We *Really* Need to Talk About Session Tickets: A Large-Scale Analysis of Cryptographic Dangers with TLS Session Tickets

Sven Hebrok, *Paderborn University;* Simon Nachtigall, *Paderborn University and achelos GmbH;* Marcel Maehren and Nurullah Erinola, *Ruhr University Bochum;* Robert Merget, *Technology Innovation Institute and Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum*

https://www.usenix.org/conference/usenixsecurity23/presentation/hebrok

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

We *Really* Need to Talk About Session Tickets: A Large-Scale Analysis of Cryptographic Dangers with TLS Session Tickets

Sven Hebrok[1], Simon Nachtigall[1,3], Marcel Maehren[2], Nurullah Erinola[2], Robert Merget[4,2], Juraj Somorovsky[1], and Jörg Schwenk[2]

[1]Paderborn University
[2]Ruhr University Bochum
[3]achelos GmbH
[4]Technology Innovation Institute

Abstract

Session tickets improve the performance of the TLS protocol. They allow abbreviating the handshake by using secrets from a previous session. To this end, the server encrypts the secrets using a *Session Ticket Encryption Key* (STEK) only know to the server, which the client stores as a ticket and sends back upon resumption. The standard leaves details such as data formats, encryption algorithms, and key management to the server implementation.

TLS session tickets have been criticized by security experts, for undermining the security guarantees of TLS. An adversary, who can guess or compromise the STEK, can passively record and decrypt TLS sessions and may impersonate the server. Thus, weak *implementations* of this mechanism may completely undermine TLS security guarantees.

We performed the first systematic large-scale analysis of the cryptographic pitfalls of session ticket implementations. (1) We determined the data formats and cryptographic algorithms used by 12 open-source implementations and designed online and offline tests to identify vulnerable implementations. (2) We performed several large-scale scans and collected session tickets for extended offline analyses.

We found significant differences in session ticket implementations and critical security issues in the analyzed servers. Vulnerable servers used weak keys or repeating keystreams in the used tickets, allowing for session ticket decryption. Among others, our analysis revealed a widespread implementation flaw within the Amazon AWS ecosystem that allowed for passive traffic decryption for at least 1.9% of the Tranco Top 100k servers.

1 Introduction

TLS (Transport Layer Security) [43] is one of the most frequently used cryptographic protocols to ensure secure communication on the Internet. It guarantees the security of different application layer protocols, including email communication and web services over HTTP. Due to the diversity of applications and the importance of TLS in general, we have seen significant developments in the area of TLS in the last three decades. This led to the development of new protocol versions and protocol extensions. Today, TLS 1.2 [17] and TLS 1.3 [43] are mostly used.

To set up a secure connection, both communication parties need to agree on a TLS version, a set of cryptographic algorithms, and they need to establish a shared secret. This is achieved by the so-called *TLS handshake*. To establish a common secret in an initial full TLS handshake (see Figure 1), the client and server use public-key algorithms, for example, the Diffie-Hellman key exchange over elliptic curves (ECDHE). There are two main drawbacks of the full TLS handshake. First, it includes computationally expensive public-key operations for both parties. Second, the full handshake requires two round trips in TLS 1.2 and one round trip in TLS 1.3; this causes a significant delay before encrypted application data can be sent.

TLS Session Resumption TLS session resumption is a mechanism with the primary goal of reducing latency in TLS connections. By performing a resumption handshake (see Figure 2), both communication parties rely on the secret state negotiated in a previous handshake which allows them to omit computationally expensive public-key cryptographic operations and reduce the number of round trips. A blog post from Cloudflare [51] reports that a resumption can be done in half the time of a full handshake with only about 4% of the CPU load compared to a full handshake. Session resumption is thus especially useful for servers under high load and clients running on low-power devices. TLS session resumption can be implemented using *session IDs*, where both server and client must store the secret state, or with *session tickets*, where storage is outsourced to the client. In this paper, we only consider the latter method.

Session Tickets TLS session tickets [45] provide a mechanism to implement session resumption without requiring an

The title is a reference to Filippo Valsorda's Blog post "We need to Talk About Session Tickets" [59].

additional database on the server. According to a study from 2018, 78% of the Alexa Top Million TLS-enabled websites supported session tickets [53]. To use session resumption with session tickets, both parties have to support it. The server generates a *Session Ticket Encryption Key* (STEK), which contains a set of symmetric keys only known to this server. It can then use the STEK to encrypt the secret TLS session state and insert the ciphertext into a session ticket which is sent to the client during the (initial) handshake. The client stores the secret TLS session state and the (public) session ticket. In the next TLS handshake, the client returns the session ticket to the server. The server can decrypt it using the STEK and thus retrieve the secret session state. Now both parties have the previous session state and can resume the session.

Security of TLS Session Tickets While TLS session tickets bring significant performance improvements for TLS connections [51], they have become a major target of criticism raised by security experts [48, 53, 59]; if an attacker can retrieve the STEK, they can impersonate the server or decrypt recorded TLS connections. Such dangers are not only theoretical; in 2020, Fiona Klute discovered a vulnerability affecting the security of the session resumption mechanism in GnuTLS [35, 62]. The server used an all-zero STEK in the initial key rotation interval, allowing an attacker to decrypt the session tickets and learn the included secret TLS state.

The security impact of a STEK compromise depends on the TLS version used.

- In TLS 1.2 and before, session tickets contain the *master secret* used to derive keys for the initial session, where the ticket was issued, and the resumed session, where the ticket is redeemed. If an attacker compromises the STEK, they can *passively* decrypt the initial and all resumed sessions, thus effectively breaking *confidentiality* and *forward secrecy* promises. Since session tickets are transferred before the TLS channel encryption is established, the attacker does not even need to wait for the session to be resumed; they can break the initial TLS session independently of the used key exchange algorithm.
- In TLS 1.3, session tickets contain the *resumption secret*, a secret derived from the previous master secret. Based on the resumption secret, the parties can derive an early secret to protect the *early application data* sent in the first round trip of the session resumption handshake. Obtaining the resumption secret always enables an attacker to decrypt this early application data. The security properties of regular application data are tied to new secrets derived for the new session. However, depending on the specific resumption mode used, a passive attacker able to eavesdrop on the new session may still be able to decrypt the new session. When resuming a session, the client and server can solely rely on the previously established key (`psk_ke` mode) or perform a new Diffie-Hellman key exchange (psk_dhe_ke) that influences the derived keys. Only the latter achieves forward secrecy for new application data but also negates some of the performance benefits of session resumption.

Independently of the used TLS version, an active attacker can use a compromised STEK to impersonate the server.

Systematic Evaluation Despite the criticism expressed by security experts [48, 53, 59] and the known negative effects of incorrect session ticket implementations on TLS security [35, 62], no systematic large-scale analysis of the dangers associated with bad implementations of session tickets exists. We structured the first such analysis along the following research questions:

> **RQ1**: Which cryptographic vulnerabilities may be introduced through faulty TLS session ticket implementations?

Klute showed a major implementation bug in the STEK generation used by GnuTLS [35, 62]. We systematically extend this test case to use it in our offline tests later. The flexibility of the session ticket standard [45] allows for many other implementation flaws. In Section 3, we describe potential implementation flaws, for example, unencrypted session tickets, reused keystreams, and the usage of weak cryptographic algorithms. These potential flaws are later evaluated in online and offline tests.

Since some of the discussed flaws are only relevant in specific configurations (e.g., keystream reuse only in connection with stream ciphers or counter modes), we needed to get an overview of the implementations of the session ticket mechanism in open-source libraries:

> **RQ2**: How do state-of-the-art open-source libraries implement session tickets?

Section 4 presents our analysis of 12 open-source TLS libraries and their implementation of TLS session tickets. While we found all libraries to be generally secure, our analysis gave us an impression of real-world implementations of session tickets. We observed that none of the libraries completely adhere to the standard recommendations [45]. Every library deviates in at least one property, be it a variable in the session ticket format, the encryption algorithm, or the authentication algorithm. This knowledge allowed us to design precise online and offline tests for session ticket implementation bugs for large-scale analyses of session ticket usage in the deployed TLS ecosystem (Section 5).

> **RQ3**: Are real-world TLS servers vulnerable to the proposed attacks?

