVersions.contains(ProtocolVersion.TLS13)) {
      executeTicketTestSuites(ticketResult, true, testPlanTls13); }
    // test version change test suite
    if(ticketResult.getTls12Results().getIssuesTickets()==TestResult.TRUE ||
        ticketResult.getTls13Results().getIssuesTickets()==TestResult.TRUE) {
      SessionTicketVersionChange versionChange = new
          SessionTicketVersionChange(handshakeHelper);
      versionChange.executeTicketTestsuite(new TicketTestSuiteParameter(supportedSuites,
          supportedVersions), ticketResult);
    }
    return ticketResult;
  } catch (Exception E) {
    LOGGER.error("Could not scan for " + getProbeName(), E);
    return new SessionTicketResult(TestResult.ERROR_DURING_TEST);
  }
}
```

Listing 5.5: *executeTest* function in class `SessionTicketProbe`

- *PlanDetail.NORMAL*: *IV Repetition*, *Zero Encryption Key*, *Zero HMAC Key*, *Unencrypted Ticket*, *Ciphersuite Change*, *Replay Attack* (only in TLS 1.3)

- *PlanDetail.HIGH*: all test suites from PlanDetail.NORMAL + *No Mac Check*, *Padding Oracle*

In the PlanDetail.NORMAL mode, all test suites are included that do not perform more than five handshakes at all. In the PlanDetail.HIGH mode, there are additionally the two test suites *No Mac Check* and *Padding Oracle* included. Both test suites perform on average more than 100 handshakes. Therefore, we only evaluate by default all test suites in TLS 1.2 which significantly increase the runtime of our scan. In the next step, we execute the test suites for TLS 1.2 with the previously described function *executeTicketTestSuites* if the server supports TLS 1.2. After that, we execute the test suites for TLS 1.3 if the server supports TLS 1.3. Then, we execute the *Version Change* test suite that does not belong to a specific TLS version. Finally, we return the *ticketResult* which contains all results of the executed test suites.

5.2.2 Handshake Implementation

The implementation and execution of the proposed test suites require the execution of two different handshake types with a server: 1. The initial full handshake, where the server issues a new ticket inside the *NewSessionTicket* message. 2. The resumptions handshake, where the client redeems the issued ticket.

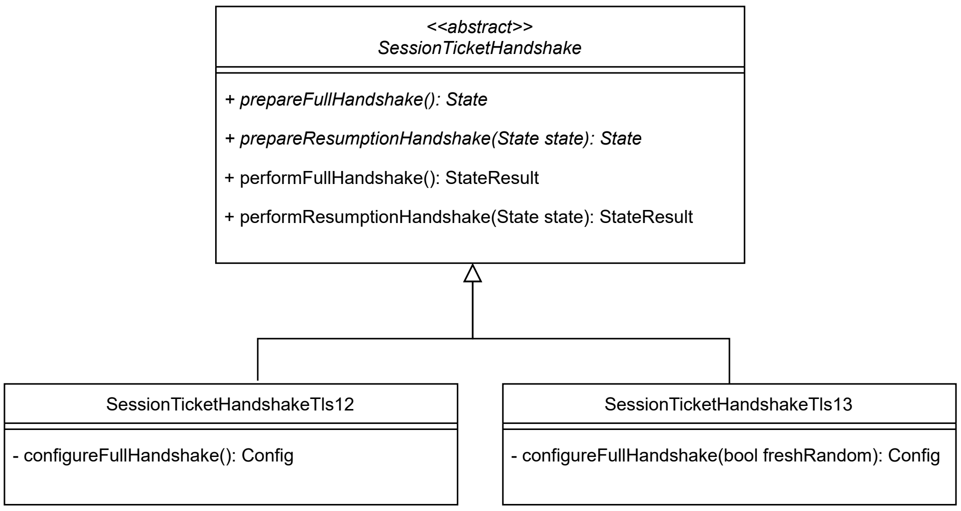

<<abstract>>
SessionTicketHandshake
+ prepareFullHandshake(): State
+ prepareResumptionHandshake(State state): State
+ performFullHandshake(): StateResult
+ performResumptionHandshake(State state): StateResult
SessionTicketHandshakeTls12
- configureFullHandshake(): Config
SessionTicketHandshakeTls13
- configureFullHandshake(bool freshRandom): Config

Figure 5.6: `SessionTicketHandshake` class diagram

We implement the abstract class `SessionTicketHandshake` which offers function that can execute both handshake types (see Figure 5.6). However, the handshakes are different depending on the TLS version. Thus, we implement for both versions a handshake class `SessionTicketHandshakeTls12` and `SessionTicketHandshakeTls13` extending the `SessionTicketHandshake`. For both handshake types, there are two function types: 1. *prepare* functions which only configure the handshake but do not execute the handshake. Test suites that want to parallelize the execution of multiple handshakes use this function type. 2. *perform* functions which automatically execute the handshake.

In the following, we describe how we implement the two handshake types. We do not implement both TLS handshakes from scratch because the TLS-Attacker Core module implements both handshakes. The `SessionTicketHandshake` implements two functions for configuring and executing a full handshake:

- *prepareFullHandshake()*: This function configures a full handshake and outputs a *State* which is ready for execution.
- *performFullHandshake()*: First, the function configures a full handshake with the *prepareFullHandshake()* function. Then, it executes the full handshake and outputs a *StateResult* which contains the resulting *state* and the handshake execution result.

Moreover, the `SessionTicketHandshake` implements two functions for configuring and executing a resumption handshake: We input a *state* that has executed a full handshake and stored the issued session ticket in its cache.

- *prepareResumptionHandshake(State state)*: This function configures and outputs a *resumeState* which is ready for executing a resumption handshake.
- *performResumptionHandshake(State state)*: First, this function calls the *prepareResumptionHandshake* function to configure a *resumeState*. Then, the resumption handshake with the *resumeState* is executed. Finally, the function outputs the *StateResult*.

5.2.3 Session Cache: Evaluating and Manipulating Session Tickets

To evaluate the different presented test suites for session tickets, we need to access the session cache. The class `TlsContext` of the TLS-Attacker Core module implements the session cache. After executing a full handshake for a *state*, we can access all necessary information for session resumption in the `TlsContext` class. The session cache stores different data depending on the selected protocol version.

In TLS 1.2, the session cache consists of the *sessionList* and the *sessionTicketTls* (see Listing 5.7). If we have performed a full handshake, then the current *session* is entered to the *sessionList* with the corresponding *ID* and *mastersecret*. Additionally, the *ticket* value in the received *NewSessionTicket* message is stored in the *sessionTicketTLS*. With this information a session resumption is possible.

```
class TlsContext {
  ...
  List<Session> sessionList;
  byte[] sessionTicketTLS;
  ...
}
  class Session{
    opaque  ID;
    byte[]  mastersecret;
  }
```

Listing 5.7: Session cache in class `TlsContext` for TLS 1.2

In TLS 1.3 the session cache consists of a list *pskSets* (see Listing 5.8). For every received NewSessionTicket message a new *PskSet* is created. The textitPskSet contains the *preSharedKeyIdentity* which is the *ticket*. The *preSharedKey* is the secret used for resuming the session. It is derived from the *mastersecret* of the current session and the *ticket_nonce*. The fields *ticketAgeAdd* and *ticketNonce* are copied from the *NewSessionTicket* message. The *ciphersuite* of the initial session is stored as well. A server may send two *NewSessionTicket* messages in one handshake, but we only store one *PskSet* in this case. If we store both tickets, we would send both tickets in the session resumption. This would sophisticate the test suite results where we redeem manipulated session tickets.

```
class TlsContext {
  ...
  List<PskSet> pskSets;
  ...
}
class PskSet{
  byte[] preSharedKeyIdentity;
  byte[] preSharedKey;
  byte[] ticketAgeAdd;
  byte[] ticketNonce;
  String ticketAge;
  CipherSuite cipherSuite;
}
```

Listing 5.8: Session cache in `TlsContext` for TLS 1.3

In order to access the session ticket of the session cache for both TLS versions, we implement two helper functions in the `SessionTicketUtil` class.

- `getSessionTicket(State state)`: This functions returns the received session ticket of the current *state*. If the selected protocol version is TLS 1.2, then the *sessionTicketTLS* value is returned. If the selected version is TLS 1.3, then the *preSharedKeyIdentity* of the stored *pskSet* (we only store one *pskSet*) is returned.
- `setSessionTicket(State state, byte[] ticket)` This function sets a new *ticket* value as the current session ticket so that in the resumption, the new

ticket value is sent to the server. In TLS 1.2, the *sessionTicketTLS* is assigned with the new *ticket* value. In TLS 1.3, the *preSharedKeyIdentitiy* of the first *PskSet* is assigned with the new *ticket* value.

These two functions allow us to evaluate issued session tickets and to manipulate and resend them to the server.

5.2.4 IV Repetition

The *IV Repetition* test suite evaluates if a server uses duplicated IVs in its session tickets. The test suite works similarly for TLS 1.2 and TLS 1.3 and is implemented in the `SessionTicketIVRepetition` class.

The test suite evaluates the issued session tickets from the *stateList*. Therefore, we first extract the session tickets from all states included in the *stateList* (see Figure 5.9). In the next step, we extract the IVs from the session tickets. The main challenge is that the IV position depends on the customized session ticket format. Our evaluation in chapter 3 has shown, that the majority of TLS implementations follow the recommendation for the IV position. Thus, we decide to extract the IVs from index 16 to 32 according to the recommendation (see Section 2.3). However, we will overlook servers with duplicating IVs that do not conform with the recommendation. Furthermore, we may see false positives if a server uses for example longer key names. After extracting the IVs, we can compare each IV pair (IV_i, IV_j) in the `testIVRepitionForTicketPair` function. If we only see one equal IV pair, then we will output that the server is vulnerable to *IV Repetition*.

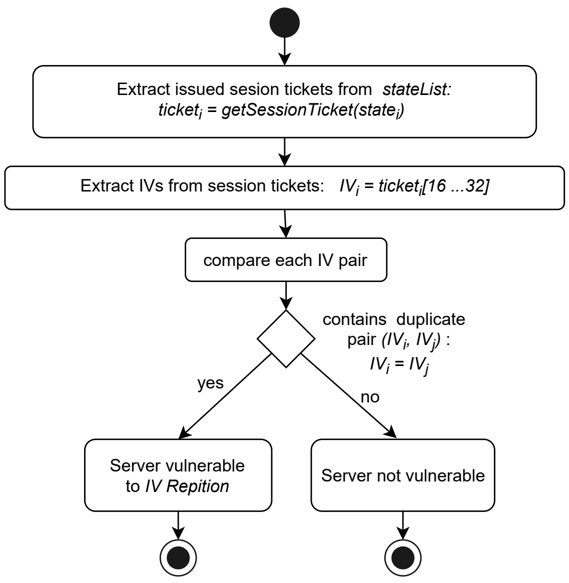

Extract issued sesion tickets from *stateList:*
$ticket_i = getSessionTicket(state_i)$
Extract IVs from session tickets: $IV_i = ticket_i[16 ...32]$
compare each IV pair
contains duplicate pair (IV_i, IV_j) : $IV_i = IV_j$
yes
no
Server vulnerable to *IV Repition*
Server not vulnerable

Figure 5.9: Activity diagram of *IV Repetition* test suite

5.2.5 Ciphersuite Change

The *Ciphersuite Change* test suite evaluates if a server allows resuming sessions with a different cipher suite. We implement it in the `SessionTicketCipherSuiteChange` class. It works analogue for TLS 1.2 and TLS 1.3.

Initially, we perform a full handshake with a default selected cipher suite *cipherA* and get a newly issued session ticket (see Figure 5.10). We expect from an invulnerable server that it stores *cipherA* in the encrypted state of the issued ticket. Then, we pick a different cipher suite *cipherB* from the hosts' *supportedCipherSuites* and set it in the Config via the *setDefaultSelectedCipherSuite* function. After that, we perform the resumption handshake with the newly selected cipher suite *cipherB*. If the server accepts the session ticket and the ServerHello message contains the newly selected cipher suite, we output that the server is vulnerable to the *Ciphersuite Change* test suite. There are two possibilities why a server may accept a session resumption with a different selected cipher suite: 1. The server does not store the cipher suite inside its session ticket. 2. The server does not check if the newly selected cipher suite matches with the cipher suite stored in the session ticket.

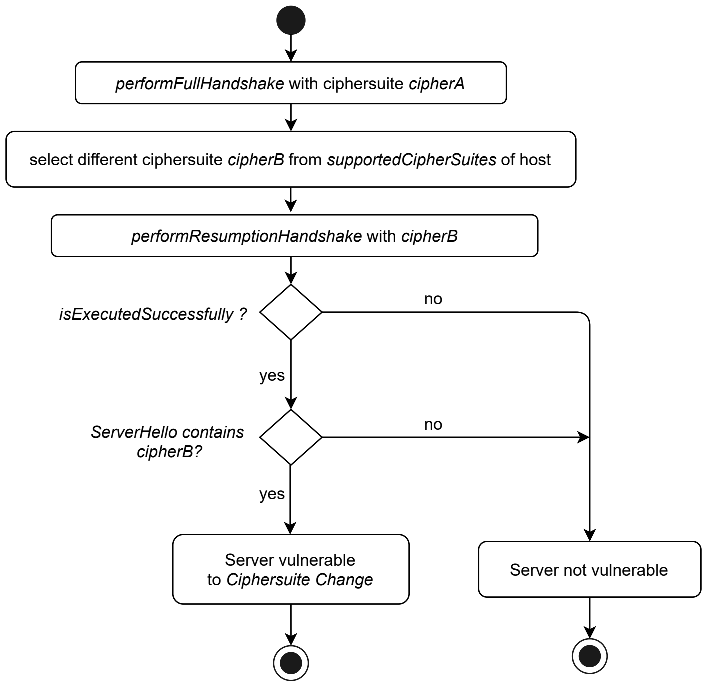

performFullHandshake with ciphersuite cipherA
select different ciphersuite cipherB from supportedCipherSuites of host
performResumptionHandshake with cipherB
isExecutedSuccessfully ?
no
yes
ServerHello contains cipherB?
no
yes
Server vulnerable to Ciphersuite Change
Server not vulnerable

Figure 5.10: Activity diagram of *Ciphersuite Change* test suite

5.2.6 Replay Attack

The test suite *Replay Attack* evaluates if a server is vulnerable to a replay attack. In the real-world scenario, an attacker duplicates the *ClientHello* and *EarlyData* message and replays it to the server. Our test suite performs two consecutive 0-RTT session resumptions with the same *ClientHello* and *EarlyData*. If both EarlyData messages are accepted, we output that the server is vulnerable. We implement the test suite in the *executeTicketTestsuite* function of the `SessionTicketReplayAttack` class. The test suite does only work in TLS 1.3 because TLS 1.2 does not support 0-RTT.

Algorithm 5.11: `Replay Attack test suite`

```
Result: TestResult
1  begin
2      state ← hsHelper.performFullHSWithTicket(TLS13, fixRandom)
3      pskSetListCopy ← state.copyPskList()
4      set fixed time provider
5      if resumeSessionWithEarlyData(state) = false then
6          return NO_EARLY_DATA_SUPPORT
7      stateReplay ← hsHelper.performFullHSWithTicket(TLS13, fixRandom)
       /* Replay early                                              */
8      if resumeSessionWithEarlyData(state, pskSetListCopy) = false then
9          return NOT_REPLAY_VULNERABLE
10     return REPLAY_VULNERABLE
```

Initially, we perform a full TLS 1.3 handshake and cache the received session *ticket* in our *pskSets* (see Algorithm 5.11). With the *freshRandom* parameter, we configure the handshake so that a fixed random value in the *ClientHello* is used. We need this property because we want to perform two session resumptions that are completely identical. Then, we copy the *ticket* with the *pskSet* out of the session cache, because we later need to reuse it in the replay attack. The PskKeyExtension contains additionally to the issued *ticket* the *obfuscatedTicketAge*, which is the ticket age from the client's perspective. The TLS-Attacker sets the *obfuscatedTicketAge* dynamically to the current system time when the *ClientHello* is sent. We prevent this behavior with a *FixedTimeProvider* that sets the *obfuscatedTicketAge* to a fixed value. Then, we can perform the first 0-RTT session resumption with fixed dummy early data for the initial *state*. The resumption is only successful if the server accepts the early data in its Encrypted Extensions with an early data extension. If the 0-RTT resumption is not successful, we output *NoEarlyDataSupport*. If the resumption succeeds, we test the replayed 0-RTT session resumption. For technical reasons, we first need to perform another full handshake for the *stateReplay*. After that, we can finally perform the 0-RTT session resumption for *stateReplay* with the same ticket (included in *pskSetList*) and the same early data. The *ClientHello* and the 0-RTT data are identical to the first resumption. If the server accepts the early data in its Encrypted Extensions, we output that it is vulnerable.

5.2.7 Unencrypted Ticket

The *Unencrypted Ticket* test suites evaluates if a server issues unencrypted session tickets. We can detect a vulnerable server by checking if an issued ticket contains session secrets. We implement the test suite for TLS 1.2 and 1.3 in the *executeTicketTestSuite* function of the `SessionTicketUnencryptedTickets` class (see listing 5.12)

```java
public void executeTicketTestsuite(TicketTestSuiteParameter ticketTestSuiteParameter,
    TicketResult ticketResult) {

  List<State> stateList = ticketTestSuiteParameter.getStateList();
  for (State state : stateList) {
      List<byte[]> secretList = ticketHandshake.generateSecretList(state);
      byte[] ticket = ticketHandshake.getSessionTicket(state);
      for (byte[] secret : secretList) {
          if (secretCheck(ticket, secret)) {
              ticketResult.setContainsPlainSecret(TestResult.TRUE);
              return;
          }
      }
    }
  ticketResult.setContainsPlainSecret(TestResult.FALSE);
}
```

Listing 5.12: *executeTicketTestsuite* function in `SessionTicketUnencryptedTicket`

The test suite evaluates issued session tickets from the *stateList*. We analyze for every state if the issued ticket contains session secrets. Therefore, we have to collect the different session secrets in the secret list with the *generateSecretList* function (see listing 5.13). In TLS 1.2, we add the *preMasterSecret* and the *masterSecret* to the *secretList*, because both are used to derive the session keys [7]. The RFC 5077 recommends to use the *masterSecret* inside the session ticket. In TLS 1.3, we add the *handshakeSecret*, the *masterSecret*, the *resumptionMasterSecret* and the *preSharedKey* to the *secretList*. These are the different values used in the key derivation (see Section 7.1 in [20]). Normally, in TLS 1.3 the *resumptionMasterSecret* is included in the session ticket. Then, we check if the ticket contains one of the secrets from the *secretList* with the *secretCheck* function. We output that the server is vulnerable if one of the issued tickets contains a secret from the *secretList*.