To answer this question, we performed three sets of scans against the hosts in the Tranco list [38] and publicly available IPv4 hosts (Section 6). Our scans and the related online and offline tests uncovered that over 1.9% of the Tranco top 100k

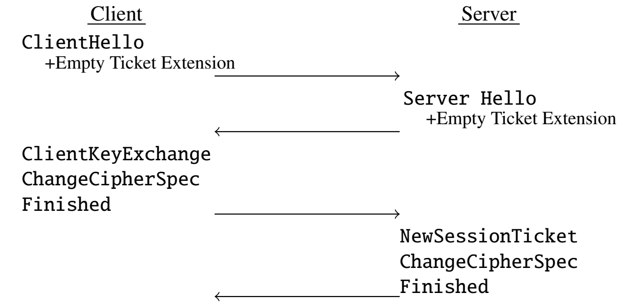

Figure 1: A TLS 1.2 handshake using a Diffie-Hellman key exchange and session ticket negotiation.

occasionally used an empty STEK to encrypt their session tickets (Section 6.2.1). The flaw could allow an attacker to passively decrypt TLS traffic as long as session tickets were used. It was caused by a faulty key rotation mechanism in AWS's ALBs (Application Load Balancers). In further scans, we discovered vulnerabilities caused by partially initialized keys (Section 6.2.2) and a reused keystream (Section 6.2.3). These allowed passive adversaries to decrypt the session tickets.

Contributions We make the following contributions:

- We systematically analyze cryptographic implementation pitfalls in the session ticket handling of TLS servers and their impact on TLS connections (Section 3).
- We conduct a source code analysis of 12 open-source systems and study their usage of cryptographic algorithms, session ticket formats, and vulnerability mitigations (Section 4).
- Based on our analyses, we implement online and offline tests for the potential vulnerabilities (Section 5) and perform large-scale security scans of the TLS ecosystem (Section 6).
- Our scans detect critical security issues in the collected session tickets, including repeating keystreams and weak keys, allowing to decrypt the TLS communication (Section 6.2.1). Most notably, we discovered an issue in a large portion of AWS hosts, affecting around 1.9% of Tranco top 100k hosts.

2 Background on TLS

The Transport Layer Security (TLS) protocol allows two communicating peers (client and server) to establish a secure channel. It is split into two phases: the handshake and the transmission of application data. The handshake establishes keys and parameters to secure the application data. In this paper, we focus on the handshake. Throughout TLS versions 1.0 to 1.2, the general protocol structure, especially the handshake, remained very similar. For better readability, we subsequently use TLS 1.2 in place of this version range.

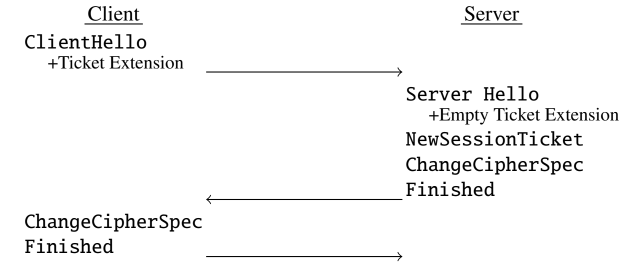

Figure 2: A TLS 1.2 resumption handshake using session tickets

As shown in Figure 1, the client initiates the TLS handshake with a `ClientHello` message that enumerates supported features, such as the protocol version and cryptographic algorithms. Some of these features are expressed through *Extensions*, such as the *SessionTicketExtension*, which requests a session ticket from the server. The server chooses suitable parameters from the offered features and communicates its choices through a `ServerHello` message. If a client requests a session ticket through its `ClientHello`, this message may also contain an empty *SessionTicketExtension* if the server is willing to issue a ticket. In TLS 1.2, the server continues to send a `Certificate` message followed by a `ServerKeyExchange` message containing an ephemeral public key if Diffie-Hellman key exchange was negotiated. The server ends its flight of messages with a `ServerHelloDone` message. The client replies with a `ClientKeyExchange` message containing its chosen key material, such as a Diffie-Hellman key share. At this point, both parties can derive all required symmetric session keys. Subsequently, the client sends a `ChangeCipherSpec` message which informs the server that all following messages will be protected using the established keys and algorithms. The client finishes its flight of messages with a `Finished` message, which provides a cryptographic checksum of the handshake transcript. The server concludes the handshake by sending a `ChangeCipherSpec` and `Finished` message of its own. If the server is willing to issue a requested session ticket, a `NewSessionTicket` message is sent before the `ChangeCipherSpec`. Note that the `NewSessionTicket` message is hence left unencrypted in TLS 1.2.

In TLS 1.3, the handshake was modified significantly to reduce the round trip times (RTT) required before exchanging application data. To achieve this, both peers already send public keys in their respective *Hello* messages. As a result, session keys can be derived after processing the `ServerHello` message, and all following handshake messages can be encrypted. Consequently, in TLS 1.3, a `NewSessionTicket` message, sent either during or after the handshake, is always protected using the derived session keys.

To eliminate any delay before the exchange of application data entirely, TLS 1.3 also provides the option to send encrypted application data along with the `ClientHello`. This application data is referred to as *early data*. In order to encrypt the early data, a client must possess a PSK that was either configured manually or established in a previous session. We expand upon the security properties of early data in Section 2.1.1.

Session Secrets TLS uses a variety of keys for cryptographic operations. These keys are derived from so-called *secrets* established during the handshake. The keys and secrets are derived using hash-based functions. We give an overview in Figure 3. In TLS 1.2, these are the premaster secret and master secret. The premaster secret is established using the key exchange.

TLS 1.3 replaced the premaster secret with the handshake secret. TLS 1.3 further introduces the early secret, resumption secret, and Pre-Shared Key (PSK)s. We give more details on the different secrets and how they are affected by session tickets in Section 3.1.

2.1 Session Resumption

Session resumption allows two parties to reuse previously established keys and parameters (henceforth just called *state*) in a new TLS session. The handshake of this resumed session does not require a new key exchange and no authentication of the peers, which eliminates the need for asymmetric cryptography. As asymmetric cryptography accounts for a large part of the handshake's latency and computational costs, a resumption handshake benefits both the client and server. Session resumption in TLS can be classified into two concepts: Session IDs and session tickets.

When using session IDs, the client and server remember the session's state, and the server sends an arbitrary ID to the client. Upon resumption, the client reuses the state and sends the ID to the server. The server uses the ID to look up the state from the earlier session. To communicate that the resumption request was accepted, the server likewise includes the ID in its `ServerHello`.

ID-based resumption may require the server to store large numbers of states. To alleviate this, session tickets have been specified [45]. Here, the server puts the state into a *session ticket* structure which it encrypts and sends to the client. The client stores the ticket for the server alongside the session state. Upon resumption, the client loads the state and sends the ticket to the server (cf. Figure 2). The server can then decrypt the ticket and extract the state from the ticket so that both parties are now in possession of the previous session's state. For this to work, it is sufficient if only the server understands the ticket format, as the client can handle tickets opaquely. Further, it is crucial that only the server is able to extract the state from the session ticket, as the same key is typically used for all clients. The key used to encrypt the session state is called a *Session Ticket Encryption Key* (STEK). We also consider the key used to authenticate the ticket to be part of the STEK.

Regardless of the chosen mechanism, the server can always reject the client's resumption request and fall back to a full handshake by sending the messages depicted in Figure 1.

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

Listing 1: Recommended session ticket format in RFC 5077. The RFC recommends that session tickets are encrypted with AES-128-CBC and protected by HMAC-SHA-256 over all previous fields.

Recommended Ticket Structure The usage of tickets was initially defined in RFC 4507 [44], which was shortly after updated by RFC 5077 [45]. The RFC further recommends a structure for session tickets, which is shown in Listing 1, consisting of four values:

`key_name`: A 16-Byte STEK identifier that the server can use to select the key to decrypt the session state. It is recommended to generate this identifier randomly for each key.

`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 to resume the session. This state is encrypted since only the server is authorized to read the contents of the session state. The length of the state (2 bytes) should be included before the actual state.

`mac`: Session tickets shoud be integrity protected. The Message Authentication Code (MAC) is computed over all previously named fields.

RFC 5077 recommends using AES-128 CBC as the encryption algorithm and HMAC-SHA-256 as the authentication algorithm. Thus, the server's STEK must consist of two keys, one for each algorithm.

Additionally, the RFC recommends a list of parameters to include in the plaintext state. These are the selected protocol version, cipher suite, and compression method along with the master secret. Besides these, it is recommended to include a timestamp to recognize expired tickets.

Note that servers do not need to comply with these RFC recommendations; they can use their own ticket structure or different cryptographic algorithms. Since the tickets are only processed by the servers and handled by clients opaquely, such RFC deviations do not cause interoperability issues.

Security Properties The implementation of session resumption with session tickets in TLS has significant consequences for the security of TLS sessions [59]. Since the

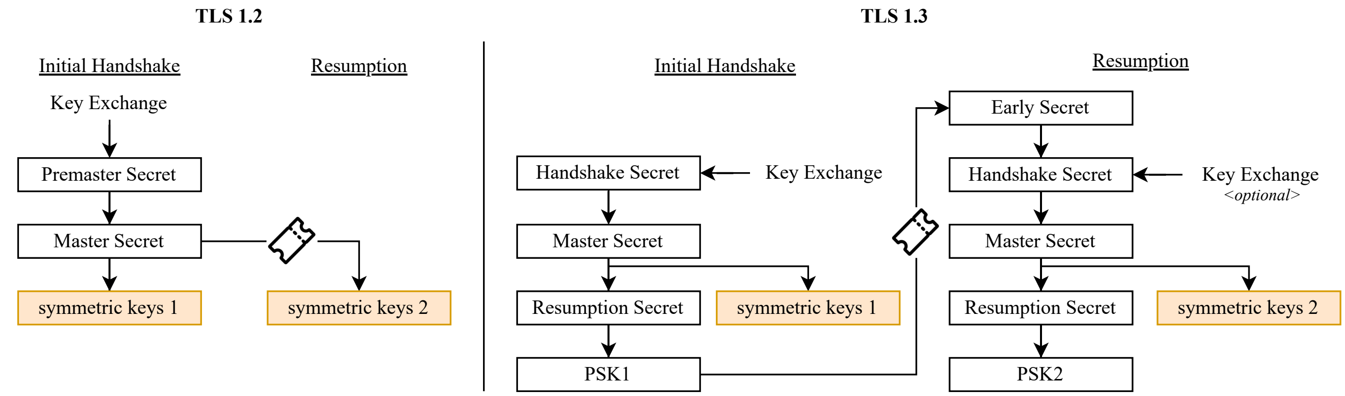

Figure 3: Overview of the key generation in TLS. The solid arrows denote a hash-based one way function. This function also takes other data as input which we omit for clarity here (e.g., random numbers from the *Hello* Messages). Upon resumption in TLS 1.2, the same master secret is reused to derive the symmetric keys. In contrast, in TLS 1.3, the PSK is used to compute the early secret which is only used to protect early application data.

session tickets contain the master secret for deriving the session keys, it is crucial that the STEK is only accessible to 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. A compromised STEK has two fatal consequences for sessions in TLS 1.2:

1. Since resumed sessions rely solely on the encrypted master secret, they never achieve forward secrecy. After obtaining the STEK, an attacker can extract and decrypt the session ticket from a recorded session before decrypting the resumed session itself.

2. Obtaining the master secret within a session ticket also enables an attacker to decrypt the previous session in which this master secret was established.

In order to mitigate the effects of a STEK compromise on forward secrecy, the RFC recommends rotating the STEK every 24 hours.

2.1.1 Session Resumption in TLS 1.3

TLS 1.3 unified the concepts of session IDs and session tickets into the notion of Pre-Shared Keys (PSK). Both, IDs and tickets, are now sent through the `NewSessionTicket` message as a ticket. The client is not able to tell if this ticket is just a reference to a PSK stored in the server's database or a session state encrypted with the server's STEK. Only the length of the ticket may hint at which mechanism is used. In contrast to TLS 1.2, the `NewSessionTicket` messages are always sent encrypted using the negotiated session keys.

Recommended Ticket Structure TLS 1.3 does not recommend a ticket structure. RFC 8446 [43] only states that the session ticket has to be self-encrypted and self-authenticated by the server. However, in practice, most TLS implementations use the same ticket structures as for TLS 1.2, as we will show in Section 4.

Security Properties To address the negative security implications that session tickets present in TLS versions up to 1.2, TLS 1.3 introduced some significant changes. To ensure the forward secrecy of the previous session in which the session ticket was issued, TLS 1.3 does not reuse the same master secret. Instead, TLS 1.3 uses a secret *derived* from the master secret. An attacker cannot reconstruct the initial master secret so a compromise of the STEK no longer affects the initial session. TLS 1.3 further allows the communicating parties to optionally perform a new Diffie-Hellman key exchange upon resumption. While this negates some of the performance advantages of a session resumption, it ensures forward secrecy for the resumed session. A resumed session still does not re-authenticate the server, that is no certificate message is sent or verified, which still provides a speedup.

Both application messages sent with the `ClientHello` (early data) and all messages exchanged in a resumed session without the optional key exchange can still be decrypted by an attacker later if the contents of the session ticket get compromised.

Since `NewSessionTicket` messages are always encrypted in TLS 1.3, obtaining the session ticket is only possible for an attacker when the client requests the session resumption by including the ticket in its `ClientHello`.

2.2 TLS-Attacker

TLS-Attacker [1, 47] is a well-established framework for systematic analyses of TLS implementations. It implements TLS while allowing to generate arbitrary protocol flows and make very granular modifications to the exchanged messages. This way, we can easily access the tickets received by a server and prepare modified tickets to send to the server. TLS-Scanner [2] builds on TLS-Attacker. It contains a multitude of tests that are run automatically against a specified server. It provides a report that allows to determine supported TLS features and potential issues quickly. To perform large-scale scans, TLS-Crawler [42] can be used. It is a framework that utilizes TLS-Scanner to scan many servers in parallel and writes the results to a database. Moreover, it allows for distributing the scan tasks across multiple machines.

3 Cryptographic Pitfalls in Session Ticket Implementations

As mentioned in Section 2.1, the session ticket format is not strict. Since the session ticket is additionally encrypted with a key only known to the server, the connecting client is unable to see if the server is using the recommended format. A server deviating from the recommended ticket design or a server making an implementation flaw would hence not be apparent to a client but could have severe consequences for the security of the connection. This section discusses potential deviations from the recommended format, their consequences, and potential cryptographic implementation pitfalls. All of these flaws allow an attacker to recover the secrets, for example, the master secret, from the ticket.

Unencrypted Session Tickets The first possible vulnerability can appear if wrongly configured servers issue session tickets with an unencrypted session state. Such configuration mistakes would allow attackers to access the session secrets directly. While the vulnerability seems contrived, the session ticket mechanism would work perfectly fine with unencrypted session states. The session ticket structure is opaque for the client, and it would not notice if connection secrets would be transmitted within it. Similar issues have been found in the context of S/MIME and OpenPGP [24, 46]; they led to sending plaintext emails even when encryption was turned on [24, 46].

Weak Encryption Keys Inspired by the bug in GnuTLS [35], another vulnerability can appear if a vulnerable server uses default keys or keys with low entropy to protect its session tickets. This vulnerability might be caused by uninitialized memory and lead, for example, to session tickets encrypted with zero-keys. This enables an attacker to decrypt tickets and extract the session secrets.

Reused Keystream If implementations deviate from the proposed encryption algorithms, algorithm-specific issues may appear. An example of this is keystream reuse in ciphers like ChaCha20, or cipher modes like GCM, CCM or CTR. This mistake was previously observed in some TLS servers using AES-GCM [12]. In these cipher modes, a keystream is generated and XORed onto the plaintext to compute the ciphertext. Concretely, with a keystream K_s, and a plaintext P, the ciphertext C is computed as $K_s \oplus P = C$. If two different plaintexts P_i and P_j use the same keystream, assuming the attacker is in possession of P_j, they can compute the other plaintext as $P_i = C_i \oplus C_j \oplus P_j$. An attacker can abuse a server that occasionally encrypts session tickets with repeating keystreams by frequently requesting session tickets until a session ticket with a keystream used in a victim's session ticket has been received. The attacker can verify the attack's success by checking whether the decrypted master secret is suitable to decrypt the victim session.

Cryptographic Wear-out Depending on the selected algorithms and the number of session tickets expected to be issued with a given STEK, a problem called cryptographic wear-out can appear. Typically, ciphers have a threshold of ciphertext that can be generated under a given key until they become insecure. For example, for AES-GCM, the nonce should never be repeated as using the same nonce under the same key with different plaintexts leaks the XOR of the plaintexts and the authentication key [12]. The nonces in AES-GCM are 12 bytes long, meaning that randomly chosen IVs will collide with 50% probability after 2^{48} session tickets. Since 50% is typically not conservative to be considered secure, real-world systems additionally require a security margin. With an acceptable collision risk of $1/2^{32}$, AES-GCM allows for the encryption of 2^{32} (≈ 4.2 billion) session tickets. This limitation has also been set by the National Institute of Standards and Technology (NIST) [23]. Depending on the longevity of the STEK and the server infrastructure, it is possible that this limit gets exhausted, which can result in a nonce collision, allowing the users of affected sessions to decrypt the other colliding session. For AES-CBC with HMAC, cryptographic wear-out is not of practical concern as the block and the initialization vector size are both 16 bytes, allowing for the encryption of 2^{48} session tickets before the threshold is passed, which is unlikely to be ever hit by a real server, even if the STEK is never rotated.

Nonce collisions are not the only threat one needs to consider when using authenticated encryption schemes. Other security properties, like integrity or indistinguishability, also degrade with increasing data sizes and rotated encryption keys [9, 16, 31, 39]. However, in the case of the considered schemes in this paper, the practical importance of this degradation on session tickets is limited.