```
public static List<byte[]> generateSecretList(State state) {
  boolean isTls13 = isTls13Selected(state);
  List<byte[]> secretList = new LinkedList<>();
  TlsContext context = state.getTlsContext();
  if (isTls13) {
    secretList.add(context.getHandshakeSecret());
    secretList.add(context.getMasterSecret());
    secretList.add(context.getResumptionMasterSecret());
    secretList.add(context.getPskSets().get(0).getPreSharedKey());
  } else {
    secretList.add(context.getPreMasterSecret());
    secretList.add(context.getMasterSecret());
  }
  return secretList;
}
```

Listing 5.13: *generateSecretList* function in `SessionTicketUtil`

5.2.8 Version Change

The *Version Change* test suite evaluates if a server allows resuming a session in a different TLS version than the ticket was issued. We implement the test suite in the `SessionTicketVersionChange` class. We implement two different types of the test suite because the implementation is technically different depending on the selected TLS versions:

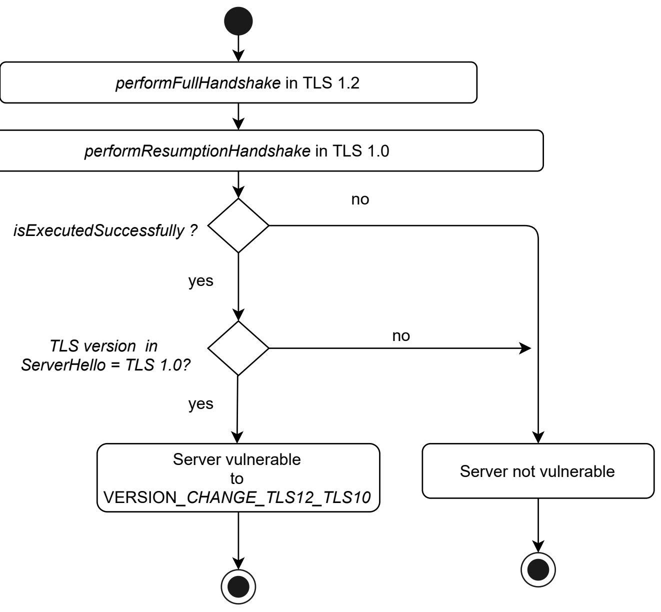

performFullHandshake in TLS 1.2
performResumptionHandshake in TLS 1.0
isExecutedSuccessfully ?
no
yes
TLS version in ServerHello = TLS 1.0?
no
yes
Server vulnerable to VERSION_CHANGE_TLS12_TLS10
Server not vulnerable

Figure 5.14: Activity diagram of *Version Change* test suite, Ticket issued in TLS 1.2 and redeemed in TLS 1.0

Redeem TLS 1.2 session ticket in TLS 1.0/1.1 The session ticket extension works similarly in TLS 1.0/1.1/1.2. Thus, the test suite is for this case not very complex. In the following, we exemplary describe how the version change works from TLS 1.2 to TLS 1.0 (see Figure 5.14). The version change from TLS 1.2 to TLS 1.1 works similarly. First, we perform a full handshake in TLS 1.2 and store the session ticket inside the session cache. In the next step, we resume the session in TLS 1.0 with the cached session ticket. We use the same state for the initial handshake and the session resumption because the session ticket is stored independently of the selected TLS version (except for TLS 1.3). If the server accepts the ticket in TLS 1.0, we output that the server is vulnerable to the version change test suite.

Redeem TLS 1.2 session ticket in TLS 1.3 and vice versa As explained in Section 5.2.2 and 5.2.3, the session ticket resumption mechanism including the session cache works differently for TLS 1.2 and TLS 1.3. Thus, the test suite implementation is more complex than in the first case because we can not reuse the state for the session resumption. In the following, we exemplarily describe how the version change from TLS 1.2 (*versionA*) to TLS 1.3 (*versionB*) works. The version change from TLS 1.3 to TLS 1.2 works analog. First, we perform a full handshake in TLS 1.2 for *stateVersionA* (see Algorithm 5.15). The issued TLS 1.2 session ticket is cached inside the state. We can technically only resume a session in TLS 1.3 for a state if a full handshake has been performed significantly. Therefore, we perform a dummy full handshake in TLS 1.3 for *stateVersionB*. In the next step, we overwrite the session ticket of *stateVersionB* in the session cache with the session ticket issued in TLS 1.2. Finally, we perform the session resumption in TLS 1.3 for *stateVersionB* with the TLS 1.2 session ticket. We expect that servers react in two different ways:

Reject Ticket The server rejects the session ticket and performs a full handshake. Then, we output that the server is not vulnerable.

Accept Ticket If the server accepts the ticket, it will answer with (SH, CCS, FIN) in TLS 1.2 and with (SH, CCS, ENC_EXT, FIN) in TLS 1.3. The challenge is that we do not know which secret the server has selected out of the session ticket. First, we do not know which secret the server has included in the session ticket. For example in TLS 1.3, the server can include either the *resumptionMasterSecret* or the *preSharedKey*. Second, the ticket formats between TLS 1.2 and TLS 1.3 may differ, because the secret sizes can have different lengths. Because we do not know the selected secret of the server, the client can not decrypt the messages after the CCS message. In this case, TLS-Attacker displays that it has received the messages (SSH, CCS, UNKNOWN). Thus, if the *serverAnswer* is (SSH, CCS, UNKNOWN), we output that the server is vulnerable to the Version Change test suite.

Algorithm 5.15: *changeTicketVersionTls12Tls13*

```
Data: versionA, versionB
Result: is server vulnerable to version change
1 begin
2     stateVersionA ← performFullHandshake(versionA) with cipherVersionA
3     stateVersionB ← performFullHandshake(versionB) with cipherVersionB
4     ticketVersionA ← stateVersionA.getSessionTicket()
5     stateVersionB.setSessionTicket(ticketVersionA)
6     serverAnswer ← performResumptionHandshake(stateVersionB, versionB)
       with cipherVersionB
7     if serverAnswer = (SH, CCS, UNKNOWN) then
8         return VULNERABLE
9     return NOT_VULNERABLE
```

5.2.9 No Mac Check

The test suite *No Mac Check* evaluates if a server includes and verifies the MAC of the session ticket. We manipulate each byte of the session ticket once and analyze if the server accepts one of the manipulated tickets. We implement the test suite for TLS 1.2 and 1.3 in the *executeTestSuite* function of the `SessionTicketNoMacCheck` class.

In the beginning, we have to define the different server answers which indicate that the server is not validating the MACs. We expect two different responses from a server that accepts a manipulated ticket:

1. The webserver responds with an *acceptFingerprint* which is either (SH, CCS, FIN) in TLS 1.2 or (SH, CCS, EncExt, FIN) in TLS 1.3. In this case, the client and server have resumed the session with the same session secret.

2. The web server responds with a *differentSecretFingerprint* (SH, CCS, UNKNOWN). In this case, the client and server have resumed the session with a different session secret. This may happen when we modify the ciphertext block that is located at the session secrets position. The server will decrypt the manipulated ticket and resume the session with the modified secret. The server will answer with the same fingerprint as in case 1, but we are not able to decrypt the messages after the CCS message. Therefore, we can only see that an UNKNOWN message type has arrived.

Initially, we perform an initial full handshake and get the issued *ticket* from the session cache. For every index *i* of the session ticket, we do the following: We create a copy *modifiedTicket* of the original *ticket*. Next, we modify the *modifiedTicket* at index *i*: *modifiedTicket[i]* := *modifiedTicket[i]* $\oplus$ 0x01. After creating modifications

Table 5.16: Evaluated authentication algorithms

Algorithm	**MAC size**	**Key size**
HMAC-MD5	16	16
HMAC-SHA1	20	20
HMAC-SHA256	32	32
HMAC-SHA384	48	48
HMAC-SHA512	64	64

for all possibles all indexes i, we perform a session resumption with every *modifiedTicket*. To improve the performance, we execute all resumptions in parallel with the *parallelExecutor*. In total, we have to perform for every byte of the session ticket one session resumption. The ticket sizes of webservers are roughly between 100 and 250 bytes. Finally, we have to evaluate the server's responses and check if the server accepted one of the modified tickets. Therefore, we compare all fingerprint responses with the two fingerprints *acceptFingerprint* and *differentSecretFingerprint*. If one fingerprint response matches with one of the two fingerprints, then we output that the server is vulnerable.

5.2.10 Zero HMAC Key

We evaluate in the test suite *Zero HMAC Key* if a server uses an all-zero HMAC key to calculate the MAC of the session ticket. We implement the test suite in the `SessionTicketZeroHmacKey` for both TLS versions TLS 1.2 and 1.3.

We evaluate for every *state* included in the *stateList* if the state's cached session *ticket* is protected with an all-zero HMAC key. The RFC 5077 recommends to use the *HMAC-SHA-256* authentication algorithm. However, TLS-implementations can choose a different authentication algorithm. Thus, we test the zero key for several popular authentication algorithms (see Table 5.16).

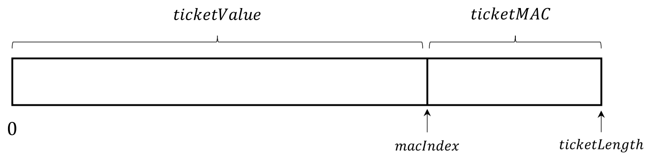


Figure 5.17: Session ticket structure with appended MAC

For every *ticket*, we recalculate the MAC with all authentication algorithms. The recalculation for an authentication algorithm *hmacAlgo* works as following: First, we have to calculate the *macIndex* which is the possible starting index of the MAC inside the session ticket (see Figure 5.17):

$$\mathit{macIndex} = \mathit{ticketLength} - \mathit{hmacAlgo.MacSize}$$

Then, we extract the MAC *ticketMAC* of the current ticket. Next, we generate an all-zero key of size *hmacAlgo.keySize*. At last, we recalculate the MAC with the *zeroKey* for the *ticketValue* which does not contain the appended *ticketMAC*:

$$\mathit{zeroKey} = \mathit{initArray}(\mathit{hmacAlgo.KeySize}, 0)$$
$$\mathit{zeroKeyMAC} = \mathit{hmacAlgo.HMAC}_{\mathit{zeroKey}}(\mathit{ticketValue})$$

If the recalculated *zeroKeyMAC* is equal to the *ticketMAC*, then we know that the server has used an all-zero HMAC key. Thus, we output that the server is vulnerable to our test suite.

5.2.11 Zero Encryption Key

The test suite *Zero Encryption Key* evaluates if a web server uses an all-zero key to encrypt its session tickets. For this, we test different encryption algorithms and ticket formats. We implement the test suite for TLS 1.2 and 1.3 in the `SessionTicketZeroEncryptionKey` class.

The *testEmptyKeyTls* function implements the test suite for a *stateList* (see Algorithm 5.18). Each *state* of the list has performed a full handshake and stored the issued session ticket inside its session cache. We want to evaluate for every state if the issued session ticket is encrypted with an all-zero key. The main challenge is that we do not know the server's ticket format. To decrypt the session ticket correctly, we need to guess the position of the *iv* and the *encrypted state* correctly. Thus, we use the *generateFormat* function to generate a list of possible ticket formats for the current ticket. We describe in the next Section in more detail how this format generation works. Another aspect is that we need to verify if we have successfully decrypted a session ticket: Therefore, we use the function *generateSessionSecrets* function to generate a list with the current session secrets (see Section 5.2.7). If the decrypted ticket contains one of these secrets, we know that the decryption was successful. In the next step, we try to decrypt the session ticket for all generated *formats* with an all-zero key with the *emptyKeyDecryption* function. As we do not know which encryption algorithm the server uses, we test different algorithms which we have observed in the code analysis. We will describe the algorithms and the decryption process in detail in the upcoming section 5.2.11.2. If the decryption is successful for one ticket format, we output that the server is vulnerable.

Algorithm 5.18: *testEmptyKeyTls*

```
Data: stateList
Result: is server vulnerable to Zero Encryption Key test suite
1  begin
2      for ( state: stateList ) {
3          ticket ← hsHelper.getSessionTicket(state)
4          secretList ← hsHelper.generateSecretList(state)
5          formatList ← generateFormats(ticket, NORMAL,keyNameLength)
6          emptyKeyDecryptor ← new SessionTicketFormatDecryptor()
7          for ( format: formatList ) {
8              if emptyKeyDecryptor.emptyKeyDecryption(format) then
9                  return VULNERABLE
10     return NOT_VULNERABLE
```

5.2.11.1 Ticket Format Generation

We implement the ticket format generation in the *generateTicketFormat* function of the `SessionTicketFormat` class. As we do not know the session ticket format, we generate different possible ticket formats for an issued session ticket. We need to consider that the number of decryptions per session ticket significantly influences the performance of our scan. Figure 5.19 shows the main idea of our format generation

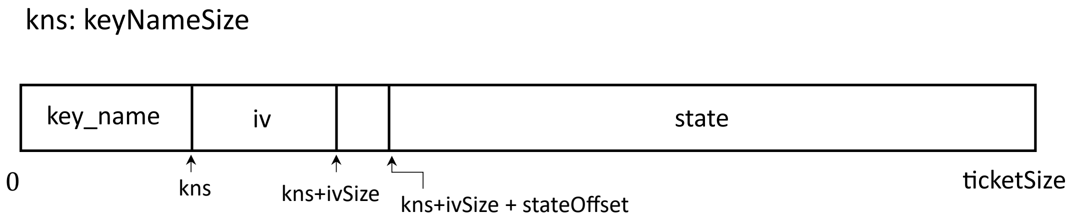


Figure 5.19: Different fields of session ticket for generating ticket formats

algorithm. The ticket consists of four different fields: *key_name*, *iv*, *stateOffset*, *state*. We assume that the order of these fields is fixed, but we allow that fields are missing. We do not need to locate the MAC field which is typically located at the end, because it does not interfere with the decryption process. The sizes of the four different fields are customizable and can be defined by these three parameters: *keyNameSize*, *ivSize*, *stateOffsetSize*. The *keyNameSize* and *stateOffsetSize* can be arbitrary chosen by the webserver. The *ivSize* depends on the the selected encryption algorithm. We assume that $ivSizes = \{0, 8, 12, 16\}$. These are the different IV sizes of the encryption algorithms which we use later for decryption (see next Section). The zero *ivSize* means that we try to decrypt the session ticket with an all-zero IV. Depending on the required resource power, we define three different ticket format generations modes (see Listing 5.20). Important to mention is that in all three modes, the *keyNameSizes* are extended with the measured key name size of the scanned host.

```
FORMAT_DETAIL.LOW:
    keyNameSizes = {0, 4, 16} ∪ {measuredKeyNameSize}
    ivSizes = {0, 8, 12, 16}
    stateOffsets = {0, 2}

FORMAT_DETAIL.NORMAL:
    keyNameSizes = {0, 1, ....,32} ∪ {measuredKeyNameSize}
    ivSizes = {0, 8, 12, 16}
    stateOffsets = {0,2}

FORMAT_DETAIL.HIGH:
    keyNameSizes = {0, 1, ...., ticket.length - 32} ∪ {measuredKeyNameSize}
    ivSizes = {0, 8, 12, 16}
    stateOffsets = {0, 1, .., 16}
```

Listing 5.20: Different ticket format modes for format generation

The ticket generation function calculates all possible combinations of the three parameters. For every combination, we extract the *iv* and *state* from the session ticket and intialize a `SessionTicketFormat` object (see Listing 5.21). Later, for every of these format objects we try to decrypt it with different encryption algorithms. For a typical session ticket of 200 bytes, the different modes will ouput the following number of ticket formats: LOW: 24. NORMAL: 250. HIGH: 25000.

```
public class SessionTicketFormat{
    private byte[] iv;
    private byte[] state;
...
}
```

Listing 5.21: Implementation of `SessionTicketFormat` class

5.2.11.2 Ticket Decryption

We implement the ticket decryption for one specific `SessionTicketFormat` in the *emptyKeyDecryption* function in the `SessionTicketFormatDecryptor` class. As we do not know which encryption algorithm the server uses, we decrypt the ticket format for different encryption algorithms (see Table 5.22). The selection is mostly based on the source code analysis of several TLS-libraries (see Section 3.2).

Table 5.22: Implemented encryption algorithms for *Zero Encryption Key* test suite

Encryption Algorithm	Key size	IV/nonce size	Block size	AEAD cipher
AES-CBC	16, 32	16	16	
DES-CBC	8	8	8	
3DES-CBC	24	8	8	
AES-CTR	16, 32	16	-	
AES-GCM	16, 32	12*	16	✓
AES-CCM	16, 32	12*	16	✓
CHACHA20-POLY1305	32	12	-	✓

* The nonce size of GCM/CCM is not fixed [22, 29]. The default size is 12 bytes and used in many implementations. Thus, we only decrypt with 12-byte IVs.

One specific `SessionTicketFormat` consists of an *iv* and an encrypted *state*. We need to consider a few different things before decrypting a *state* for a specfic encryption algorithm *algo*:

- Padding: If the size of the encrypted *state* is not a multiple of the algorithm's block length, we simply pad the *state* with zeros.
- Zero-IV: If the IV is empty, we use an all-zero array of size *algo.getIVSize()* as the decryption IV.
- Invalid IV-size: If the IV size does not match with *algo.getIVSize()*, we skip the decryption for the encryption algorithm *algo*.
- key_{zero}: We create an all-zero array of size *algo.getKeySize()*. We use it as the decryption key. If an algorithm supports multiple key sizes, then we will test the key_{zero} for each key size.

In the next step, we can decrypt the encrypted *state* with the *iv* and key_{zero}. We distinguish the decryption process between AEAD and Non-AEAD algorithms. The decryption with the a Non-AEAD algorithm *algo* is straightforward: We decrypt the encrypted *state* with the selected *iv* and key_{zero}.

$$state_{dec} = Dec_{algo}(state, iv, key_{zero})$$

The decryption with an AEAD algorithm is more complicated. In contrast to Non-AEAD algorithms, AEAD algorithms provide additional integrity protection. However, we only need to decrypt the encrypted *state*. Verifying the integrity is unnecessary in our case. Moreover, we would have to guess where the Associated Data (AD) is located. Every AEAD-algorithm uses internally a Non-AEAD algorithm for en-/decryption. Thus, we only decrypt with the internally used encryption algorithm. In the following, we show how the decryption works for all three AEAD-algorithms in more detail:

- AES-GCM: GCM uses internally the AES-Counter Mode stream cipher. Normally, AES-GCM expects a 12-byte IV per default and the AES-CTR mode expects a 16-byte IV. We simply append the initial $ctr_{initial} = 0002$ to the iv_{gcm}. To get the exact value of the $ctr_{initial}$, we debugged the GoTLS and mbedTLS library and extracted the value. The following equation outlines the decrytion with the AES-CTR mode:

$$\begin{aligned} iv_{ctr} &= iv_{gcm}||ctr_{initial} = iv_{gcm}||0002 \qquad , |iv_{gcm}| = 12 \\ \Rightarrow state_{dec} &= Dec_{AES\text{-}CTR}(state, iv_{ctr}, key_{zero}) \end{aligned}$$

- AES-CCM: CCM uses also internally the AES-Counter Mode stream cipher. AES-CCM expects a 12-byte IV per default and the AES-CTR mode expects a 16-byte IV. We construct the 16 byte iv_{ctr} from the iv_{ccm} as following [29]:

$$\begin{aligned} iv_{ctr} &= Q||iv_{ccm}||001 = 2||iv_{ccm}||001, \qquad Q = 15 - |iv_{ccm}| - 1 = 2 \\ \Rightarrow state_{dec} &= Dec_{AES\text{-}CTR}(state, iv_{ctr}, key_{zero}) \end{aligned}$$

- CHACHA20-POLY1305: CHACHA20-POLY1305 uses internally the CHACHA20 stream cipher. We select the same 12-byte IV in CHACHA20 as in CHACHA20-POLY1305. Additionally, CHACHA20 expects a counter input. We evaluated the Bouncy Castle implementation[3] to find the correct counter value for the decryption:

$$\Rightarrow state_{dec} = Dec_{ChaCha20}(state, iv, ctr = 1, key_{zero})$$

Finally, we check if the decrypted state $state_{dec}$ contains a session secret from the *secretList*. If yes, we output that the server is vulnerable.

5.2.12 Padding Oracle

The *Padding Oracle* test suite evaluates if a server is vulnerable to padding oracle attacks in the context of session tickets. It is implemented in the `SessionTicket-PaddingOracle` class. At the beginning, we will explain how padding oracles are detected in TLS-Scanner and why we can not reuse the implementation. Then, we will describe the main concepts of our padding oracle test suite.

[3]https://www.bouncycastle.org/

Detecting padding oracles in TLS with TLS-Scanner In the past, security researchers have found padding oracle vulnerabilities in several TLS implementations [4, 16]. These classical vulnerabilities can be used to recover application data encrypted with a CBC cipher suite. TLS-Scanner also implements a test suite that detects TLS padding oracle vulnerabilities. It works as following: The client establishes a TLS connection with a server using a CBC cipher suite. Then, the client constructs and encrypts its own malformed records and sends them to the server. If the server delivers different responses depending on the validity of the padding, we can assume that the server is vulnerable. However, we can not reuse the implementation to detect padding oracles in the session ticket context. We will explain the reasons for that in the following:

- We do not know the symmetric key of the session ticket. Therefore, we can not construct and encrypt malformed records in the same way.
- We do not know the format of the session ticket. We can only guess where the last block that includes the padding bytes is located. The location of the last block is important for us, because that block contains the padding bytes.
- We do not know the authenticated encryption algorithm the server uses to encrypt the session tickets. Even if a server uses a CBC cipher suite, we do not know if it uses Encrypt-then-MAC or MAC-then-Encrypt. We do not have the opportunity to evaluate servers more detailed if they use a CBC cipher suite.
- We do not know which padding scheme is used.

We conclude that, in the context of session tickets, we have fewer capabilities to detect padding oracles.

Detecting padding oracles for session tickets In the following, we present our approach to detect padding oracles for session tickets. One challenge is that we do not know where the last ciphertext block including padding bytes is located, therefore we test 6 different positions. At each of these positions, we try to create a valid 1-Byte padding by resending modified session tickets and evaluating the server's responses. We can detect a valid padding, if we get different responses from the server. If the 1-Byte padding was successful, then we try to create a valid 2-Byte padding at the same ciphertext block. If the creation of the 2-Byte padding is successful, we assume that the server is vulnerable. Otherwise, we output that we created a possible 1-Byte padding. Every time we find a valid padding, we resend the same modifications several times to the server and evaluate the responses with a statistical test. In this way, we try to exclude the case that the server only accidentally responded differently. In the following, we will describe the different steps in more detail:

Locating padding bytes In order to detect padding oracles, we need to know where the last ciphertext block containing the padding bytes is located in the session ticket. We explain for the different operation modes of CBC where the last block is located.

1. MAC-then-Encrypt: If the server uses the MAC-then-Encrypt scheme, it may be vulnerable to padding oracle attacks [28]. Figure 5.23 shows how the session ticket will look like for this scheme. First, the MAC is calculated and appended to the state. Then, the padding is appended and the ciphertext is finally encrypted. The padding bytes are located at end of the last ciphertext block which is at the end of the ticket. If the server does not append any MAC at all, then the padding bytes will be located at the end of the session ticket as well.

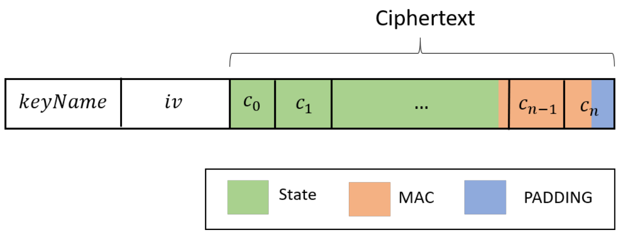


Figure 5.23: Session ticket structure when MAC-then-Encrypt is used

2. Encrypt-then-MAC: If the server implements the Encrypt-then-MAC scheme correctly, it is not vulnerable to padding oracle attacks. However, a wrongly implemented server may perform an incomplete MAC verification or fully skip the verification. In this case, the server may be vulnerable to padding oracles as well. Figure 5.24 shows how a session ticket will look like in this case. The padding bytes are located at the end of the last ciphertext block directly in front of the MAC. However, we do not know how large the MAC is, because the used MAC algorithm is hidden to us. Therefore, we evaluate different MAC sizes $MacSizes = \{16, 20, 32, 48, 64\}$.

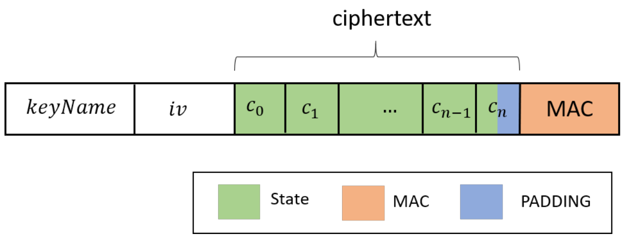


Figure 5.24: Session ticket structure when Encrypt-then-MAC is used

3. Encrypt-and-MAC: Depending on the implementation, servers using the Encrypt-and-MAC approach may also be vulnerable to padding oracle attacks. The

session ticket structure including the position of the last block is identical compared to the Encrypt-then-MAC approach (see Figure 5.24).

From the three different scenarios, we combine all six possibilities where the last block may be located:

$$\mathsf{LastPaddingByteOffsets} = \{0, 16, 20, 32, 48, 64\}$$

We can then calculate for a specific session ticket the possible indexes for the last padding byte:

$$\begin{aligned} \mathsf{lastPaddingByteIndex} &= \mathit{ticket.length} - \mathsf{lastPaddingByteOffset} \\ \mathsf{lastPaddingByteOffset} &\in \mathsf{LastPaddingByteOffsets} \end{aligned}$$

Padding schemes Another important aspect is that we need to know which padding scheme is used. We assume that the server uses the PKCS#7 padding [12]. We also include the same padding scheme, but starting with the 1-Byte padding at "00" instead of "01". We refer in the following to the PKCS#7 padding starting with "00" as *Pkcs00* and to the PKCS#7 padding starting with "01" as *Pkcs01*. The maximum size of the padding depends on the block size of the encryption algorithm. In our case, we assume that the block size is fixed to 16 bytes. Typically, most popular CBC cipher suites use AES which has a block size of 16. The last padding byte can then consists of 16 different values for each scheme:

$$\begin{aligned} \mathit{paddingValues}_{Pkcs00} &= \{\mathtt{00}, \mathtt{01}, \mathtt{02}, \mathtt{03}, \mathtt{04}, \mathtt{05}, \mathtt{06}, \mathtt{07}, \mathtt{08}, \mathtt{09}, \mathtt{0a}, \mathtt{0b}, \mathtt{0c}, \mathtt{0d}, \mathtt{0e}, \mathtt{0f}\} \\ \mathit{paddingValues}_{Pkcs01} &= \{\mathtt{01}, \mathtt{02}, \mathtt{03}, \mathtt{04}, \mathtt{05}, \mathtt{06}, \mathtt{07}, \mathtt{08}, \mathtt{09}, \mathtt{0a}, \mathtt{0b}, \mathtt{0c}, \mathtt{0d}, \mathtt{0e}, \mathtt{0f}, \mathtt{10}\} \end{aligned}$$

Create 1-Byte padding In this section, we describe our approach to create a 1-Byte padding oracle at a vulnerable server. For simplification, we assume that the server uses the *Pkcs01* padding. Later, we will explain the procedure for *Pkcs00* as well. Additionally, we assume that we know where the last two blocks of the ciphertext are located. The main idea is that we modify the session ticket at a fixed position with different values and try to resume the session with the modified tickets. Then, we analyze if the server responds differently to these modified session tickets so that we are able to create a padding oracle.

For creating a valid 1-Byte padding, we have to consider the decryption of the last block P_n as we can see in Figure 5.25. The last Plaintext block P_n always contains at least one padding byte. That means that p_{16} contains one of the 16 possible padding

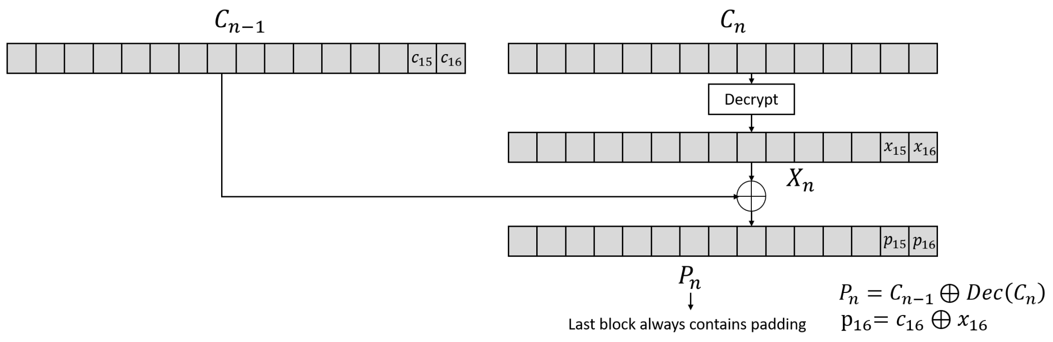

C_{n-1}
C_n
c_{15} c_{16}
Decrypt
x_{15} x_{16}
X_n
p_{15} p_{16}
P_n
Last block always contains padding
$P_n = C_{n-1} \oplus Dec(C_n)$
$p_{16} = c_{16} \oplus x_{16}$

Figure 5.25: CBC decryption of last two blocks

bytes from $paddingValues_{Pkcs01}$. Our goal is to create a valid 1-Byte padding at p'_{16}. Therefore, we have to evaluate which values of $z \in Z_{0x01}$ are needed to create the 1-Byte padding:

$$\begin{aligned} p'_{16} = p_{16} \oplus z = 0x01, \qquad & p_{16} \in paddingValues_{Pkcs01} = \{01, ..., 10\} \\ & z \in Z_{0x01} = ? \end{aligned}$$

If we use the fact that $p_{16} \in paddingValues_{Pkcs01}$, we can calculate all possible values for $z \in Z_{0x01}$ so that $p'_{16} = 0x01$ (see detailed calculation in Section A.1 Table A.1):

$$Z_{0x01} = \{00, 02, 03, 04, 05, 06, 07, 08, 09, 0a, 0b, 0c, 0d, 0e, 0f, 11\}$$

There is one edge case we need to consider. How do we detect if the original padding value p_{16} is a 1-Byte padding. In this case, we do not need to modify the session ticket at all at c_{16} (see value 0x00 in Z_{0x01}). However, if we send an unmodified session ticket to the server, it will of course accept it. Therefore, we simply change the previous ciphertext value $c'_{15} = c_{15} \oplus \mathit{0xff}$ so that we make sure that the MAC check will fail. To simplify the whole process also for the other padding scheme later, we modify c_{15} in the same way for all values of z. In the next step, we have to modify the last ciphertext byte of the second last block C_{i-1} for all calculated possibilities of z.

$$c'_{16} = c_{16} \oplus z, \qquad\qquad z \in Z_{0x01}$$

That means that we modify in total 16 session ticket with all 16 possibilities of z and resend the ticket to the server. A vulnerable server will first decrypt the session ticket. In one case $z = z_{validPadding}$, p'_{16} will be a correct 1-Byte padding and in the other 15 cases the padding will be invalid. A vulnerable server will respond for one session ticket with the valid padding $z = z_{validPadding}$ differently. [4]

If the padding is invalid, we assume that the server responds uniformly. There is one exception: If the MAC is not validated at all, it may also be possible that the server responds with two different response types to an invalid padding: 1. We preserve the session ticket structure after our modification. 2. We destroy the session ticket structure. For both cases, the server may respond differently. For simplicity, we assume that the server answers uniformly for an invalid padding. However, we store the number of different responses for every host and can later perform additional evaluations for servers which responded in three different ways.