Broken Authenticated Encryption Similar to the encryption scheme, an implementation may use different algorithms to protect the integrity and authenticity of its session tickets. An implementation may even decide not to use authenticated encryption or no authenticity and integrity protection at all. This could, for example, be the omission of the HMAC or its validation. Even if authentication is enforced, similarly to the default STEKs, servers might use weak HMAC keys. This would allow an attacker to re-compute correct HMACs over modified session tickets, rendering authenticated encryption useless.

Broken authenticated encryption does not directly lead to session ticket decryption. However, it can open doors for new potential attacks. For example, RFC 5077 [45] recommends the usage of AES-CBC for session ticket encryption along with HMACs, in the Encrypt-then-MAC scheme. Broken authentication might allow the attacker to modify the ciphertexts and make the implementation vulnerable to a padding oracle attack [4, 47, 60]. The padding oracle attack exploits the malleability of the Cipher Block Chaining (CBC) mode of operation and strict padding validation by the receiving application. It allows the attacker to query the server with carefully modified ciphertexts. If the server leaks information about the padding validity, the attacker can decrypt the plaintext and thus the session ticket after a few hundred queries.

Note that even if authentication with HMACs is enforced, a server might still reveal a padding oracle vulnerability when using MACs in an insecure way. For example, servers can use the MAC-then-Encrypt scheme, whose application makes secure padding oracle prevention very challenging [4, 42, 47]. The same scheme is used in TLS-CBC, making it tempting for implementations to reuse existing code.

Weak Algorithms If the server uses a weak algorithm, different attacks to recover the plaintext may be possible. For example, the algorithm could use short keys (e.g., 56 bytes in the case of DES), allowing an attacker to brute-force the STEK (cf. weak keys). Other attacks might be feasible depending on the algorithm.

Decryption Side Channels Depending on the structure of the session ticket, an implementation may be vulnerable to side-channel attacks, such as a timing side channel [4, 15, 54]. For example, a server could leak information about the ticket plaintext based on how fast it rejects a manipulated ticket and falls back to a full handshake. These attacks are hard to exploit for a remote attacker due to the noise introduced by network nodes.

3.1 Impact on the TLS Sessions

All presented pitfalls allow an attacker to decrypt the session ticket and, therefore, potentially retrieve the master secret. However, due to the design differences of session resumption in TLS 1.2 and TLS 1.3, the impact differs between these two versions. If the attacker gets possession of the session secrets stored in a TLS 1.2 session ticket, they can decrypt all sessions where the compromised session ticket was either issued or redeemed. In TLS 1.3, the attacker can decrypt the 0-RTT data of all sessions where the compromised session ticket was redeemed. If no additional key exchange is performed, the following application data can also be decrypted. The impact also depends on the secret contained in the ticket. If the ticket contains the master secret instead of the PSK, the session in which the ticket was issued can be decrypted.

For some of the presented attacks, a passive Mallory-in-the-Middle (MitM) attacker suffices to break the security of the TLS session and eavesdrop on the exchanged application data. An active attacker can impersonate the server when the client resumes the session, in all TLS versions.

If the attacker can also forge new tickets (for example, due to a *Session Ticket Encryption Key* (STEK) compromise), they can also set the client identity stored in the ticket. This may allow them to circumvent client authentication or impersonate another identity.

4 Analysis of Session Tickets in Open-Source TLS Libraries

To start our evaluation, we looked at the session ticket implementation in common open-source TLS libraries. We manually analyzed the source code of TLS libraries for their used session ticket format and the potential implementation flaws proposed in Section 3. In total, we evaluated nine different TLS libraries. We additionally evaluated three web servers. While these web servers also use libraries covered in our open-source analysis, they might behave differently due to how they utilize them. Our goal was to better understand how different servers handle session tickets and if they fulfill the recommendations of RFC 5077 [45]. We summarize the results in Table 1.

Encryption & Authentication Algorithms We first look at the encryption and authentication algorithms used by the different implementations. RFC 5077 recommends using *AES-128-CBC* to encrypt the tickets and *HMAC-SHA-256* to authenticate the tickets (including the encrypted contents). All tested libraries use authenticated encryption with strong algorithms. Note that GnuTLS uses *HMAC-SHA1*. While still secure, this is weaker than the recommended *HMAC-SHA256*. Most libraries diverge from the recommended algorithms and use different modes. Rustls is the only library not to use AES, and uses ChaCha20 instead. Further, half of the analyzed libraries use an AEAD algorithm instead of an HMAC algorithm for authentication. Apache and Nginx both use OpenSSL, but compared to OpenSSL's example server, Apache uses *AES-128-CBC* instead of *AES-256-CBC* for encryption and Nginx supports both options. OpenLiteSpeed uses BoringSSL by default with the same session tickets configuration as BoringSSL's example server.

Session Ticket Format Next, we evaluate the format of the session tickets and compare them to the recommended session ticket format specified in RFC 5077 (Listing 1). The web servers use the format of the underlying libraries. Most libraries follow the ticket format recommendation. However, apart from GnuTLS and mbedTLS, they do not include the size of the encrypted state within the ticket. The encrypted state is the only field with a variable size in the ticket; hence the format is still unambiguous. Most deviations are simple changes in the length of the IV or MAC because they use different algorithms.

We found two libraries that notably diverge from the recommendation: Rustls and Botan. Rustls does not include any `key_name` at all. Botan includes additional fields: An 8-byte `magic_constant` and a 16-byte `key_seed`. With just

		Session Ticket Format						Symmetric Algorithms	
Library	**Version**	`magic`[a]	`key_name`	`seed`[a]	`iv`[b]	`len`	`mac`	**Encryption**	**Authentication**
RFC 5077		**–**	**16**	**–**	**16**	**2**	**32**	**AES-128-CBC**	**HMAC-SHA256**
BoringSSL	2021[c]	**–**	**16**	**–**	**16**	–	**32**	**AES-128-CBC**	**HMAC-SHA256**
Botan	2.19.2	8	4	16	12	–	16	**AES**-256-GCM	(GMAC)
GnuTLS	3.7.6	**–**	**16**	**–**	**16**	**2**	20	**AES**-256-**CBC**	**HMAC**-SHA1
GoTls	go1.18.3	**–**	**16**	**–**	**16**	–	**32**	**AES-128**-CTR	**HMAC-SHA256**
MatrixSSL (TLS 1.2)	4.3.0	**–**	**16**	**–**	**16**	–	**32**	**AES**-256-**CBC**	**HMAC-SHA256**
MatrixSSL (TLS 1.3)	4.3.0	**–**	**16**	**–**	12	–	16	**AES**-256-GCM	(GMAC)
mbedTLS[d]	3.1.0	**–**	4	**–**	12	**2**	16	**AES-128**/256-GCM **AES-128**/256-CCM	(GMAC) (CBCMAC)
OpenSSL	3.0.3	**–**	**16**	**–**	**16**	–	**32**	**AES**-256-**CBC**	**HMAC-SHA256**
Rustls	0.20.6	**–**	–	**–**	12	–	16	ChaCha20	Poly1305
s2n	1.3.15	**–**	**16**	**–**	12	–	16	**AES**-256-GCM	(GMAC)
Apache	2.4.54	*Format of OpenSSL*						**AES-128-CBC**	**HMAC-SHA256**
Nginx	1.22.0	*Format of OpenSSL*						**AES-128**/256-**CBC**	**HMAC-SHA256**
OpenLiteSpeed	1.17.6	*Format of BoringSSL*						**AES-128-CBC**	**HMAC-SHA256**

a: These fields are only added by Botan.
b: IV or Nonce.
c: BoringSSL does not use releases. We analyzed the commit `dddb60e` from 2021-08-31.
d: mbedTLS can be configured to use different algorithms.

Table 1: Ticket format of evaluated TLS libraries and web servers. Similarities with the session ticket format according to RFC 5077 are highlighted in bold. All libraries used either AEAD algorithms or the Encrypt-then-MAC paradigm.

4 bytes, the key name is shorter than recommended. Additionally, Botan derives the key name from the STEK and a per-ticket key used for encryption from the seed and the STEK.

Cryptographic Wear-out We discovered that for some libraries, cryptographic wear-out is a potential danger. MatrixSSL in TLS 1.3 and s2n use AES-GCM with random IV's to encrypt their session ticket, meaning that these implementations should only encrypt up to 2^{32} session tickets under a given STEK. The situation is similar for Rustls, which uses ChaCha20 with a random IV to encrypt its session tickets. For the same reasons as with GCM, ChaCha20 should only be used to encrypt up to 2^{32} session tickets until the threshold is reached. If it is realistic to reach this limit depends on the key rotation policy and the load the used server is capable of handling.