We can reconstruct the original p_{16} with our created padding oracle :

$$p_{16} = z_{validPadding} \oplus 0x01$$

In the following, we consider the case that the server uses the *Pkcs00* scheme. The procedure is nearly identical only the modification values for Z_{0x00} slightly differ. We want to find a $z \in Z_{0x00}$ which creates a valid 1-Byte padding $p'_{16} = 0x00$:

$$p'_{16} = p_{16} \oplus z = 0x00, \qquad p_{16} \in paddingValues_{Pkcs00} = \{00, ..., 0f\}$$
$$z \in Z_{0x00} = ?$$

Analog to Z_{0x01}, we calculate all possible $z \in Z_{0x00}$ for every possible padding byte value at p_{16} which may create a valid 1-Byte padding (see detailed calculation in Section A.1 Table A.2).

$$Z_{0x00} = \{00, 01, 02 , 03 , 04, 05, 06, 07, 08, 09, 0a, 0b, 0c, 0d ,0e ,0f\}$$

Now, we combine both padding schemes into one algorithm:

[4]Note that we found out after all evaluations were completed that we overlooked an edge case in our padding oracle implementation. We overlooked an edge case that may appear if the session ticket contains originally a 1-Byte padding. It may happen that we create two valid paddings in rarely cases. However, this edge case did not occur in our validation of our test suite where we tested a vulnerable server with all possible paddings inclusive the a 1-Byte padding. It is very unlikely, that we overlooked vulnerable servers since of this edge case.

1. We modify the session ticket with all possible z at $c'_{16} = c_{16} \oplus z, z \in Z_1$.

   $$\begin{aligned} Z_1 &= Z_{0x00} \cup Z_{0x01} \\ &= \{00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0a, 0b, 0c, 0d, 0e, 0f, 11\} \end{aligned}$$

   Additionally, we always modify $c'_{15} = c_{15} \oplus$ 0xff. Then, we send 17 modified session tickets to the server and evaluate the responses.

2. If the server responds for one $z = z_{validPadding}$ differently than from the other 16 modifications, we assume that we successfully hit a 1-Byte padding. Next, we want to reconstruct the original p_{16} value. However, we do not know the padding scheme of the session ticket. Therefore, we present two different options for p_{16} depending on the used padding scheme:

   - *Pkcs00* $\Rightarrow p_{16} = z_{validPadding}$
   - *Pkcs01* $\Rightarrow p_{16} = z_{validPadding} \oplus 0x01$

Create 2-Byte padding After creating a valid 1-Byte padding with the session ticket, we continue to create a valid 2-Byte padding to confirm that the server is indeed padding oracle vulnerable. In the 2-Byte padding case, we again have to distinguish between the two possible padding schemes, because we still do not know the padding scheme. Therefore, we try to create for both padding schemes a valid 2-Byte padding in a similar way as for the 1-Byte paddings. If the 2-Byte padding succeeds in one case, we assume that the server is vulnerable to padding oracle attacks.

First, we assume that the server uses the *Pkcs01* scheme. In this case, a valid 2-Byte padding is "02 02" so that $p''_{15} = p''_{16} = 0x02$. Additionally, we assume from the first step that $p_{16} = z_{validPadding} \oplus 0x01$. As we know the value of p_{16}, we construct c''_{16} so that the modified plaintext p''_{16} results in "02":

$$\begin{aligned} c''_{16} &= c_{16} \oplus 0x03 \\ \Rightarrow p''_{16} &= p_{16} \oplus 0x03 = 0x02, \end{aligned}$$

Next, we construct c''_{15} so that we create the 2-byte padding at the second last byte $p''_{15} = 02$. Here, we distinguish between two different cases depending on the value of p_{16}.

- $p_{16} \neq 0x01$: In this case, we assume that p_{15} is also a padding byte. If we know p_{16}, then we also know p_{15}. Thus, normally we do not need to test again different possible values for z to find p_{15}. However, we use the same procedure as used in the 1-Byte padding to find the valid 2-Byte padding since we want to know if the server answers again in one case differently. We use different values $z \in Z_2$ to construct c''_{15} so that we create a valid 2-Byte padding at $p''_{15} = 0x02$:

$$\rightarrow p''_{15} = p_{15} \oplus z = 0x01, \quad p_{15} \in paddingValues_{Pkcs01} \setminus \{01\} = \{02, ..., 10\}$$
$$z \in Z_2 = ?$$

Then, we calculate the set Z_2 as following (see detailed calculation in A.1 table A.3):

$$Z_2 = \{\text{00, 01, 04, 05, 06, 07, 08, 09, 0a, 0b, 0c, 0d, 0e, 0f, 12}\}$$

Then we modify c''_{15} for all 15 values of z and the 15 modified session tickets to the server and evaluate the server's response. If the server responds for one $z = z_{validPadding2}$ differently than for the 14 others, we assume that we have successfully created a valid 2-Byte padding. We reconstruct the original plaintext value $p_{15} = z_{validPadding2} \oplus 0x02$. We can confirm that the server is vulnerable by comparing the padding guess for the last byte p_{16} with our current guess for the second last byte p_{15}. If both are equal, we can assume that the server is indeed vulnerable.

- $p_{16} = 0x01$: In this case, we assume that the original session ticket contains a valid 1-Byte padding. We assume that p_{15} is not a padding byte and can contain any possible byte value $b \in \{0, 1, ..., 255\}$. Therefore, we have test all possible byte values $Z_2 = \{00,01,...,ff\}$ for c''_{15} to create a valid 2-Byte padding at p''_{15}:

$$\rightarrow p''_{15} = p_{15} \oplus z = 0x02, \qquad p_{15} \in \{\texttt{00}, \texttt{01}, ..., \texttt{ff}\}$$
$$z \in Z_2 = \{\texttt{00}, \texttt{01}, ..., \texttt{ff}\}$$

  In this case, we send 256 modified session tickets to the server and evaluate the server's responses. A vulnerable server may answer differently in one case when we hit for $z = z_{validPadding2}$ a valid 2-Byte padding. Then, we can reconstruct the original plaintext value $p_{15} = z_{validPadding2} \oplus 0x02$.

Second, we assume that the server uses the *Pkcs00* padding scheme. The procedure is analog to the *Pkcs01* padding, but there are slight differences on how we manipulate the session ticket. For this scheme a valid 2-Byte padding is "01 01" so that $p''_{15} = p''_{16} = 01$. From step 1, we assume that $p_{16} = z_{validPadding}$. As we know the value of p_{16}, we construct c''_{16} so that the modified plaintext p''_{16} results in "01":

$$\begin{aligned} c''_{16} &= c_{16} \oplus 0x01 \\ \Rightarrow p''_{16} &= p_{16} \oplus 0x01 = 0x01, \end{aligned}$$

Next, we construct c''_{15} so that we create the 2-byte padding at the second last byte $p''_{15} = 01$. Here, we distinguish as for the *Pkcs01* scheme between two different cases depending on the value of p_{15}.

- $p_{16} \neq 0x00$: In this case, we assume that the original session ticket contains padding larger than 1 byte. Our goal is to modify c''_{15} such that $p''_{15} = 0x01$:

  $$\begin{aligned} \rightarrow p''_{15} = p_{15} \oplus z = 0x01, \quad & p_{15} \in paddingValues_{Pkcs00} \setminus \{01\} = \{01, ..., 0f\} \\ & z \in Z_2 = ? \end{aligned}$$

  We see that the problem is nearly identical compared the 1-Byte padding for the *Pkcs01* scheme. Therefore, we use the same values for $z \in Z_2 = Z_{0x01} \setminus \{0x11\}$. We only exclude the edge value 0x11, because it belongs to the byte value 16 which is not allowed in the *Pkcs00* scheme.

- $p_{16} \neq 0x00$ If the session ticket only contains a 1-Byte padding, we do the exact same procedure as for the *Pkcs01* scheme. We then send 256 modified session tickets to the server and evaluate if we receive one different answer for $z = z_{validPadding2}$. Then, the only difference is the reconstruction of the original plaintext byte p_{15}. It works as follows: $p_{15} = z_{validPadding2} \oplus 0x01$.

5.2.13 Implementation Details

In this section, we explain some implementation specific details of our padding oracle detection algorithm:

- We have to detect different server responses to detect a padding oracle. TLS-Attacker classifies two server responses as equal if the following properties are equal:
  - number of received messages
  - order of received messages
  - message types of all received records
  - if alert messages are sent, then the content of the alert has to be equal

  - records inclusive fragmentation
  - socket state

  If one of these conditions does not apply, then the responses are classified as differently.

- Every time, we hit either a valid 1-Byte or 2-Byte padding, we confirm our hypothesis with a statistical test. Consider the case that we hit a valid 1-Byte padding. We have sent 17 different ticket modifications to the server and the server answered in one case different. To confirm our hypothesis, we resend each of the 17 modifications 10 times to the server and store the responses. Then, we use the `InformationLeakTest` implemented in the TLS-Attacker. It evaluates whether the server responses are deterministically for our sent modifications or not.
- Parallel execution: We try to create 1-Byte paddings for 6 different padding offsets. We execute all the resumption handshakes for the creation of 1-Byte paddings in parallel. In total, these are 102 handshakes (17 per padding offset). All resumption handshakes used for the statistical test (at least 170 handshakes) and the creation of the 2-Byte padding are executed in parallel as well.
- We store the following information for every position where we try to create a valid padding in the `SiteReport`.
  - the servers responses
  - the number of different server responses to the sent modifications
  - if the creation of the padding was successful we additionally store the
    - results of the statistical test
    - server responses for valid padding
    - server responses for invalid padding
    - padding guess for p_{15} and p_{16}

5.3 Testing

In this section, we describe how we verify that our test suites work correctly. We modify the GoTLS[5] implementation so that it is vulnerable to our proposed test suites. GoTLS implements TLS 1.2 and 1.3 so that we can evaluate most test suites for both versions. To verify each test suite we do the following. We manually run the modified GoTLS version which is only vulnerable to the currently tested test

[5]https://pkg.go.dev/crypto/tls

suite and evaluate if our test suite correctly detects it. Additionally, we implement a modified GoTLS version that is vulnerable to several test suites. Thus, we can verify that our implementation detects several vulnerabilities at one host as well. For some test suites, we also verify our test suites with modified versions of OpenSSL and MbedTLS. In the following, we describe for every test suite how we implemented the vulnerable test server:

IV Repetition We implement the vulnerability in GoTLS: We edit the code so that every fourth ticket is issued with the same IV.

Unencrypted tickets We implement the vulnerability in GoTLS: We remove the encryption of the session state and add the plain session state to the session ticket. As a result, the session ticket contains the plain *mastersecret* in TLS 1.2 and the plain *resumption mastersecret* in TLS 1.3.

Cipher suite change We implement the vulnerability in GoTLS: In TLS 1.2, GoTLS checks if the cipher suite in the session resumption is equal to the cipher suite inside the session ticket. We disable the check so that the server allows resumption with other cipher suites. In TLS 1.3, GoTLS does not check if the cipher suite is equal. Thus, the server is automatically vulnerable to the test suite.

Version change We implement the vulnerability in GoTLS: Our test suites evaluates two different cases of version change with session tickets:

1. Resume a session in TLS 1.0/1.1 with a ticket issued in TLS 1.2: In this case, we edit the implementation for TLS 1.2 and older versions in the following way: GoTLS checks if the version in the resumption is equal to the version included in the session ticket. We remove this check.
2. Resume a session in TLS 1.2 with a ticket issued in TLS 1.3 and vice versa: A vulnerable server for this case is more complex because the ticket structure of TLS 1.2 and TLS 1.3 differs. We will not explain the details of our implementation here.

Zero HMAC-key We implement the Zero HMAC-key in OpenSSL and GoTLS. We replace the default *hmac key* with an all-zero key. We implement the test suites for the following HMAC algorithms:

- GoTLS: HMAC-MD5, HMAC-SHA1, HMAC-SHA256(default), HMAC-SHA384, HMAC-SHA512

- OpenSSL: HMAC-SHA256

Zero encryption-key We test all-zero keys for all three evaluated TLS-implementations and their implemented encryption algorithms. We implement additional encryption algorithms in GoTLS which use all-zero keys.

- GoTLS: AES_128_CTR (default), 3DES_CBC, AES_128_CBC, ChaCha20-Poly1305, AES_128_GCM
- OpenSSL: AES_256_CBC
- MbedTLS: AES_GCM and AES_CCM (both with 128 and 256 bit keys)

No Mac check We disable the MAC check in the code of OpenSSL and GoTLS. In Appendix A.2 we can see the test result for a modified GoTLS version.

Padding oracle We implement the padding oracle vulnerability in GoTLS: Normally, GoTLS uses the AES-Counter mode to encrypt session tickets. We replace it with the AES-CBC cipher suite to make the webserver padding oracle vulnerable. As explained in 5.2.12, there are different possibilities where the last block of the ciphertext is located. Therefore, we implement the following vulnerable webserver:

- MAC-then-Encrypt: We implement the MAC-then-Encrypt scheme for session tickets (see Figure 5.23). If the server receives a session ticket with an invalid padding, it will throw an exception internally and this will lead to no response. If the session ticket has a valid padding, the MAC verification will fail. This leads in GoTLS by default to a full 1-RTT handshake.
- Encrypt-then-MAC: We implement the Encrypt-then-MAC scheme (see Figure 5.24). However, we will decrypt the ciphertext before checking the MAC which makes the server padding oracle vulnerable. The server responses are identical to the MAC-then-Encrypt scheme. Additionally, we implement it for the different HMAC algorithms HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512 to test the different positions of the last ciphertext block in our test suite.

We implement the two different padding schemes *Pkcs00* and *Pkcs01* for both schemes. We also test all possible padding sizes inside the session ticket and especially the edge case that the session ticket contains a 1-Byte padding.

Replay attack OpenSSL is the only of the three evaluated TLS-implementations which support early data. Therefore, we use it to configure a vulnerable server. We do not edit the code, because we can configure all properties at the server start. OpenSSL does not support early data per default, so we have to include the *early_data* flag. If early data is enabled, OpenSSL turns per default the replay protection on. Thus, we include the *no_anti_replay* flag in the server configuration:

```
$ openssl s_server -key key.pem -cert cert.pem -early_data -no_anti_replay
```

6 Large-scale scanning of Internet

In this chapter, we present the result of our large-scale scan. First, we present the tools that we used for our large-scale scan. Then, we will shortly describe some scan details as the runtime. Finally, we present the results of the large-scale scan for the implemented test suites.

6.1 Used Tools

The TLS-Crawler tool allows us to perform large-scale scans on the Internet. The project is not publicly available and maintained by the Chair for Network and Data Security from the Ruhr-University Bochum and the Research Group Systems Security from the Paderborn University. While TLS-Scanner only allows the scanning of one specific host, TLS-Crawler spawns several instances of the TLS-Scanner such that multiple hosts are scanned in parallel. All the results of the implemented test suites are written to a Mongo Database. We will not explain the functionality of the TLS-Crawler in more detail, because it is not relevant for this thesis. The large-scale scan is executed on the high-performance computer provided by the *RUB-NDS*.

6.2 Scan Details

We performed a large-scale scan of the Top Million hosts included in the Tranco list[1] [14]. The scan took around 20 days. In order to evaluate the *passive* test suites, we evaluate 10 issued session tickets for every host. As explained in Section 5.2.11.1, the test suite *Zero Encryption Key* provides different ticket format generation modes. In our large-scale scan, we use the *NORMAL* mode. For performance reasons, we could not execute all test suites for TLS 1.2/1.3. Thus, we excluded the two most costly test suites *NoMacCheck* and *Padding Oracle* test suite for TLS 1.3. However, we evaluated them later in an additional large-scale scan for only 30000 randomly selected hosts in TLS 1.3. For some test suites, we performed additional large-scale scans, but this will be explicitly mentioned in the test suite results.

[1]Available at https://tranco-list.eu/list/KLXW

6.3 Results

In the following, we will describe the results of our large-scale scans. First, we will look at the evaluated session ticket support in the Tranco Top Million host list for TLS 1.2/1.3. Then, we will describe the results of the session ticket sizes and the used session ticket formats. Finally, we present the results for all implemented test suites.

6.3.1 Session Ticket Support

First, we evaluate the session ticket support in the scanned Tranco Top million host list. We evaluate for TLS 1.2/1.3 the following three properties:

1. Number of hosts supporting TLS 1.2/1.3.
2. Number of hosts issuing a session ticket.
3. Number of hosts accepting the issued session ticket in the resumption.

Table 6.1 shows the results for TLS 1.2 ordered by the host ranks. In total, 75.9% of all webservers support TLS 1.2 and 59.8% have issued a TLS 1.2 session ticket. 54.7% of all hosts accepted the issued session ticket. As we can see, not all servers

Table 6.1: TLS 1.2 Session ticket support results for Tranco Top Million hosts

Tranco rank	supports TLS 1.2	issued TLS 1.2 ticket	accepted ticket
Top 1k	898	672	529
Top 10k	8570	6020	4993
Top 100k	81401	60582	54145
Top 1M	759763 (75.9%)	594238 (59.8%)	547159 (54.7%)

that issued a session ticket also accepted it in the resumption. We shortly want to discuss possible reasons why webservers may reject a session ticket. We tested a small number of webservers that rejected the session ticket and observed two different patterns. 1. Webservers always reject the issued session tickets. 2. Webservers sometimes reject an issued session ticket. In the first case, an explanation is that the server is not configured correctly and therefore rejects all tickets. In the second case, an explanation is that we connect to different Load Balancers in the initial and the resumption session. If the Load Balancers session caches are not synchronized correctly, then the server may reject the session ticket, because they are not able to find the corresponding STEK.

Table 6.2 shows the results for TLS 1.3. In total, 44.9% of all webservers support TLS 1.3 and 39.0% have issued a TLS 1.3 session ticket. 37.2% of all hosts accepted the issued session ticket. Interestingly, the session ticket support increases for higher Tranco ranks. As observed in TLS 1.2, in TLS 1.3 not all webservers that issued session tickets also accepted it.

Table 6.2: TLS 1.3 Session ticket support results for Tranco Top Million hosts

Tranco rank	supports TLS 1.3	issued TLS 1.3 ticket	accepted ticket
Top 1k	453	339	295
Top 10k	4485	3296	3070
Top 100k	45541	38640	36991
Top 1M	441286 (44.9%)	390792 (39.0%)	372906 (37.2%)

6.3.2 Session Ticket Format

In the following, we evaluate the session ticket sizes and formats for the Tranco Top Million hosts. At the end of the Section, we also describe some interesting observations.

First, we consider the sizes of the issued session tickets. In Table 6.3, we see the evaluated session ticket sizes. In TLS 1.2, nearly all-session tickets (99.9%) have a size between 104 to 260 bytes. In TLS 1.3, 90% of the issued session tickets are in the same size range. Additionally, around 9.8% of all session tickets have a size of 32 bytes. This results from the fact that in TLS 1.3 both resumption mechanisms (session ticket and session-cache) were combined into the Pre-Shared-Key mechanism. In the session mechanism, a session identifier (typically 32-byte long) is sent to the client. In TLS 1.2, the session identifier is sent in an additional *SessionID* extension . However, in TLS 1.3, the session identifier is sent like the session ticket via the *NewSessionTicket* message. Thus, in TLS 1.3, we assume that all session tickets with a 32-byte size are session identifiers for session resumption. Interestingly, a small number of hosts issue session tickets larger than 260 bytes.

Table 6.3: Session ticket sizes from hosts in Tranco Top 1 Million list

ticket length in bytes	TLS 1.2	TLS 1.3
32	12 (<0.01%)	38506 (9.8%)
104-260	593871 (99.9%)	351986 (90.0%)
261-999	342 (0.06%)	269 (0.06%)
1000-7000	13 (<0.01%)	9 (<0.01%)

In the RFC 5077, it is recommended to include a *key name* size of 16 bytes. Our main goal is to evaluate how many servers in the wild follow the recommendation.

For this, we calculated the longest common prefix of two issued session tickets by a server. We assumed that the result is the *key name* size of the server. However, after performing the large-scale scan we noticed the following problem: For around 10 % of all webservers we could not identify a common prefix larger than 0. We randomly evaluated some of these servers in more detail. For this, we connected several times to these servers and evaluated their issued session tickets. We describe exemplarily the pattern we observed for session tickets of many servers. We evaluated four tickets that were issued directly one after the other. The first and the third ticket contained the longest common prefix of 16 bytes. The same goes for the second and fourth tickets. However, the longest common prefix between the first and the second ticket was zero. One explanation for that scenario is that we connect to different Load Balancers and each of them may use a different *key name*. Thus, we reevaluated all hosts with a zero *key name* size by calculating the longest common prefix for any pair of ten issued session tickets. Note that we excluded session tickets with 32 bytes in TLS 1.3 for our evaluation. As explained before, these tickets are used as session ids and do not include any *key name* at all. Table 6.4 shows the evaluation results for the Tranco Top Million hosts. The results for TLS 1.2 and TLS 1.3 are very similar. The vast majority of hosts (98.7%) use the recommended *key name* size of 16 bytes. 0.9% do not use any *key name* at all. However, it is also possible that these servers issued ten tickets with ten different *key names* so that we were not able to detect the correct size.

Table 6.4: *key name* sizes from hosts in Tranco Top Million list

KeyName length in bytes	TLS 1.2	TLS 1.3[a]
0	5503 (0.9%)	3946 (0.9%)
4-12	1508 (0.2%)	16 (<0.01%)
16[b]	586795 (98.7%)	348319 (98.8%)
20	284 (0.04%)	0
107-200	141 (0.02%)	0

[a] We do not include tickets with a size of 32 bytes in the results. As explained before, they are used as *session IDs* and therefore do not contain any *key name* at all.
[b] Recommended *key name* size in RFC 5077.

The RFC 5077 recommends that the *key name* consists of random bytes. We detected that some servers only included ASCII values in their *key names*. Thus, we evaluated this property for all hosts in TLS 1.2. In total, 14132 hosts used a *key name* with only ASCII values.

Moreover, we looked more closely at the 141 hosts with *key names* larger than 100 bytes. We evaluated the HTTP headers of these servers with Curl[2] and identified

[2]https://curl.se/

them as Microsoft IIS 8.5 webservers[3]. These webservers are integrated into Windows 8.1 and Windows Server 2012 R2. We noticed that these webservers include an ASN.1 encoded object containing a DPAPI object in their session ticket [24]. We could see that some Microsoft servers include the path to the corresponding STEK file in their session ticket. In the appendix A.3, we show two exemplary session tickets from Microsoft IIS 8.5 webservers.

6.3.3 IV Repetition

In the *IV Repetition* test suite we evaluate if a webserver sents duplicated IVs for ten issued session tickets. In the recommended ticket format, the IV is located in the session ticket from index 16 to 32. We only evaluate this test suite for the recommended IV position. Our scan results show, that in TLS 1.2 143 webservers and in TLS 1.3 one webserver may be vulnerable. However, we have to filter out false-positive webservers which may use a different ticket format. We see in TLS 1.2 that 141 of the 143 detected webservers have a *key name* length larger than either 107 or 120 (see Table 6.4). As explained in Section 6.3.2, we identified them as Microsoft webservers that use a totally different session ticket structure. They do not follow the recommendation for the IV position. Thus, we identify them as false positives. After filtering out all false positives, we get the final result for the *IV Repitition* test suite (see Table 6.5). Three webservers are vulnerable. Since they use the recommended *key name* size, we can assume that they use the recommended IV position as well.

- A Brazilian website repeats an IV vector four times in TLS 1.2.
- A governmental website from the United Arab Emirates repeats an IV vector three times in TLS 1.2.
- A Finnish website repeats an IV vector twice in TLS 1.3.

Table 6.5: Scan results for the *IV repitition* test suite

	Vulnerable hosts	
Tranco rank	TLS 1.2	TLS 1.3
Top 1M	2	1

6.3.4 Unencrypted Ticket

The test suite *Unencrypted Ticket* evaluates if webservers issue unencrypted session tickets so that the session secrets are sent in plain over the internet. In the Tranco

[3]https://de.wikipedia.org/wiki/Microsoft_Internet_Information_Services

Top Million host list, we did not find any webserver that issued unencrypted session tickets (see Table 6.14).

Table 6.6: Scan results of the *Unencrypted Ticket* test suite for Tranco Top Million hosts

	Vulnerable hosts	
Tranco rank	TLS 1.2	TLS 1.3
Top 1M	0	0

6.3.5 Zero Encryption Key

The test suites *Zero Encryption Key* evaluates if a host uses an all-zero STEK to encrypt its session tickets. In TLS 1.2, this allows an attacker to passively decrypt the related sessions even after the STEK is rotated. Additionally in TLS 1.2/1.3, an attacker is able to perform a Man-In-The-Middle attack while the host is vulnerable. During our test scans, we discovered that webservers hosted by Amazon Web Services (AWS) and Stackpath were vulnerable to our implemented test suite. Because the security impact of this vulnerability is so high, we immediately informed both companies. We did not have the time to scan the entire Tranco Top Million hosts to find all vulnerable hosts related to these companies. In the following, we present the evaluation results for vulnerable servers hosted by both companies. Finally, we present the final results of the Tranco Top Million Scan.

6.3.5.1 AWS

In an initial test scan, we evaluated the Alexa Top 1000 hosts and found out that around 15 hosts used an all-zero STEK to encrypt their session tickets. All of them were hosted by AWS. The tickets were encrypted with the *AES-256-CBC* encryption algorithm and the ticket format was identical to the session ticket format used in OpenSSL (see Table 3.1). Then, we immediately tried to scan the Tranco Top Million hosts to learn how many hosts are affected on a larger scale. However, at that time the runtime of our large-scale scan was too high. For that reason, we periodically scanned the Tranco Top 100k host list and informed AWS about our findings. AWS fixed the vulnerability and informed their customers [1]. They explain that the vulnerability was introduced in September 2020 and entirely fixed in April 2021. In the following, we will describe the results of our scans in more detail:

Scan Results In total, we performed seven scans of the Tranco Top 100k hosts until AWS fixed the problem. Note that one scan (45000 hosts) was not complete. We performed the scans from the 8th to the 13th of April. One entire scan took around 19 hours. In all scans combined, we found 1903 vulnerable AWS hosts in the Tranco Top 100k list (see Table 6.7). Interestingly, all vulnerable webservers support TLS 1.2 but not TLS 1.3. Thus, the effect of this security vulnerability is more severe. In TLS 1.2, after the vulnerability has been closed, it is still possible to passively decrypt all recorded sessions where vulnerable session tickets were either issued or redeemed. The only condition for that is that the corresponding traffic has been recorded.

Table 6.7: Vulnerable AWS hosts found in seven scans for Tranco Top 100k list

	Vulnerable AWS hosts	
Tranco rank	TLS 1.2	TLS 1.3
Top 1k	27	-
Top 10k	302	-
Top 100k	1903	-

In the following, we look at the results of every individual scan run (see Figure 6.8). On average, we found 645 vulnerable AWS hosts per scan of the Tranco Top 100k hosts. In every scanning run, we found new vulnerable AWS hosts. Important to mention is that not every vulnerable AWS host was detected as vulnerable in every scanning run.

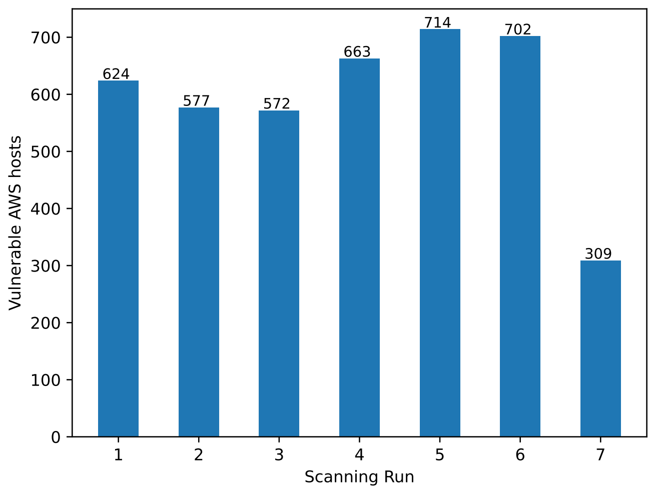


Figure 6.8: Vulnerable AWS hosts for every seven scans of Tranco Top 100k hosts

Thus, it is very likely that our results do not include all vulnerable AWS servers of the Tranco Top 100k list. Additionally, we count for every found vulnerable AWS host, how many times they were vulnerable in our seven scans (see Figure 6.9). On average, every host was vulnerable 2.1 times. However, AWS webservers in higher ranks tend to be vulnerable more often: The average count in the Tranco Top 1k hosts is 2.5 times, in the Tranco Top 10k hosts 2.3 times and in Tranco Top 100k 2.1 times.

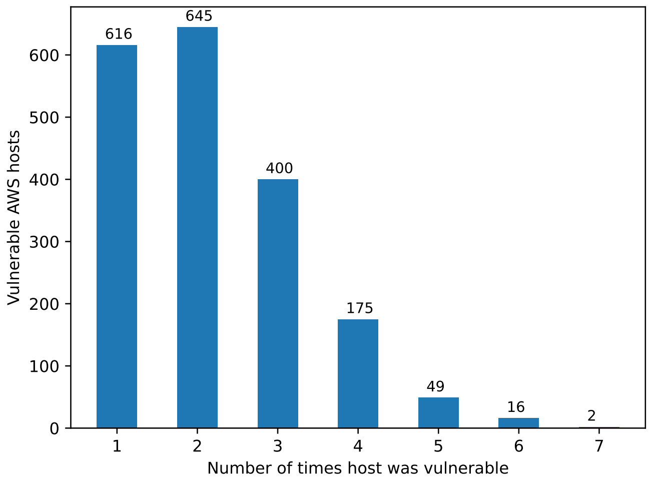


Figure 6.9: Number of times AWS hosts were vulnerable for seven scanning runs of Tranco Top 100k

Evaluation of Daily Patterns After we discovered the vulnerability in AWS hosts, we noticed the following property: If we discovered a vulnerable host at for example 14:00, then the host was vulnerable again the next day at a similar time. Thus, over an entire day, we periodically scanned a smaller sample of vulnerable AWS hosts to observe some time-related patterns. Our sample consisted of 138 AWS hosts from the Tranco Top 10k which we were vulnerable in one of our initial test scans on the 7th of April between 15:00 and 18:30. We started our scan on the 8th of April at 19:00 and reevaluated the sample about every few hours until 17:00 the next day. Figure 6.10 shows the results of our scans. Initially, at 19:00 around 60% of the scanned hosts were vulnerable. Then, the number decreased so that around 3:00 to 6:00 only around 20 % of the host were vulnerable. Until 15:00, the number of vulnerable hosts significantly increased to about 75%. At 18:00 we observed the same number of vulnerable hosts. From the results we can see the following pattern: Two days ago we found 138 vulnerable servers, 2 days later a large amount of these servers is vulnerable at the same time. Only a small amount of the 138 vulnerable webservers was vulnerable at different times. Likely, many vulnerable hosts in the Tranco Top 10k are regularly or daily vulnerable around the same time.

In our seven large-scale scans of the Tranco Top 100k, we evaluated per host 15 issued tickets and checked if they are vulnerable. However, at that time, the implemented test suite did store the number of vulnerable tickets or the tickets themselves. Thus, we have no data on how many of the issued tickets were vulnerable. We introduced this feature for the previously described scans of 138 vulnerable servers to evaluate the daily patterns. If the server was detected as vulnerable, then on average 13.8 tickets were issued with an all-zero STEK. Thus, we have to assume that most vulnerable hosts issued in their vulnerability period mainly vulnerable tickets. This means for the vulnerability periods that all sessions may be affected and passive decryption is still possible.

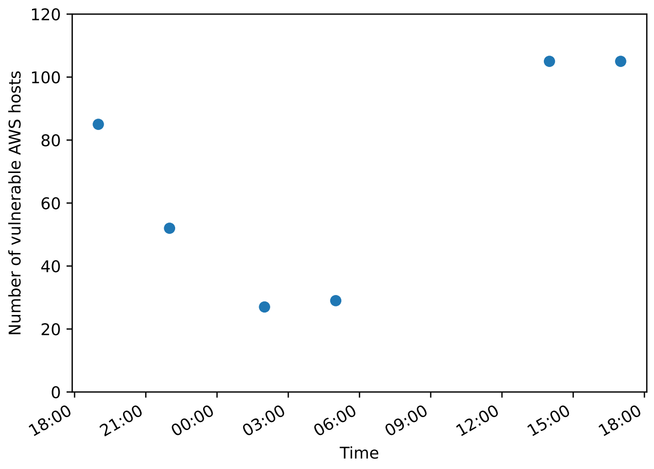


Figure 6.10: Results of periodic scan of 138 vulnerable AWS hosts

Redirections AWS informed us that they identified the problem and fixed the vulnerability. All vulnerable servers were Nginx webservers using OpenSSL. The vulnerability came up because of a bug in their STEK rotation in Nginx. Additionally, they informed us that many of the found hosts were redirecting to a different domain (for example *website.com* redirects to *www.website.com*). For simplicity, we will call the domain to which is redirected the *final domain*. We call the domain that we access at first the *initial domain* which may redirect to a *final domain*. We observed that in many cases the *final domain* was not contained in the Tranco Top 100k host list and thus was not scanned during our scans from 8th April to 13th April.

In the following, we explain the security impact if vulnerable hosts were redirecting. If the *initial domain* is a redirector, then in total two TLS handshakes are performed: First, TLS handshake with the *initial domain*. Second, TLS handshake with the *final domain*. The final session is established with the *final domain* which means that all application data is sent over this connection. If the *initial domain* was vulnerable and the *final domain* was not, an attacker can not decrypt the recorded application data of these hosts. However, during the time the *initial domain* was vulnerable, an attacker could break the authenticity of the server and perform a Man-In-The-Middle attack. If the *final domain* was also vulnerable, then passive decryption of application data still is possible.

As explained before, the *final domains* were not included in our large-scale scans and therefore we do not know if they were also vulnerable. Moreover, we learned from the redirecting after the vulnerability was fixed, so that we could not scan the *final domains* with our test suite. Nevertheless, we do additional analysis to estimate how many *final domains* were also affected so that we can estimate for how many hosts passive decryption still is possible. In the first step, we evaluate how many of 1903 detected vulnerable AWS hosts are redirectors. For this, we write a python script that sends an HTTPS request to the *initial domain* and evaluates if we are redirected to a different *final domain.* If yes, then we assume that the *initial domain* is a redirector. In total, 689 vulnerable AWS hosts do not redirect to a different domain (see Category 1 in Table 6.11). Thus, we assume that passive decryption for these hosts is still possible. For 48 hosts, the evaluation failed (see Category 5 in Table 6.11). All remaining hosts were redirectors. In the second step, we evaluate all redirecting *initial domains* and their corresponding *final domains* in more detail. Our goal is to find out for every host, if the *final* and the *initial domain* share their STEKs. If yes, then it is very likely that the *final domain* was vulnerable as well. To evaluate this, we compare the 16-byte STEK identifier for each 250 tickets of the *inital* and *final domain.* If we find one matching STEK identifier pair, then we assume that both domains share their STEKs. It is very important that we compare tickets that were issued around the same time because we observed that vulnerable AWS webservers rotated their STEK very fast. Thus, we do not use TLS-Crawler for that, because TLS-Crawler randomizes the scanning order and we cannot determine that the *inital* and *final domain* are scanned in parallel. For that reason, we construct our own bash script that uses the OpenSSL client to connect to the *inital* and *final domain* and stores respectively 250 session tickets. Then, we use a python script that compares the STEK identifiers of both domains. If we find a matching identifier, then we assume that both domains share their STEKs. In our evaluation, we detected that 536 vulnerable AWS hosts share their STEKs with their *final domains.* It is very likely, that passive decryption of application data for these hosts still is possible.

If we do not find a matching STEK identifier, we evaluate if both domains may have used the same TLS-configuration (OpenSSL+Nginx). We assume that the same TLS-configuration is used when the ticket sizes of the *initial* and *final domains* are equal. We have observed that most TLS-libraries issue tickets with a different ticket size as OpenSSL. If the same TLS-configuration was used, it may be possible that the *final domain* also issued vulnerable tickets. However, we do not have enough information to draw a final conclusion. We detected that 297 vulnerable AWS hosts

Table 6.11: Evaluation results for different categories of vulnerable AWS hosts

	Decryption[a]	MITM[b]	Number of hosts
Category 1 *No redirection*	✓	✓	689 (36.2%)
Category 2 *Shared STEK*	(✓)[c]	✓	536 (28.1%)
Category 3 *No-Shared STEK*	?[d]	✓	297 (15.6%)
Category 4 *Different TLS-configuration*	✗	✓	333 (17.4%)
Category 5 Error	?	✓	48 (2.5%)

[a] Decryption of recorded application data for host still possible.
[b] Host was vulnerable to Man-in-the-middle attack.
[c] Decryption very likely possible.
[d] We do not have enough information to decide if decryption is possible.

fall into this category (see Category 3 in Table 6.11). The last category of hosts is when the *initial domain* is using a totally different TLS-configuration for session tickets than the *final domain*. This is the case when the *final domain* either does not issue session tickets at all or issues session tickets with a different size. Then, we assume that the *final domains* were not vulnerable. Our evaluation has shown that 333 vulnerable AWS hosts use a different TLS configuration for their *initial* and *final domain* (see Category 4 in Table 6.11). Thus, we assume for these hosts that passive decryption of recorded application data is not possible.

In total, 68% of the vulnerable AWS hosts are located in category 1 or 2, the category where we expect that passive decryption of application data still is possible. It shows us that the redirection weakens the vulnerabilities impact of some hosts. However, the majority of vulnerable AWS hosts are very likely still fully impacted.

AWS Vulnerability Cause AWS informed us that the vulnerability was caused by an incorrectly implemented key rotation mechanism. As explained before, vulnerable AWS hosts used Nginx webservers with OpenSSL. They used a customized key rotation mechanism implemented in Nginx. Furthermore, they informed us, that one of their leads to find the vulnerability cause was that Nginx changed their data structure where STEKs are stored[4]. In Appendix A.4, we evaluate the vulnerability cause in more detail and present a possible scenario that may have led to the all-zero STEKs.

6.3.5.2 Stackpath

While we were evaluating the AWS vulnerability, we discovered that websites hosted by Stackpath were also vulnerable. In the seven Tranco Top 100k scans, we discovered 13 Stackpath hosts issuing session tickets with an all-zero STEK. We were able to distinguish them from vulnerable AWS hosts because they were using the *AES-128-CBC* algorithm to encrypt their session tickets instead of *AES-256-CBC* that was used by vulnerable AWS hosts. We immediately informed Stackpath about our findings. Additionally, we used the *yougetsignal*[5] tool to get all websites hosted on the same IP address as the vulnerable Stackpath hosts. In total, 171 hosts including redirectors were hosted on the same IP. On the 27th of April, we scanned this sample once to find out how many additional Stackpath hosts were vulnerable. The scan differs in one respect from our large-scale scans of the Tranco Top 100k where we evaluated 15 tickets per host. Now, we evaluate 1000 issued tickets per host since we observed that vulnerable Stackpath hosts only issued sporadically vulnerable tickets.

Table 6.12: Number of Stackpath hosts vulnerable to *Zero Encryption key* testsuite

Tranco rank	Vulnerable Stackpath hosts TLS 1.2
Top 1k	0
Top 10k	5
Top 100k	17
Top 1M	20
Total	90

In total, 90 webservers hosted by Stackpath were vulnerable (see Table 6.12). The evaluation showed that only a small number of the issued session tickets used an all-zero STEK. On average 14 (1.4%) out of the 1000 issued session tickets per host were vulnerable. Interestingly, all vulnerable tickets had the following property: If we convert the 16-byte STEK identifier value to ASCII, then we see that the first 14-characters consist of a hex value (see Figure 6.13). The last 2 bytes of the STEK identifier were always zero. We could not find out which webserver and TLS-implementation Stackpath is using.

We did not evaluate regular patterns of vulnerable Stackpath hosts as we did it for vulnerable AWS hosts, because significantly fewer hosts were affected.

[4]https://github.com/nginx/nginx/commit/c2d3d82ccbea18f0504fbaceeee6efb62da8d1d8
[5]https://www.yougetsignal.com/tools/web-sites-on-web-server/