We deem the other implementations to not suffer from wear-out. At first glance, Botan also seems to suffer from the issue, but a random seed is used to construct a short-term STEK that drastically reduces wear-out. The GoTls implementation uses AES-CTR with a random IV. This allows GoTls to encrypt up to 2^{48} blocks with the same key until the threshold is reached. Since session tickets are typically small, it is unlikely that the block limit is exceeded before key rotation happens.

5 Scanning and Evaluation Methodology

While in a lab setting only a limited number of servers can be examined, a much bigger range of servers can be evaluated in a real-world setting. Previous research has shown that real-world evaluations of TLS servers yield interesting new insights [8, 11, 14, 20, 21, 29, 36, 41, 42, 56, 57]. Since the session ticket format can be chosen arbitrarily by the server, the major challenge was to design black-box tests for a limited evaluation of session tickets suitable for large-scale studies on the TLS ecosystem.

Based on our results of the evaluation of open-source libraries, we implemented tests for a subset of the implementation pitfalls and vulnerabilities mentioned in Section 3 with TLS-Scanner. We divided the tests into two categories: online and offline tests. Online tests demand active modifications of session tickets during the scanning, while offline tests only require access to collected session tickets. Cryptographic wear-out cannot reasonably be evaluated with black-box tests and is therefore outside the scope of this analysis. Timing side-channel attacks have also been excluded as they require either very precise or many measurements for each host to determine accurate results.

5.1 Online Server Scans

Session Ticket Support For each tested TLS version, we start by testing whether a server issues and resumes tickets. For this, we perform an initial handshake and check whether it contained a ticket. If it contained a ticket, we perform ten additional handshakes to collect more tickets. Should the server not issue tickets in all connections, we consider it to only support tickets partially. In addition to the session tickets, we also store the corresponding secrets used to derive keys for each connection. For TLS 1.2, these are the premaster secret and master secret. For TLS 1.3, the secrets are the handshake secret, master secret, resumption secret, and PSKs. To test whether the ticket mechanism is functional, we also check whether the server accepts one issued ticket.

Missing Authenticity Protection To test the authentication and integrity algorithms, we perform tests where we manipulate the session ticket. As shown in our open-source analysis (cf. Table 1), every session ticket byte is protected using a

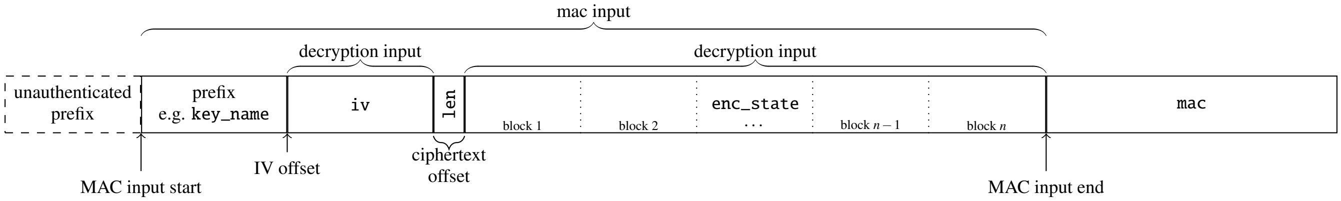

Figure 4: Assumed ticket format. The `enc_state` is split into blocks in case a block cipher is used. The correct inputs for the MAC and decryption are annotated.

MAC. Therefore, any session ticket modification must lead to ticket invalidation. To test whether this is correctly implemented, we induce bitflips in the ticket and send it to the server. For each byte in the ticket, we once induce a single bitflip and observe the server behavior. If the server accepts any ticket, we know it does not properly check the ticket authenticity.

Padding Oracle Attacks While TLS-Scanner provides tests for TLS padding oracles [60], these could not be applied to our use case. These tests assume that TLS-Scanner is in possession of symmetric encryption keys and thus can precisely encrypt the ciphertext, allowing it to include malformed padding on purpose. In our case, we do not know which padding is included, which block size is used, or where the ciphertext in the ticket is ending. Based on our analyses in Section 4, we assume servers use a format where the ticket contains the ciphertext and MAC as the last fields (cf. Figure 4).

To detect a CBC padding oracle vulnerability, we need to modify the second to last ciphertext block $n-1$. Based on the assumed format, we combine possible MAC lengths (0, 16, 20, 28, 32, 48, 64) with possible block lengths (8, 16) to get possible positions for block $n-1$. To get the last byte, we need to modify this block such that the last block n contains a valid 1-byte padding. To this end, we xor the last byte of block $n-1$ with each guessed plaintext p and a one byte padding. If we guessed correctly, the last block will contain a 1 byte padding and a vulnerable server behaves differently.

We consider two possible padding schemes: PKCS#7 [33] and padding as defined for block ciphers in TLS 1.2 [18]. To speed up our attack, we assume the last block to already contain padding. Therefore, we have less possible plaintexts to test.

We send each modified ticket to the server and observe its behavior. We observe differences in the message types, how messages are split into TLS records, and the state of the TCP socket. If a difference is observed, we confirm the vulnerability by guessing the second to last byte. For this byte, we do not assume a specific value but test all possible byte values. If the behavior is again different, we consider the server vulnerable.

We send each ticket multiple times to counteract independent behavior changes being detected as behavior differences to our padding changes. We initially send each ticket twice. If some behavior difference is detected, we send each ticket again to check for statistical significance. For the last byte, we send the ticket eight additional times, and for the second byte, two additional times. We use the statistical tests from [41] to test whether the behavior difference was significant ($p < 0.05$).

5.2 Offline Analysis

The last group of tests needs no further interaction with the server. First, we perform basic analyses of the retrieved tickets. We store the length of every session ticket. Next, we attempt to identify prefixes, like the key name, and continue with the analysis of cryptographic issues.

Detecting Prefixes To detect the prefixes, we build a prefix tree of the received tickets. The root node represents an empty string at depth zero. For each ticket, we add each byte to the tree. After the tree is built, we sum up the degrees of each depth. That is, we count how many divergences there are at each position in the tickets. We consider the depth with the highest degree to be the prefix length. This should detect the prefix length, even if we receive tickets with different key names, for example, due to a load balancer.

Unencrypted Tickets The first issue we check is whether the tickets are encrypted at all. For this, we iterate over each ticket and check whether one of the corresponding secrets is contained directly in the ticket bytes. If we find a secret, we consider the server vulnerable.

Reused Keystream Another issue we analyze is the reuse of keystreams. To check whether a keystream was reused, we take two tickets and their corresponding secrets. Let $T_1 = K_s \oplus p_1$ and $T_2 = K_s \oplus p_2$ be two tickets, where K_s is the reused keystream used to encrypt the plain state p_i for each ticket. To check whether a keystream was reused, we XOR their bytes: $T_x := T_1 \oplus T_2 = K_s \oplus p_1 \oplus K_s \oplus p_2 = p_1 \oplus p_2$. We then XOR the secrets corresponding to these tickets, for example, the master secrets ms_1 and ms_2: $\mathsf{ms}_x := \mathsf{ms}_1 \oplus \mathsf{ms}_2$. If the XOR value of the tickets $T_x = p_1 \oplus p_2$ contains the XOR value of the secrets ms_x, we know the tickets used the same keystream. We repeat this for each pair of tickets received from the server and their corresponding secrets.

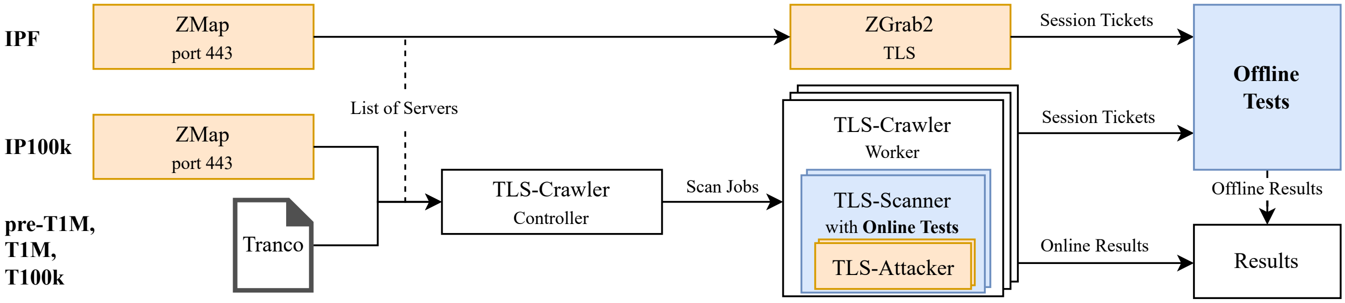

Figure 5: Setup for our scans. Blue parts contain our contribution. Orange parts are responsible for communicating with other servers on the internet.