```
Key Name Hex:     30 78 37 66 66 [redacted] 61 33 30 00 00
Key Name Ascii:   0  x  7  f  f  [redacted] a  3  0  .  .
```

Figure 6.13: *key name* structure of vulnerable Stackpath webservers

6.3.5.3 Top Million Scan

In the following, we will describe the results of the final Tranco Top Million scan. We will not include vulnerable webservers which are hosted by one of the two presented companies. In total, we found three webservers using an all-zero STEK. We enumerate all three vulnerable hosts with additional information:

1. A Japanese webserver issues permanently vulnerable session tickets in TLS 1.2 and 1.3. We evaluated the HTTP headers and found out that it is an Apache webserver that very likely uses OpenSSL.
2. A Chinese webserver issues permanently vulnerable session tickets in TLS 1.2. We did not find out which webserver is used.
3. Another Chinese webserver issues permanently vulnerable session tickets in TLS 1.2. We evaluated the HTTP headers and found out that it is an Apache webserver that very likely uses OpenSSL.

All vulnerable webservers have in common that the session tickets are encrypted with *AES-128-CBC*. Moreover, not only the STEK consisted entirely of zeros. As in the GnuTLS bug [6], the STEK identifiers consisted entirely of zeros as well.

Table 6.14: Scan results of the *Zero Encryption Key* test suite for Tranco Top Million hosts

	Vulnerable hosts	
Tranco rank	TLS 1.2	TLS 1.3
Top 1M	3	1

6.3.6 Zero HMAC-key

The *Zero HMAC-key* test suite evaluates if a host uses an all-zero HMAC-key to protect its session tickets. We recalculate the HMAC with an all-zero key and compare the result with the appended HMAC in the session ticket. If they do match, then we assume that the host is vulnerable. The results are closely related to the test suites result of the *Zero Encryption Key* test suite. The results of our evaluation show that no webserver uses an all-zero HMAC key without using an all-zero encryption key for its session ticket as well. Thus, we will describe which webservers that were vulnerable to the *Zero Encryption Key* test suite are also vulnerable for the *Zero HMAC-key* test suite. Initially, we did not evaluate the AWS hosts using an all-zero STEK for this test suite, because the idea for this test suite initially came after detecting the vulnerable AWS hosts. For that reason, we evaluated a small sample of stored vulnerable AWS tickets and recalculated the HMAC value. As vulnerable AWS hosts are using OpenSSL, we know that the used HMAC-algorithm is *HMAC-SHA-256*. Moreover, we know over which fields the HMAC is calculated. However, the recalculated HMACs did not match with the HMACs included in the session tickets. Thus, we assume that all vulnerable AWS session tickets are not vulnerable for this test suite. However, it is possible that the *HMAC-key* was only partially initialized. By partially initialized we mean that only parts of the key-value are initialized with zeros. We did not evaluate this further. The Stackpath hosts and the three hosts that were using an all-zero STEK also used an all-zero HMAC-key to protect their session tickets. All vulnerable hosts used the recommended HMAC-algorithm *HMAC-SHA-256* to protect their session tickets.

6.3.7 Ciphersuite Change

The *Ciphersuite Change* test suite evaluates if we can resume a session with a different selected cipher suite than in the initial session. Table 6.15 shows the results of the Tranco Top million scan. In TLS 1.2, we did not detect any vulnerable server. In TLS 1.3, 6.7% of the Tranco Top million hosts are vulnerable to the evaluated test suite.

Table 6.15: Scan results o the *Ciphersuite change* test suite for Tranco Top Million hosts

	Vulnerable hosts	
Tranco rank	TLS 1.2	TLS 1.3
Top 1k	0	107
Top 10k	0	1000
Top 100k	0	10285
Top 1M	0 (0.0%)	66378 (6.7%)

6.3.8 Version Change

The *Version Change* test suite evaluates if we can resume a session in a different version than in the initial session. Table 6.16 shows the results of the Tranco Top Million scan. We did not find any vulnerable server allowing a version change from TLS 1.2 to TLS 1.3 or vice versa. However, a small number of web servers allow resuming a session in TLS 1.0/TLS 1.1 with session tickets issued in TLS 1.2. Almost all servers which are vulnerable for TLS 1.0 are also vulnerable for TLS 1.1. The number of vulnerable servers for TLS 1.1 (1.5%) is slightly higher than for TLS 1.0 (1.4%) because in the Tranco Top million list the support for TLS 1.1 is slightly higher than for TLS 1.0.

Table 6.16: Scan results of the *Version change* test suite for Tranco Top Million hosts

Tranco rank	Version change			
	TLS 1.2 to TLS 1.0	TLS 1.2 to TLS 1.1	TLS 1.2 to TLS 1.3	TLS 1.3 to TLS 1.2
Top 1k	12	13	0	0
Top 10k	85	85	0	0
Top 100k	928	1004	0	0
Top 1M	14095 (1.4%)	15373 (1.5%)	0	0

6.3.9 No Mac Check

The test suite *No Mac Check* evaluates if webservers verify the MAC of a session ticket in the session resumption. Therefore, we modify each byte of the session ticket and resend it to the server and evaluate if the server accepts the ticket. We scan the test suite for all hosts included in the Tranco Top million list. However, for performance reasons, we only scan the test suite for TLS 1.2. Initially, the scan results indicated that 90 web servers are vulnerable. We tried to confirm the results and discovered a minor bug that led to the number of false positives. However, the bug did not influence the remaining results. After fixing the bug, we rescanned the 90 websites and it turned out that they are not vulnerable (see table 6.18). Table 6.18 shows the results of the scan: No server is vulnerable to the test suite. Additionally, we evaluated 30.000 randomly selected hosts from the Tranco Top million list in TLS 1.3. As in TLS 1.2, we did not find any vulnerable host.

Table 6.17: Scan results of the *No Mac Check* test suite for Tranco Top Million hosts

Tranco rank	Vulnerable hosts TLS 1.2
Top 1M	0

6.3.10 Padding Oracle

The test suite *Padding Oracle* evaluates if a host is vulnerable to padding oracles in the context of session tickets. For performance reasons, we only scan the test suite for TLS 1.2. We sent modified session tickets to the webserver and evaluate if the server responds differently to our modifications. With this, we try to create a valid 2-Byte padding for a session ticket. As we do not know the position of the last ciphertext block in the session ticket, we try to create the 2-Byte padding at 6 different positions where the last block may be located. Additionally, we use statistical tests to confirm our evaluations. In the case, we hit a valid padding we send all ticket modifications 10 times to the server and evaluate if the server reacts deterministically to our modifications. It may happen that we only create a 1-Byte padding, but then the creation of the 2-Byte padding fails. In this case, we output that we created a 1-Byte padding.

Table 6.18: Scan results of the *Padding Oracle* test suite for Tranco Top Million hosts

Tranco rank	Vulnerable hosts TLS 1.2
Top 1M	0

In the Tranco Top Million, we did not find any vulnerable servers. Our evaluation results only showed that we created 149 1-Byte paddings and no valid 2-Byte padding. We re-scanned all hosts where we created a 1-Byte padding, but we could not create this padding again. We assume that these hosts answered non-deterministically and the statistical test did not have enough data to detect it. Likely, the number of repetitions (10) that we perform for our statistical test was too low. As a consequence, we get a small number of false-positive results. However, we did not detect any valid 2-Byte padding. Thus, we assume that no vulnerable server was found.

Our test suite only considered webservers which answered with two different responses to modified session tickets. However, as explained in section 5.2.12, if the MAC is not validated at all a vulnerable server may respond with 3 different responses. Thus, we made some further evaluations. For every webserver we stored the number of different responses at every position. We evaluate all 117 servers that responded at exactly one position with three with 3 different responses and at all other positions uniformly. If the server responded at other positions differently, then we assume that the server responds non-deterministically. For all 117 webservers, we sent similar ticket modifications to the server and evaluate if the server responds again differently. If yes, we use a statistical test with 100 repetitions. It turned out none of the evaluated servers responded significantly differently for ticket modifications. Thus, we assume that they are not padding oracle vulnerable.

Additionally, we evaluated 30.000 randomly selected hosts in TLS 1.3 for the *Padding Oracle* test suite, but we did not detect any vulnerable server.

In the following, we want to discuss possible explanations for why no vulnerable server was found. One explanation could be that none of the servers is vulnerable since all servers which use the CBC mode also use the secure Encrypt-then-MAC scheme. Our results from the library analysis support this explanation since all of the evaluated TLS-libraries using CBC also used the Encrypt-then-MAC scheme (see Section 3.2). Furthermore, the RFC 5077 implicitly recommends using this scheme. In contrast to that, in TLS CBC cipher suites are used per default with the MAC-then-Encrypt scheme to protect application data. Another explanation is that we overlooked vulnerable servers that use a totally different session ticket format as for example Microsoft webservers (see Section 6.3.2). In this case, we may have not guessed the correct position of the last ciphertext block and thus were not able to create a valid padding. One last explanation is that webservers using the MAC-then-Encrypt scheme, do not offer a padding oracle. We evaluated the source code of OpenSSL to analyze how the server responds if the session ticket decryption fails. The server responded for an invalid MAC or decryption fail always identical with a full handshake. Thus, even if OpenSSL would use the MAC-then-Encrypt scheme, our test suite could not distinguish valid from invalid paddings.

Creating Padding Oracles for Webservers Vulnerable to the Zero HMAC-key Test Suite After we found webservers that are vulnerable to the *Zero HMAC-key* test suite, we had the following idea. We make a customized padding oracle attack for these webservers. For this, we modify the session tickets in the same way as in the original test suite, but we recalculate and append the MAC for every modified session ticket before sending it back to the server. We can do the recalculation because we know the value of the all-zero HMAC-key. We rescanned all webservers which were vulnerable to the *Zero HMAC-key* test suite. At one Chinese webserver, we were able to create a valid 2-Byte padding. We were able to confirm our results by comparing our padding guesses with the padding values inside of the session ticket. This was possible because the server also used an all-zero encryption key. The vulnerable server responded in two ways: 1. If the padding was valid, first they responded with a full handshake. After we sent the CCS and FIN message, they responded with an Alert *Unexepcted Message* (see Figure 6.19).
2. If the padding was incorrect, they responded also with a full handshake. How-

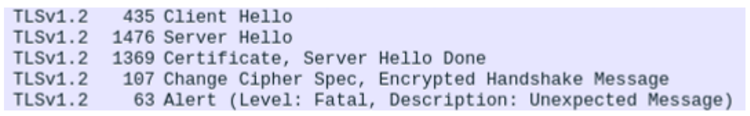
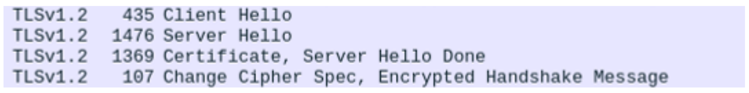

```
TLSv1.2   435 Client Hello
TLSv1.2  1476 Server Hello
TLSv1.2  1369 Certificate, Server Hello Done
TLSv1.2   107 Change Cipher Spec, Encrypted Handshake Message
TLSv1.2    63 Alert (Level: Fatal, Description: Unexpected Message)
```

Figure 6.19: Vulnerable Chinese webserver responding to a session ticket with a valid padding

ever, after we sent the CCS and FIN message, they did not respond at all (see Figure 6.20).



```
TLSv1.2   435 Client Hello
TLSv1.2  1476 Server Hello
TLSv1.2  1369 Certificate, Server Hello Done
TLSv1.2   107 Change Cipher Spec, Encrypted Handshake Message
```

Figure 6.20: Vulnerable Chinese webserver responding to a session ticket with an invalid padding

With this padding oracle, we are able to learn the session secret stored inside the session ticket. This allows an attacker to decrypt recorded application data.

6.3.11 Replay Attack

The test suite *Replay Attack* evaluates if a webserver allows an attacker to replay early data in TLS 1.3. In our large-scale scan, we evaluate if webservers support early data. If yes, then we try to replay early data and evaluate if the servers accept it twice. If yes, then we assume that the server is vulnerable to replay attacks. Important to mention is that the scan only evaluates replay vulnerability on the TLS-layer. Webservers may prevent replay attacks on the application layer. In our initial test scans (around 80k hosts), the scan results showed that 129 webservers (0.01%) support early data. Because we expected higher early-data support, we made some additional research. In 2017, Cloudflare announced that it enables 0-RTT support per default for all websites using their free plan. Paid customers have the option to enable it [18]. According to W3Techs around 10 to 15% of all websites are using Cloudflare [2]. However, our early data scan results did not include any popular Cloudflare webserver as for example *cloudflare.com*. Thus, we used two different TLS scanner tools to evaluate if Cloudflare webservers support early-data: *SSLabs*[6] and *SSLyze*[7]. We scanned a small sample of popular Cloudflare webservers with both tools. *SSLyze* displayed that the webservers support early-data and *SSLabs* did not. We tried to confirm the *SSLyze* results with the OpenSSL client and we observed that the session tickets were issued around ten seconds after the handshake was finished. Our implementation has a default timeout of two seconds and therefore our connection reaches the timeout before the issued session ticket arrives. Without any issued session ticket, we skip the replay attack test suite and assume that the server is not vulnerable. We fixed the problem by sending a dummy HTTPS request message after every handshake because then Cloudflare webservers issue immediately a new session ticket.

Finally, Table 6.21 shows the result of the Tranco Top Million scan with our adapted implementation. In total, 3.38% of the webservers support early data. Nearly all of them are vulnerable to replay attacks (3.37%). These results also confirm the findings of Mihael Liskij in his master thesis. He discovered that the replay

[6]https://www.ssllabs.com/ssltest/
[7]https://github.com/nabla-c0d3/sslyze

Table 6.21: Scan results of the *Replay Attack* test suite for Tranco Top Million hosts

Tranco rank	early-data support	replay vulnerable
Top 1k	42	42
Top 10k	457	454
Top 100k	3972	3958
Top 1M	33898 (3.38%)	33758 (3.37%)

protection in Nginx, which is per default turned on, was not working correctly [15].

7 Conclusion and Future Work

In this thesis, we first evaluated different TLS-libraries according to their session ticket handling. In the next step, we presented and implemented different test suites for vulnerabilities that may appear in the session ticket handling of webservers in the wild. Finally, we evaluated in a large-scale scan of the Tranco Top Million hosts how many webservers were vulnerable to the presented vulnerabilities.

The most severe finding was that several thousand websites hosted by AWS used around six months regularly an all-zero STEK to encrypt their session tickets in TLS 1.2. The vulnerability was very similar to the mentioned GnuTLS bug [11] which was the motivation for our thesis. In both cases, an incorrectly implemented key rotation mechanism caused the vulnerability. Further evaluations have shown that for at least 68% of the vulnerable AWS websites, likely recorded traffic can still be decrypted even after the vulnerability was fixed. The all-zero STEK vulnerability affected a smaller hosting provider and three singular webservers as well. Since we observed this vulnerability in several different cases, we assume that there is systematic risk in different TLS-implementations for this type of vulnerability. How can we prevent that this vulnerability occurs again on a larger scale again? A countermeasure against an all-zero STEK is to implement a zero-check before initializing the STEK. Even if a wrongly implemented key rotation mechanism outputs a zero value, the check prevents that the STEK is initialized with all-zeros. In future work, it could be interesting to evaluate if popular TLS-libraries implement this countermeasure. Moreover, we would like to see that this countermeasure is added in many TLS-libraries. This feature is especially useful in OpenSSL since it supports the implementation of customized key rotation mechanisms. Another idea is to extend the countermeasure to detect if a larger part of the STEK is uninitialized. If for example only the first byte of the STEK is initialized, then the zero-check will not detect it. As a consequence, attackers can simply reconstruct the STEK with little effort.

In our large-scale scan, we only evaluated the all-zero STEK test suite for HTTPS webservers. However, TLS is also used in E-Mail Transfer protocols as SMTPS or IMAPS. Thus, we suggest performing large-scale scans of E-Mail webservers to evaluate if a larger amount of servers also use an all-zero STEK.

Another interesting evaluation result is that nearly all webservers that are supporting 0-RTT session resumption in TLS 1.3 are vulnerable to replay attacks. However, we only evaluated the vulnerability on the TLS-layer. Webservers could prevent these attacks by implementing countermeasures on the application layer. Future work should evaluate if the detected vulnerable webservers prevent replay attacks on the application layer. Ideally, this can be automatized in TLS-Scanner.

A novel insight from our large-scale-evaluation is that around 98.7% of all webervers supporting session tickets in the Tranco Top Million list follow the recommendation from the RFC 5077 to include a 16-byte STEK identifier in their session tickets. We can use this information in future large-scale evaluations of session tickets.

Another interesting finding is that Microsoft webservers used a totally different session ticket format than recommended. They encode their session tickets as an ASN.1 object and include sensitive information as the STEK file path in their session ticket. In future work, it would be interesting to find out in more detail what other values are included in the ticket and if servers may be vulnerable to different attacks.

A Appendix

A.1 Padding Oracle

p_{16}	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	10
$z \in Z_{0x01}$	00	03	02	05	04	07	06	09	08	0b	0a	0d	0c	0f	0e	11
$p'_{16} = p_{16} \oplus z$	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01

Table A.1: Calculation of Z_{0x01}

p_{16}	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
$z \in Z_{0x00}$	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
$p'_{16} = p_{16} \oplus z$	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Table A.2: Calculation of Z_{0x00}

p_{15}	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	10
$z \in Z_2$	00	01	06	07	04	05	0a	0b	08	09	0e	0f	0c	0d	12
$p''_{15} = p_{15} \oplus z$	02	02	02	02	02	02	02	02	02	02	02	02	02	02	02

Table A.3: Calculation of Z_2

A.2 Testing No Mac Check

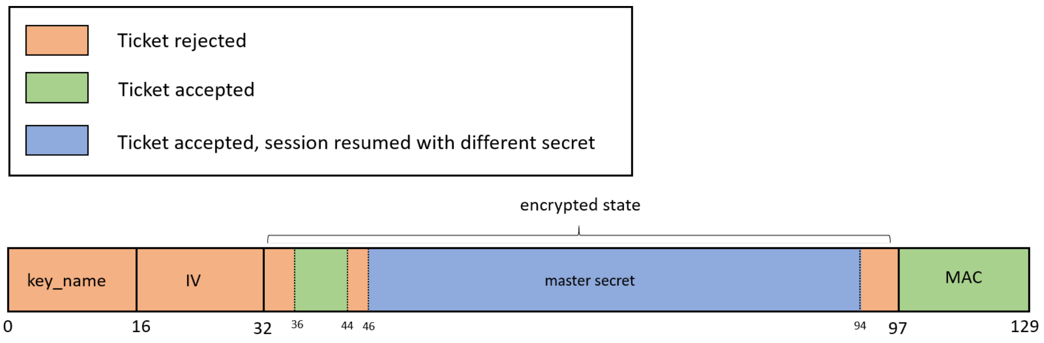


Figure A.4: GoTls servers answers to manipulated tickets

In order to illustrate how a vulnerable server would react to our requests, we evaluate a vulnerable modified GoTLS test server in TLS 1.2. GoTLS uses the counter mode to encrypt session tickets. When we modify the session ticket's encrypted state at index i, the StatePlaintext will only be manipulated at index i as well. If the server uses a different encryption algorithm (for example CBC), several blocks of the StatePlaintext may be altered. Figure A.4 shows the GoTLS server responses to our requests. The color at each byte position i of the ticket represents the server's answer to a manipulated ticket at index i. If we manipulate the key_name, the server does not find any key with the key_name identifier and rejects the ticket. If the IV is modified, the decryption of the whole encrypted state fails, because the IV influences every plain text block in the counter mode. If we modify the session ticket's encrypted state, the reactions are different. At some positions, the ticket is accepted with the same secret and in some positions, the ticket is rejected. If we modify the encrypted state at the positions, where the master secret is located (position 46 - 93), the server accepts the ticket and resumes the session with a different master secret. If we modify the MAC that is not validated, the server accepts the ticket with the same secret.

A.3 Microsoft Session Tickets

We examplary show two different session tickets from two Microsoft IIS 8.5 webservers (see Listing A.5 and A.6). Both contain an ASN.1 encoded object in their session ticket.

```
SEQUENCE {
    OBJECTIDENTIFIER 1.2.840.113549.1.7.3 (envelopedData)
    [0] {
        SEQUENCE {
            INTEGER 0x02 (2 decimal)
            SET {
                [2] {
                    INTEGER 0x04 (4 decimal)
                    SEQUENCE {
                        OCTETSTRING 41fe455500e2014ea14ba79780c3fdb3
                        SEQUENCE {
                            OBJECTIDENTIFIER 1.3.6.1.4.1.311.74.1
                            SEQUENCE {
                                OBJECTIDENTIFIER 1.3.6.1.4.1.311.74.1.12
                                SEQUENCE {
                                    SEQUENCE {
                                        SEQUENCE {
                                            UTF8String 'KeyFile'
                                            UTF8String '%ALLUSERSPROFILE%\Microsoft\Crypto
                                            \TlsSessionTicketKeys\S-1-5-18\SessionTicketKey.key'
                                        }
                                    }
                                }
                            }
                        }
                    }
                    SEQUENCE {
                        OBJECTIDENTIFIER 2.16.840.1.101.3.4.1.45 (id-aes256-wrap)
                    }
                    OCTETSTRING 43439b9b03f279f1ff9370cd9a339b4cce934d47ebf0278d25
                    a30d34c38b358dd2f895ef44ad572
                }
            }
            SEQUENCE {
                OBJECTIDENTIFIER 1.2.840.113549.1.7.1 (data)
                SEQUENCE {
                    OBJECTIDENTIFIER 2.16.840.1.101.3.4.1.46
                    SEQUENCE {
                        OCTETSTRING 5fcb807f75dcaff1710cca50
                        INTEGER 0x10 (16 decimal)
                    }
                }
                [0]
                0263f22062aebbf6b7c1dbd232d7afb54f668a918efd881176c33a47ef03576986
                b1078a0945472a2825e9a7b67a66993963027e81d921928f6b7f8b9ac00cab7d6d
                496d839660841ffcd9fbff336468366a79334d78197bae3bdb51cef84c027be006
                793cd5a7dcfc3868e953b7c14126fde2b1a6d8234f57ca545ea90dcfb5aed15b0f
                be6657d92cb388ee8d3cd295454e269c75f0f39b7e3a46e0511063e25938981eb9
                a4b6b2085d84ef298ddd3c48103d6592d8c407a28da860ae8d73b86ee3151c
            }
        }
    }
}
```

Listing A.5: Examplary session ticket from Microsoft IIS 8.5 webserver. The session ticket contains an ASN.1 encoded object. We can see that the path of the session ticket key file is included in the ticket.

```
SEQUENCE {
    OBJECTIDENTIFIER 1.2.840.113549.1.7.3 (envelopedData)
    [0] {
        SEQUENCE {
            INTEGER 0x02 (2 decimal)
            SET {
                [2] {
                    INTEGER 0x04 (4 decimal)
                    SEQUENCE {
                        OCTETSTRING
                        01000000d08c9ddf0115d1118c7a00c04fc297eb01000000fb4bc7b7545f
                        9842be46126ddf9b69130000000002000000000010660000000100002000
                        0000bec539768acb5de7ffd0a395d7e6fd632207da3cf6cf
                        77ba566809ce394b5183000000000e80000000020000200000000c047f3a
                        55b05189dbd3c9ddaef8f33608b96762fdfcbda0b742296f8be2071a3000
                        00009924541dbe184c01be0a3a1281c6e9dd9b45460febc7669f86a878f6
                        fb37007edf2e3e46a08e12dc7017ea2ad45b94d340000000aadeb5ec5fed
                        6ab4edf5e93917987190cf12269c57e41de3f35062bc79c3d6befc15152c
                        e75da4e3e1e1afad7f75eecd621ddd0e48ec941280fe3ad722f6d268
                        SEQUENCE {
                            OBJECTIDENTIFIER 1.3.6.1.4.1.311.74.1
                            SEQUENCE {
                                OBJECTIDENTIFIER 1.3.6.1.4.1.311.74.1.8
                                SEQUENCE {
                                    SEQUENCE {
                                        SEQUENCE {
                                            UTF8String 'LOCAL'
                                            UTF8String 'user'
                                        }
                                    }
                                }
                            }
                        }
                    }
                    SEQUENCE {
                        OBJECTIDENTIFIER 2.16.840.1.101.3.4.1.45 (id-aes256-wrap)
                    }
                    OCTETSTRING 1f9cde9caa8eea9a33f872d6d5c933e622185e9cd19bfb4f5
                    00ddc8054cdce961007649639fd6f77
                }
            }
            SEQUENCE {
                OBJECTIDENTIFIER 1.2.840.113549.1.7.1 (data)
                SEQUENCE {
                    OBJECTIDENTIFIER 2.16.840.1.101.3.4.1.46
                    SEQUENCE {
                        OCTETSTRING cadea44f62415f6f94818157
                        INTEGER 0x10 (16 decimal)
                    }
                }
                [0] 153636b9e09a01913c4882a046e04d797538cb1ed3445b9663bdebca
                1a15e80851f12d2cafa36d7cb557aa1442be1798bee8b5d4dc99634c0292
                f8fcade46c299569d1e142dd9f71f349b74330dd83afa503744c4f24e1f7
                5d0efd536df184f34b57e109956bb21d5255cb130b4dd9ef563a95f1f79b
                dfbb6ce96cb46d83a932817fde140b9ba0ba6b6ea27a1532484f4f8d04a4
                e098b6e6f651914010f6a69617f4ad8ac0d74c2cabb41d5bd82899849300
                3afa907b151984c6319f7789301f7f4c5de6
            }
        }
    }
}
```

Listing A.6: Examplary session ticket from Microsoft IIS 8.5 webserver. The session ticket contains an ASN.1 encoded object. We can see that the user is encoded in the ticket.

A.4 AWS Vulnerability - Cause

In the following, we explain in more detail what may have caused the vulnerability in AWS hosts. As explained before, vulnerable AWS hosts used Nginx webservers with OpenSSL. Nginx customizes the key rotation of OpenSSL by implementing a callback function (see Section 3.3). AWS informed us that they use a customized key rotation mechanism implemented in Nginx to regularly rotate their STEKs. Important to mention is that their key rotation is directly implemented in Nginx. They do not use the provided session ticket key files for that. According to AWS, their key rotation did not work correctly and therefore sometimes initialized the STEK with all-zeros. They informed us, that one of their leads for causing the vulnerability was that Nginx changed their data structure where STEKs are stored. Thus, we evaluate the Nginx source code and code change in more detail and try to learn what may have caused the vulnerability.