5.2.1 Weak Keys

To test the issued tickets for the usage of weak keys in encryption and authentication algorithms, we created a list of possible key candidates. This list contains keys with repeating byte values (e.g., `0x000000...`) and keys with consecutively increasing bytes (e.g., `0x000102...`). We chose these specific keys to account for manually set or hard-coded keys that were chosen by humans or might have been copied from code examples. In addition, to target keys possibly constructed from uninitialized memory, we included known debug values from different applications [61], for example, uninitialized heap memory for applications compiled with Microsoft's C/C++ in debug mode (`0xCDCD...`) [10]. Inspired by [26], we also included keys used as examples in the NIST specification of AES [6]. In total, our list includes 144 different weak keys. We list our selection of keys in Section A.2.

We use our list of weak keys for extensive tests of the retrieved tickets. For the default encryption key, we attempt to decrypt the given ticket and test whether the decrypted plaintext contains one of the secrets corresponding to the ticket. For the default HMAC key, we compute the HMAC tag and test whether the ticket contains the resulting tag. By performing these tests, we consider two additional parameters:

Algorithm In our source code analysis (cf. Table 1), we observed eight different encryption algorithms and five different authentication algorithms. We extended the list of encryption and authentication algorithms with different key sizes. This led us to a total of 15 encryption algorithms and 5 HMAC algorithms. We provide a complete list of the used algorithms in Section A.1.

Ticket format As we perform a black-box evaluation, we do not know the structure of a received ticket. In our open-source analysis, we have seen that libraries generally follow a similar format. We do not know which format closed-source libraries use. Nonetheless, we make the following assumptions about the format of received tickets: We assume that the ticket might start with some prefix, like the key name. Additionally, we allow for a prefix that is not authenticated using the MAC. Next, we have an IV, followed by an optional length field (`len`), the ciphertext, and the MAC. The assumed format is shown in Figure 4.

In our implementation, this translates into the following constraints: When checking for a default HMAC key, we consider two offsets describing the HMAC input start and the HMAC input end. We assume that the prefix and suffix lengths are multiple of 8 bytes. Further, we assume that the prefix length is between 0 to 128 bytes and the suffix length is between 16 to 64 byes. This results in up to 126 combinations.

When checking for a default encryption key, we consider two offsets pointing to the IV/nonce and to the ciphertext, respectively. We assume the IV starts at any position in the range of 0 to 128 bytes. For the offset of the ciphertext, we assume that it starts immediately after the IV or two bytes after it. We also allow for an IV that is set to all zeroes and is not contained in the ticket. This results in up to 434 considered formats.

6 Large-Scale Evaluation

Scan Setup In Figure 5, we give an overview of the tools we used in our scans. We applied two approaches for the scans. Our first approach uses TLS-Crawler to perform the scans. We first determined which hosts to scan using the Tranco list [38] or ZMap [22] to find random IPv4 hosts that responded to a TCP SYN on port 443. These were fed into TLS-Crawler, which is split into controller and workers. The controller creates a list of scan jobs that are handled by worker instances. The workers run TLS-Scanner, where our online tests are implemented, against multiple servers simultaneously. To connect to the server, our tests utilize TLS-Attacker. After the online tests are run, the session tickets, including the session key material, are passed to the offline tests to analyze. All results are collected in a database.

Our second approach, for larger scans, did not use TLS-Crawler. Instead, we used ZGrab2 [3], as it is more fitted to scans on the whole IPv4 space. ZGrab2 stores the sesison tickets and corresponding key material, which we fed into our offline tests to generate results for these hosts.

We performed multiple scans against hosts on the Tranco list and publicly available IPv4 hosts. This mix represents the most popular websites as well as random smaller TLS servers,

			Statistics			Offline Analysis			Online Analysis	
Scan	Date	Tested Versions	Supports TLS	Issues Ticket	Resumes Ticket	Unencrypted Ticket	Weak STEK	Reused Keystream	Missing Auth. Protection	Padding Oracle
pre-T1M	2021-04	1.2	66,992	53,059	–	0	1,923	–	–	–
T1M	2021-05	1.2 – 1.3	760,293	594,238	547,159	0	3	–	–	–
T100k	2022-04	1.0 – 1.3	71,200	58,069	55,003	0	1	0	0	0
IP100k	2022-04	1.0 – 1.3	80,972	57,493	55,969	0	0	0	0	0
IPF	2022-08	≤1.2	39,390,365	29,621,531	–	0	189	1	–	–

Table 2: Results from our scans. We performed the pre-T1M and T1M scans to retrieve a first insight into the session ticket ecosystem. Subsequently, we added tests for keystream reuse and introduced the online analysis. For IPF, we performed an offline analysis on tickets obtained with ZGrab2 and tested only the highest protocol version, up to TLS 1.2, supported by the server.

covering a variety of implementations. Overall we performed the following scans.

T1M (and pre-T1M) We performed the first scan of the Tranco top 1M in May 2021 to retrieve the first insights into the session ticket ecosystem. This scan only covered a subset of our proposed vulnerabilities. We only scanned for keys consisting exclusively of zero-bytes in TLS 1.2 and 1.3. The other vulnerabilities and keys were envisioned after this scan and hence were not scanned here.

Before performing the final T1M scan, we performed test scans, subsequently summarized as the pre-T1M scan. The pre-T1M scan is an artifact of our research workflow meant to verify the first version of our scanner. Unexpectedly, our scanner quickly detected a large portion of Amazon servers vulnerable to zero-key flaws (cf. Section 6.2). Due to the severity of the discovered issue, we had to notify Amazon immediately before we could extend the original study with further ideas. As the issue fix would alter the results of subsequent scans, the pre-T1M entry presents the findings of this initial scan.

T100k and IP100k We performed two smaller but more detailed scan of 100k hosts each in April 2022. These were chosen as the top 100k from the Tranco list and 100k random IPv4 hosts that responded on port 443. For these, we performed the extensive online and offline tests to check whether these seem relevant. We tested TLS versions 1.0 through 1.3 for each host.

IPF Last, we scanned the entire IPv4 address range in August 2022. As earlier scans did not reveal issues in the online tests, we only performed offline tests. Only evaluating the offline tests further saves resources on not only our end but also on the scanned servers. To this end, we only collected session tickets and performed the offline tests afterward. This also means that we do not try to resume tickets. We performed three TLS connections per host using ZGrab2, each time attempting to obtain a session ticket. This is a reduction from the previous scans, where we performed ten connections to collect tickets. We chose this number as it reduces the number of connections while hopefully still managing to connect to different servers if there is a load balancer. Note that ZGrab2 does not support TLS 1.3, hence this scan only contains data up to TLS 1.2.

Ethical Considerations & Responsible Disclosure Our scans respected the rules for Internet-wide scanning proposed by Durumeric et al. [22]. All scans were performed from a dedicated server from our university that was set up in cooperation with our ISP. Its routing and IP address are separated from the main university network to not interfere with it.

To reduce the impact on the scanned servers, we verified locally that our tests do not cause any harm to locally deployed servers. By shuffling the connections made to the servers, we spread the computational load over time, reducing the risk of interrupting services. This is especially relevant for tests with a high amount of requests, like tests for CBC padding oracle attacks. Further, we established a reverse DNS entry for our IP, where we provided a website with contact information and information about the purpose of our scans. We cooperated with our ISP, which forwarded abuse emails to us. Using this information, the administrators could contact us and we were able to put their IPs on a block list for further scans or convince them about the benign nature of our scans.

We responsibly disclosed all of our findings to the respective developers and our national CERT. AWS developers acknowledged our findings and promptly fixed the issue on their servers [5]. While we did not get feedback from Stackpath, the issue was also resolved on their servers. We further contacted a hosting provider and an infrastructure company. The hosting provider told us they would forward our report to their customers. The infrastructure company did not respond. We sent the remaining list of servers to our local CERT.

6.1 Online Tests

In the T100k scan, we observed that 82% of the TLS servers issue tickets in at least one TLS version, as shown in Table 2. For the IP100k scan, 71% of TLS servers issue tickets in at least one version. Of these, around 95% (T100k) and 97% (IP100k) supported resumption with the issued tickets. In both scans, we observed that less than 1% issued tickets only on some connections. Most servers either always sent tickets or never. Support for session tickets in the T100k seemed higher in newer versions, while in the IP100k, it was similar

		Encryption		Authentication	
Scan	Servers Found	Algorithm	Key	Algorithm	Key
pre-T1M	1903	AES-256-CBC	00 00 ... 00 00	–	–
	20	AES-128-CBC	00 00 ... 00 00	HMAC-SHA256	00 00 ... 00 00
T1M	3	AES-128-CBC	00 00 ... 00 00	HMAC-SHA256	00 00 ... 00 00
T100k	1	AES-128-CBC	00 00 ... 00 00	HMAC-SHA256	00 00 ... 00 00
IPF	5	AES-256-CBC	00 00 ... 00 00	–	–
	94	AES-128-CBC	00 00 ... 00 00	HMAC-SHA256	00 00 ... 00 00
	12	AES-256-CBC	00 00 ... 00 00	HMAC-SHA384	00 00 ... 00 00
	3	AES-128-CBC	10 11 ... 1e 1f	HMAC-SHA256	20...2f 00...00[a]
	75	AES-256-CBC	31...31 00...00[b]	HMAC-SHA256	31...31 00...00[b]

a: This key consists of 16 consecutively increasing bytes followed by 16 0x00 bytes.

b: This key consists of 16 0x31 bytes followed by 16 0x00 bytes.

Table 3: Weak keys discovered in our scans.

for all TLS versions. We provide an overview of the ticket support per version in Table A.4.

Authenticated Tickets and Padding Oracles In our scans, we could not detect missing authenticity protection or a server vulnerable to padding oracle attacks. Our open-source analysis identified that the implementations use Encrypt-then-MAC or AEAD ciphers, which prevent such attacks. This may also be the case for servers deployed on the Internet. However, our tests did not search for more elaborate side channels (e.g., timing-based side channels). We thus cannot rule out all vulnerabilities enabling these attacks.

6.2 Offline Tests

In the scans focusing on offline tests, we determined similar support for session tickets. We did not find any unencrypted tickets, however, we found several servers that still allowed us to recover the plaintext.

6.2.1 Weak STEKs: AWS and Stackpath

We identified several weak STEKs in the offline tests for our scans. The results are shown in Table 3. Below, we further analyze our findings.

AWS In the initial scans, we found 1,903 unique hosts, all within the Tranco top 100k, belonging to AWS ALBs that used an all-zero STEK. These tickets were encrypted using AES-256-CBC and the format of RFC 5077 without the length field. We did not find the HMAC key and assume it was set correctly. Due to the high impact, we decided to report our finding before continuing with the full scan. We could see from the already obtained data that the hosts were not vulnerable at all times but only in some time intervals. This suggested an error in the key rotation, which was later confirmed to us by AWS developers. They stated that the issue mostly affected so-called redirector hosts. These are responsible for redirecting clients from an initial domain (e.g., `example.com`) to a final domain (e.g., `www.example.com`). However, we want to stress that connections to the initial domain often also contain HTTP cookies which should be confidential. Further, a MitM attack is also still possible.

Stackpath During the initial prescans, we also found 20 hosts belonging to Stackpath that used an all-zero STEK and HMAC key. These tickets were encrypted using AES-128-CBC, authenticated using HMAC-SHA256, and used the format of RFC 5077 without the length field. Again, we immediately reported the issue. The affected servers behaved differently than AWS servers. Only few tickets were vulnerable, but this was not limited to a specific timeframe as in the case of AWS. Interestingly, we found that all servers were hosted on the same IP address. Using an online reverse DNS lookup tool, we found a total of 171 domains on the same IP.[1] By rescanning these and collecting 1,000 tickets per hostname, we determined 90 hostnames for which we could observe vulnerable tickets. We identified that on vulnerable hosts, on average 1.4% of the issued tickets per host were affected. Stackpath did not give us any insight into how this came to be but resolved the issue.

6.2.2 Further Weak STEKs

During the final T1M scan, we found three more hosts using an all-zero key for both encryption and HMAC. These hosts used AES-128-CBC with HMAC-SHA256 and supported TLS 1.2. One of these hosts also supported TLS 1.3, also using zero-keys for session tickets. We did not further investigate these hosts. For the T100k, we found the host supporting TLS 1.2 and 1.3 to still be using a zero-key.

During the IPF scan, we found 189 servers using a weak key. Out of these, 111 servers used an all-zero key. For five hosts, we did not detect a weak key for the HMAC algorithm. Twelve hosts used SHA384, which no analyzed open-source library uses. This implies that they use a different implementation or reconfigured the used library. The remaining 78 servers did not use keys that solely consisted of zero-bytes.

[1] We used https://www.yougetsignal.com/tools/web-sites-on-web-server/.

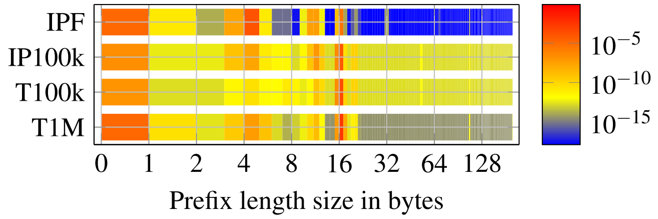

Figure 6: Detected ticket prefix sizes in our scans. The color is logarithmic and denotes the frequency observed in our scan, where 100% is the sum of all hosts.

We explore the structure of these keys below.

Non-Constant Keys Three servers encrypted their tickets using AES-128-CBC using the following key consisting of increasing bytes `0x101112...1F`. Looking at the first bytes of the ticket, we identified a key name that also uses bytes that count up, starting at zero (i.e., `0x000102...0F`). We further discovered that the authentication tag was computed using HMAC-SHA256 with the key `0x2021...2f`. By connecting to the server and taking a look at the HTTP response, we determined that it is an nginx server. We briefly explore a possible cause for this in Section 7.1.

Partial Keys We found another set of 75 servers that used an HMAC key that only consisted of the byte `0x31` (ASCII '`1`'). These servers used HMAC-SHA256 but with a 16-byte key, which internally gets padded with zero-bytes [37]. Investigating these servers, we could find that they also use AES-256-CBC with a key consisting of 16 × `0x31` bytes followed by 16 × `0x00` bytes. We assume the key was only partially initialized, and the initialized part was set to a weak value.

6.2.3 Reused Keystream

In our IPF scan, we tested for reused keystreams and found one affected server. Upon connecting to the server at a later date, the server no longer reused keystreams. The web interface shows that the server runs a software called 'GateManager'. Further, the format of the observed tickets resembles mbedTLS. Hence, we assume that the server had low entropy when scanned. When looking at the format (assuming mbedTLS), we can see that the tickets during the scan reused the same nonce. We contacted the software vendor but could not identify the root cause of this issue.

6.3 Session Ticket Structures

While performing our scans, we also collected statistics about the session ticket structure. As shown in Table 1, most analyzed open-source libraries use a key name of 16 bytes. This was also the dominant prefix length we observed throughout our scans, followed by a prefix length of zero bytes. We further found many hosts that used a prefix length of four bytes, which is also consistent with our open-source analysis. We provide an overview of the observed prefix lengths in Figure 6.

We further found that servers issue session tickets of vastly different lengths. Most ticket lengths were multiples of 16 in the range of 160 to 240 bytes, as shown in Figure A.9. However, not all servers responded with tickets of reasonable size. As shown in Figure A.8, many servers also responded with tickets longer than 1,000 bytes. The longest tickets we received were over 9,000 bytes. We could not observe such deviations in the tickets from the Tranco top 100k hosts; all the collected tickets were smaller than 700 bytes (cf. T100k).

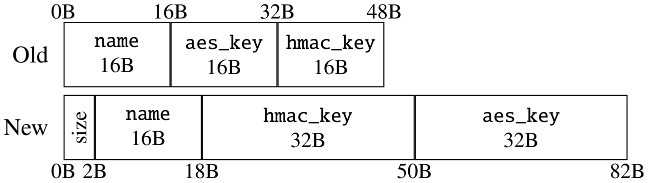

Figure 7: Internal structure of Nginx to store session key material. The format changed in December 2016. We show the format before (old) and after (new) this change.

We also received surprisingly short tickets. While the RFC defines no minimum ticket length, we expected that a ticket would contain enough bytes to encrypt the 48 bytes of the master secret in order to achieve a stateless session resumption or at least 32 bytes to mimic a session ID. The shortest "tickets" we received were 14 bytes and consisted of the ASCII string `TICKET FAILURE`. We observed this behavior for 18 servers during our IPF scan.

7 Discussion

We discovered two critical issues in our evaluation: using weak keys and reusing a nonce. In the following, we take a closer look at possible causes for the usage of weak keys. Afterward, we discuss possible countermeasures for both issues.

7.1 Potential Causes for Weak Keys

The web servers we analyzed in Section 4 allow the STEK to be provided as a file. For all servers, this file contains the key name, the AES key, and the HMAC key. Since the servers simply read the file contents, a static config file containing only zero-bytes or predictable structures, such as `00 01 02 ... 2f`, could result in weak keys.

The internally used structures of the analyzed web servers are similar to the file formats. In the following, we take a closer look at Nginx as it changed its internal structure in December 2016. The old and the new format are shown in Figure 7. The new format is longer and introduces an additional size field. If the size is set to 48, Nginx will use AES-128-CBC. Otherwise, it will use AES-256-CBC. We argue that this change could be the cause for some servers with weak keys we discovered with uninitialized or partially initialized keys. If an external program modifying this struct is unaware of the change, it might still assume the old format, while Nginx uses the new format. This can cause two behaviors, which we both observed in our scan. Either the `hmac_key` is initialized, but the `aes_key` is left empty. Or, only 16 bytes of each hmac_key and aes_key are set.