```
typedef struct {
    u_char          name[16];
    u_char          aes_key[16];
    u_char          hmac_key[16];
} ngx_ssl_session_ticket_key_t;
```

Listing A.7: STEK data structure in Nginx before December 2016

```
typedef struct {
    size_t          size;
    u_char          name[16];
    u_char          hmac_key[32];
    u_char          aes_key[32];
} ngx_ssl_session_ticket_key_t;
```

Listing A.8: new STEK data structure in Nginx introduced in December 2016

Until 2016, OpenSSL used *AES-128-CBC* as the algorithm to encrypt their session tickets. Then, OpenSSL switched to *AES-256-CBC* with version 1.1.0[1]. Nginx also supported *AES-128-CBC* for encryption of session tickets until 2016. When OpenSSL switched to *AES-256-CBC*, Nginx also added the support for *AES-256-CBC*[2]. As a result, Nginx now supports both encryption algorithms. If a session ticket key file is provided, then depending on the key file size the corresponding encryption algorithm is selected (see Section 3.3).

In the next step, we evaluate the STEK data structure in Nginx and how it was changed in 2016. Listing A.7 shows the *ngx_ssl_session_ticket_key_t* structure which represents one STEK before the code change. It contains a 16-byte key *name*, a 16-byte *aes_key* and a 16-byte *hmac_key*. Listing A.8 shows the data structure after the code change. They added a *size* field that should include the size of the key file. However, there is not any documentation what the exact matter of the

[1]https://github.com/openssl/openssl/issues/514
[2]https://github.com/nginx/nginx/commit/c2d3d82ccbea18f0504fbaceeee6efb62da8d1d8

field is and which values are legal. Furthermore, you can confuse the field with the key size of the contained key values in the structure. Additionally, they updated the field sizes of both key values to 32 bytes which means that now keys for *AES-256-CBC* can be stored. Interestingly, they also changed the order of the *aes_key* and *hmac_key* inside the struct. The reason for this was to be compatible with the *SSL_CTX_set_tlsext_ticket_keys()* function in OpenSSL which allows to set customized STEKs. However, this function is not documented and we could not understand the behavior of this function in OpenSSL in the source code analysis. Normally, webservers customize the key rotation in OpenSSL with a different callback function[3]. We evaluated the source code of Apache, Nginx, and OpenLiteSpeed and none of them are using this function to set their STEKs.

```

ngx_int_t ngx_ssl_session_ticket_keys(...)
{
  u_char      buf[80];

  ngx_array_t *keys;
  keys = ngx_array_create(cf->pool, paths->nelts, sizeof(ngx_ssl_session_ticket_key_t));

  for (i = 0; i < paths->nelts; i++) { // read all provided key files

    ngx_read_file(&file, buf, size, 0);
    ngx_ssl_session_ticket_key_t* key = ngx_array_push(keys);

    if (size == 48) {
      key->size = 48;
      ngx_memcpy(key->name, buf, 16);
      ngx_memcpy(key->aes_key, buf + 16, 16);
      ngx_memcpy(key->hmac_key, buf + 32, 16);
    } else {
      key->size = 80;
      ngx_memcpy(key->name, buf, 16);
      ngx_memcpy(key->hmac_key, buf + 16, 32);
      ngx_memcpy(key->aes_key, buf + 48, 32);
    }
  }
  SSL_CTX_set_ex_data(ssl->ctx, ngx_ssl_session_ticket_keys_index, keys); //store keys
   array internally in OpenSSL structure
}
```

Listing A.9: *ngx_ssl_session_ticket_keys* function (ngx_event_openssl.c) initializes the STEK from the provided key files

In the following, we shortly explain how the STEK initialization from a key file works at server start. We need this to better understand how a key rotation mechanism may initialize the keys and how Nginx maintains their STEKs (see Listing A.9).

[3]https://www.openssl.org/docs/man1.0.2/man3/SSL_CTX_set_tlsext_ticket_key_cb.html

First, a new *keys* array is created which will store all provided STEKs. Note that the first key in the array is always used to encrypt session tickets, the other only can be used for decryption. In the second step, Nginx creates for every provided key file a new STEK data structure and appends it to the *keys* array. We explain this procedure exemplary for one key file. First, the key file is read and stored in the *buf* variable. Then, the STEK data structure is initialized with the data stored inside *buf*. If the key file only provides 48 bytes, then only the first half *aes_key* and *hmac_key* are initialized. Additionally, the size of the key file is stored. After all key files are parsed, the *keys* array is stored internally in a OpenSSL data structure[4]. We expect that a customized key rotation mechanism works similarly. It creates a *keys* array with several STEKs and the *keys* array is updated regularly to rotate the keys.

[4]https://www.openssl.org/docs/man1.1.1/man3/SSL_CTX_set_ex_data.html

```

static int ngx_ssl_session_ticket_key_callback(...)
{
  keys = SSL_CTX_get_ex_data(ssl_ctx, ngx_ssl_session_ticket_keys_index);
   ...
   if (enc == 1) {   /* encrypt session ticket */

     if (key[0]. size == 48) {
       cipher = EVP_aes_128_cbc();
       size = 16;
     } else {
       cipher = EVP_aes_256_cbc();
       size = 32;
     }

     ...
     if (EVP_EncryptInit_ex(ectx, cipher, NULL, key[0].aes_key, iv) != 1) {
       ngx_ssl_error(NGX_LOG_ALERT, c->log, 0,
       "EVP_EncryptInit_ex() failed");
       return -1;
     }

     if (HMAC_Init_ex(hctx, key[0].hmac_key, size, digest, NULL) != 1) {
       ngx_ssl_error(NGX_LOG_ALERT, c->log, 0, "HMAC_Init_ex() failed");
       return -1;
     }

     ngx_memcpy(name, key[0].name, 16);
     return 1;
   }
   ...
}

```

Listing A.10: *ngx_ssl_session_ticket_key_callback* function in *ngx_event_openssl.c* implements the OpenSSL callback function to customize session ticket handling

Next, we look at the implemented callback function for session tickets in Nginx (see Listing A.10). The callback function is called before session tickets are en- or decrypted in OpenSSL. It initializes several parameters that are needed for session tickets as for example the encryption algorithm with the current STEK. We explain how the function works if a session ticket has to be encrypted. We use the first STEK stored in the *keys* array to encrypt the session ticket. Depending on the size stored in the STEK structure either *AES-128-CBC* or *AES-256-CBC* is selected as the encryption cipher. As we have seen in the STEK initialization from the key files, the size should be either 48 or 80 bytes. However, in the else case it is not explicitly checked if the *size* is 80 bytes. By default for every *size* value different than 48 *AES-256-CBC* is selected. We would prefer that the function returns an error if a different size than 48 or 80 bytes is selected because likely the field is misused. Then, the encryption cipher is initialized along with the IV, the *aes_key* and the HMAC algorithm with the *hmac_key*.

```
ngx_ssl_session_ticket_key_t *key = malloc(sizeof(ngx_ssl_session_ticket_key_t));

//Scenario 1
RAND_bytes((unsigned char *) key->name, 16);
RAND_bytes((unsigned char *) key->aesKey, 16);
RAND_bytes((unsigned char *) key->hmacKey, 16);

//Scenario 2
RAND_bytes((unsigned char *) key, 48);

//Scenario 3
RAND_bytes((unsigned char *) key->name, 48);
```

Listing A.11: Three possible scenarios how STEK data structure may be initialized

In the following, we describe what may have caused the vulnerability. We look at three typical scenarios how the STEK structure *ngx_ssl_session_ticket_key_t* may be initialized before the code change (see Listing A.11). We use the *RandBytes* function of OpenSSL to initialize the struct members with random bytes. In the first scenario, we initialize each member of the structure individually with respectively 16 random bytes. For scenarios 2 and 3, we can use the fact that struct fields are stored in the memory one after the other. Thus, we can initialize the whole struct in one step with random bytes. In Scenario 2, we input the memory address of the *key* struct and our struct size of 48 bytes into the *RandBytes* function. OpenLiteSpeed is exactly doing that to initialize their STEKs. In Scenario 3, we can also enter the memory address of the first member *name* and initialize the whole struct with random bytes. All three scenarios lead to the same result as shown in Figure A.12 that all members are correctly initialized with random bytes.

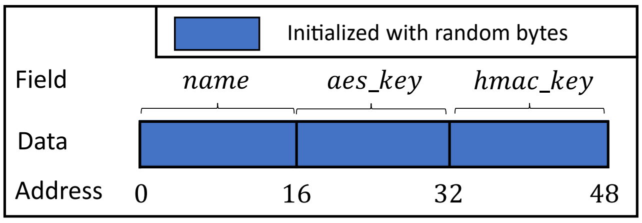


Figure A.12: Resulting Values for different initialization scenarios of the STEK struct *ngx_ssl_session_ticket_key_t*

Now, we look at what happens if the exact same code is used to initialize the *key* struct after the fields in the STEK data structure were adapted (see Listing A.8). This represents the scenario if AWS did not adapt its key rotation mechanism to the introduced code change in Nginx. We verified our findings with a small C project where we rebuilt the old and the updated STEK data structure from Nginx. Then, we initialized them with the three presented scenarios and evaluated the resulting values.

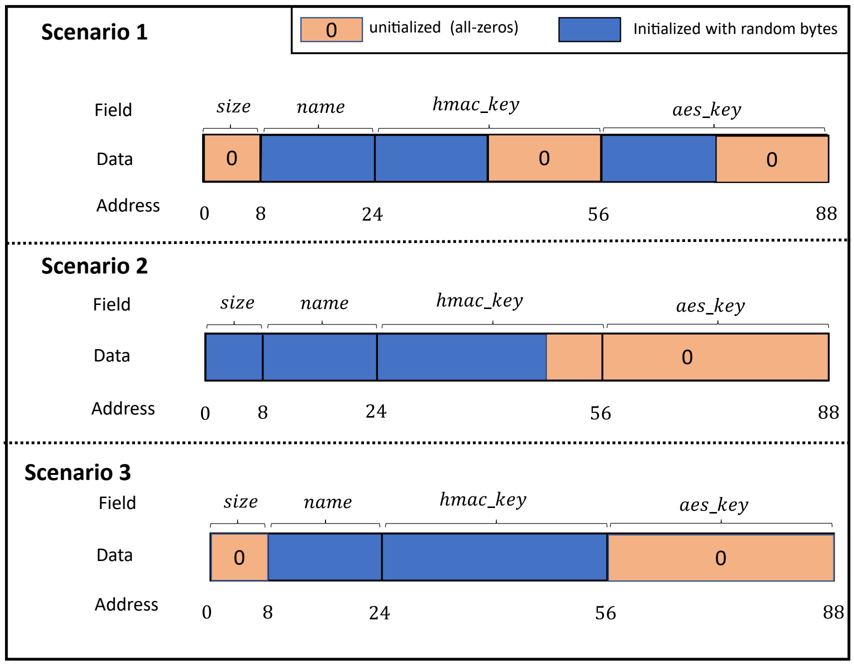


Figure A.13: Resulting Values for different initialization scenarios of the STEK struct *ngx_ssl_session_ticket_key_t*

In Scenario 1, all members are initialized with 16 random bytes. Both key values are only half initialized (see Figure A.13). However, the *size* field is not initialized. In the second scenario, only the first 48 bytes of the struct are initialized. Now, the *size* value is accidentally initialized with random bytes. The *name* and the first 24 bytes of the *hmac_key* are initialized with random bytes as well. However, the last 8 bytes of the *hmac_key* and the entire *aes_key* remain uninitialized. In the third scenario, the two members *size* and *aes_key* remain uninitialized. In Scenario 2 and 3, the *aes_key* is not initialized and thus the session tickets will be encrypted with an all-zero key. The *size* field does not contain 48 in all scenarios and thus *AES-256-CBC* always will be used to encrypt the tickets. We see that the missing *size* check (see Listing A.10) does not detect the misuse of the *size* field.

We can conclude that the presented combination of the struct initialization and the code change may have caused the security vulnerability at AWS hosts. However, we can not explain why the vulnerability only sometimes occurred.

Bibliography

[1] *Resolved: Application load balancer session ticket issue.* https://aws.amazon.com/security/security-bulletins/AWS-2021-002/, visited on August 5, 2021.

[2] *Usage statistics of web servers.* https://w3techs.com/technologies/", visited on August 5, 2021.

[3] *Advanced Encryption Standard (AES).* National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.

[4] N.J. AlFardan and K.G. Paterson: *Lucky thirteen: Breaking the TLS and DTLS record protocols.* pp. 526–540, 2013.

[5] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic: *Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS.* Cryptology ePrint Archive, Report 2016/475, 2016. http://eprint.iacr.org/2016/475.

[6] David Ziemann: *Analysis of the gnutls session ticket bug (cve-2020-13777).* https://www.hackmanit.de/de/blog/118-analysis-of-the-gnutls-session-ticket-bug-cve-2020-13777, visited on August 5, 2021.

[7] T. Dierks and E. Rescorla: *The Transport Layer Security (TLS) Protocol Version 1.2.* RFC 5246 (Proposed Standard), Aug. 2008. ISSN 2070-1721. https://www.rfc-editor.org/rfc/rfc5246.txt, Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447.

[8] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, *et al.*: *The matter of heartbleed.* In *Proceedings of the 2014 conference on internet measurement conference*, pp. 475–488, 2014.

[9] Filippo Valsorda : *Ticketbleed (cve-2016-9244).* https://filippo.io/ticketbleed/, visited on August 5, 2021.

[10] Filippo Valsorda: *We need to talk about session tickets.* https://blog.filippo.io/we-need-to-talk-about-session-tickets, visited on August 5, 2021.

[11] Fiona Klute: *Cve-2020-13777: Tls 1.3 session resumption works without master key, allowing mitm.* https://gitlab.com/gnutls/gnutls/-/issues/1011, visited on August 5, 2021.

[12] R. Housley: *Cryptographic Message Syntax (CMS).* RFC 5652 (Internet Standard), Sept. 2009. ISSN 2070-1721. https://www.rfc-editor.org/rfc/rfc5652.txt.

[13] B. Kaliski: *Pkcs# 7: Cryptographic message syntax version 1.5*, 1998.

[14] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen: *Tranco: A research-oriented top sites ranking hardened against manipulation.* In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, Feb. 2019.

[15] M. Liskij: *Masterthesis: Survey of tls 1.3 usage.* https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/MihaelLiskij_Thesis.pdf, visited on August 5, 2021.

[16] R. Merget, J. Somorovsky, N. Aviram, C. Young, J. Fliegenschmidt, J. Schwenk, and Y. Shavitt: *Scalable scanning and automatic classification of TLS padding oracle vulnerabilities.* pp. 1029–1046, 2019.

[17] Nick Sullivan: *A detailed look at rfc 8446 (a.k.a. tls 1.3).* https://blog.cloudflare.com/rfc-8446-aka-tls-1-3, visited on August 5, 2021.

[18] Nick Sullivan: *Introducing zero round trip time resumption (0-rtt).* https://blog.cloudflare.com/introducing-0-rtt/, visited on August 5, 2021.

[19] Nick Sullivan: *Tls session resumption: Full-speed and secure.* https://blog.cloudflare.com/tls-session-resumption-full-speed-and-secure, visited on August 5, 2021.

[20] E. Rescorla: *The Transport Layer Security (TLS) Protocol Version 1.3.* RFC 8446 (Proposed Standard), Aug. 2018. ISSN 2070-1721. https://www.rfc-editor.org/rfc/rfc8446.txt.

[21] Richard Fussenegger: *nginx session ticket key rotation.* https://github.com/Fleshgrinder/nginx-session-ticket-key-rotation, visited on August 5, 2021.

[22] J. Salowey, A. Choudhury, and D. McGrew: *AES Galois Counter Mode (GCM) Cipher Suites for TLS.* RFC 5288 (Proposed Standard), Aug. 2008. ISSN 2070-1721. https://www.rfc-editor.org/rfc/rfc5288.txt.

[23] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig: *Transport Layer Security (TLS) Session Resumption without Server-Side State.* RFC 5077 (Proposed Standard), Jan. 2008. ISSN 2070-1721. https://www.rfc-editor.org/rfc/rfc5077.txt, Obsoleted by RFC 8446, updated by RFC 8447.

[24] D. Springall, Z. Durumeric, and J.A. Halderman: *Measuring the security harm of tls crypto shortcuts.* In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, p. 33–47, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450345262. `https://doi.org/10.1145/2987443.2987480`.

[25] E. Sy, C. Burkert, H. Federrath, and M. Fischer: *Tracking Users across the Web via TLS Session Resumption.* In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, p. 289–299, New York, NY, USA, 2018. Association for Computing Machinery, ISBN 9781450365697. `https://doi.org/10.1145/3274694.3274708`.

[26] E. Sy, C. Burkert, H. Federrath, and M. Fischer: *Tracking users across the web via tls session resumption.* In *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 289–299, 2018.

[27] Tim Taubert: *Botching forward secrecy - the sad state of server-side tls session resumption implementations.* `https://timtaubert.de/blog/2014/11/the-sad-state-of-server-side-tls-session-resumption-implementations/`, visited on August 5, 2021.

[28] S. Vaudenay: *Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS...* pp. 534–546, 2002.

[29] D. Whiting, R. Housley, and N. Ferguson: *Counter with CBC-MAC (CCM).* RFC 3610 (Informational), Sept. 2003. ISSN 2070-1721. `https://www.rfc-editor.org/rfc/rfc3610.txt`.

List of Figures

List of Tables

List of Algorithms

List of Listings

B Java Code