The first case occurs if the external program assumes all fields are aligned in memory and writes 48 bytes into the struct. In this case, the name and hmac_key are set, and aes_key is left uninitialized. Since size is not explicitly set to 48, Nginx will expect 32-byte keys. However, even if size is set to 48, the aes_key is still uninitialized. We have seen such behavior in our AWS finding and five times in the IPF scan (cf. Table 3).

The second case occurs if the external program assumes that the keys are still expected to be 16 bytes and writes each key independently. In this case, only 16 bytes are written to the hmac_key and aes_key. We have seen this behavior for the 75 hosts that used a key consisting of 16 0x31 bytes followed by 16 0x00 bytes (cf. Table 3).

7.2 Countermeasures

While the root causes for the uncovered issues can be hard to fix generically, we propose that TLS libraries deploy defense-in-depth countermeasures to protect themselves from catastrophic implementation defects. Note that TLS 1.3, due to its changes to the key resumption, message flow, and re-keying option, already mitigates some issues and is therefore preferred to TLS 1.2, as discussed in Section 2.1.1.

Weak Keys To prevent the usage of weak keys, we propose that libraries validate the keys when they are set. That is, whenever the STEK is rotated, the libraries validate the key material. This validation could be as trivial as checking that not every byte is a zero-byte. In most cases, this check is rather cheap and would have already prevented most of our discovered issues. However, since most STEKs live for at least an hour [48] and the STEK only has to be validated on a change, the check could also be more complex. Libraries could, for example, check the entropy using the hamming weight or the average hamming distance between each byte. While these tests may not be computationally free, more sophisticated tests might be justified, given the impact of a weak STEK.

Reused Nonces & Cryptographic Wear-out The impact of reused nonces and cryptographic wear-out can be mitigated by sticking to AES-CBC with HMAC, as proposed in RFC 5077. If these algorithms are not available, the library should respect the safety limits of the used cipher and mode of operation. This could, for example, be achieved through an internal counter that automatically switches the STEK once the threshold is reached. Since AES-GCM fails catastrophically in the case of nonce reuse, misuse-resistant algorithms should be preferred. Examples of this include AES-SIV [27], AES-GCM-SIV [28], AEZ [30], and MRO from the MEM-AEAD [25] cipher family. If these algorithms are not implemented, validating the reuse of the last nonce as a minimal check would already mitigate many vulnerabilities related to repeated randomness. This is, for example, already implemented in the Bouncy Castle cryptography library for AES-GCM in general.[2]

8 Related Work

TLS Large-Scaled Scans Several studies focused on the evaluation of the TLS ecosystem. Kotzias et al. conducted the first longitudinal study based on passive measurements and scans of the entire IPv4 address space over the course of six years [36]. Durumeric et al. analyzed the certificates used for HTTPS traffic [21]. These studies primarily focused on port 443. Holz et al. and Mayer et al. conducted studies that analyzed the ecosystem of other application protocols that utilize TLS to establish a secure channel, such as SMTP, IMAP, and XMPP [32, 40].

Large-scaled scans have also been conducted to estimate the attack surface for various known vulnerabilities and implementation pitfalls. Durumeric et al. scanned the IPv4 address space to determine how common the Heartbleed vulnerability was [20]. Valenta et al. identified several servers that did not validate key exchange parameters correctly [56, 57]. Dorey et al. further analyzed the prevalence of composite Diffie-Hellman moduli [19]. Böck et al. searched for servers vulnerable to Bleichenbacher's attack [11]. Merget et al. conducted large-scaled scans to estimate the impact of padding oracle attacks [42] and the Raccoon attack [41]. Brinkmann et al. searched for servers vulnerable to protocol confusion attacks [14]. Sullivan et al. analyzed collected TLS traffic and found servers that compute faulty RSA signatures allowing an attacker to obtain the secret key [50].

Entropy of Key Material Several studies identified insufficient entropy for generated key material for TLS and for operating systems that run the libraries. In a large-scale scan, Heninger et al. found TLS servers that shared RSA primes enabling an attacker to break the security of the affected RSA keys [29]. In a subsequent study, Heninger et al. also found servers generating exploitable ECDSA signatures due to flawed PRNGs [13]. Strenzke evaluated OpenSSL's PRNG and found issues that may surface for embedded devices [49]. Hughes analyzed the use of randomness inside TLS implementations based on collected traffic and identified various cases of low entropy [34].

TLS Session Tickets The effects of TLS session tickets on forward security have been studied in several articles [55, 59]. In 2016, Springall et al. evaluated how performance enhancement mechanisms, such as session tickets, weaken the forward secrecy of TLS in practice [48]. They discovered that 41% of the websites continuously listed in the Alexa Top Million rotated their STEK daily to mitigate the effects on forward secrecy. However, about 10%, in contrast, reused the STEK for at least 30 days. While they also provided an overview of the STEK identifiers used by TLS implementations, they did

[2] Added in release 1.56. https://www.bouncycastle.org/releasenotes.html

not test for implementation pitfalls.

In 2018, Sy et al. evaluated how users can be tracked across the web via TLS session resumption mechanisms, such as session tickets [53]. Their analyses of collected traffic showed that 65% of all users could be tracked permanently by at least one website using TLS session tickets with a lifetime of at least seven days due to a continuous renewal of the ticket. For a ticket lifetime of 24 hours, only 1.3% of users visit a website frequently enough to be tracked continuously. A theoretical analysis of the privacy properties of session tickets in TLS 1.3 was done by Arfaoui et al. [7].

In 2016, Valsorda found 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 [58]. Due to the similarity to the Heartbleed bug [20], the vulnerability was named Ticketbleed. Valsorda performed a large-scale scan of the Alexa Top Million list and showed that 949 web servers were vulnerable.

9 Conclusions

Our large-scale analysis of session tickets confirmed critiques raised by many cryptographic experts [48,53,59]. We showed that improper usage of session tickets can have devastating consequences for TLS connections and break forward secrecy guarantees delivered by TLS. The concrete issues discovered seem unlikely, but are an important reminder that looking for seemingly trivial or contrived issues can be worthwhile for auditors.

While our study could already detect a significant number of vulnerable servers, it was limited by the number of tests we could perform in a large-scale black-box evaluation. We only performed a white-box analysis of some servers in the lab. This did not cover all deployed TLS stacks, especially closed-source stacks. A white-box analysis of the used implementations might find more issues, making session tickets a generally risky feature.

Since even servers administrated by well-established cloud providers were affected by these issues, the community must acknowledge that catastrophic software defects happen in the real world and that we must build resilient security systems and avoid dangerous shortcuts.

At its core, we attribute the observed issues to the unauditability of session tickets. Since the STEK and the layout are unknown and never revealed to the client, clients cannot simply validate the strength of the key, the presence of a MAC, or even the algorithms used. This lack of transparency creates a space for bad implementations, silently failing crypto, and hidden backdoors. While key generation always seems to have this potential, it is especially severe for one-sided symmetric keys, as with asymmetric keys, at least the public key can be audited externally [29,52]. Having hard-to-analyze keys in a protocol at a place where a weak or leaked key causes the protocol to fail catastrophically is a huge risk that requires careful consideration. Since these keys cannot be audited externally, we argue that libraries should start auditing them themselves before they use them. In public key cryptosystems, it is already common practice to ensure that the generated key material is of a specific form, for example, to ensure that the key material will result in a strong key or as a safety net for failing random number generators. Adding additional checks to randomly drawn symmetric keys could, at least to some extent, ensure that accidentally weak key material does not break the protocol.

On the positive side, we could not observe any padding oracle in our scans. This vulnerability resurfaced in many recent scientific papers analyzing TLS record processing [4,47] and affected nearly 2% of the Alexa Top 1 Million servers in 2019 [42]. Compared to TLS record encryption, which uses the MAC-then-Encrypt construction, RFC 5077 recommends using Encrypt-then-MAC for AES-CBC with HMAC. Encrypt-then-MAC cryptographically prevents CBC padding oracle attacks, and we believe this is the main reason for not finding this vulnerability in session tickets. This example shows how a standard can positively affect a large-scale deployment of a cryptographic protocol.

Another positive development connected to session tickets is their usage in TLS 1.3, which reduces the impact of session ticket implementation dangers. First, TLS 1.3 servers send session tickets over an encrypted TLS channel, allowing attackers to obtain them only during the session resumption. Second, TLS 1.3 session tickets can be used such that revealing their contents does not break forward secrecy and only affects the resumed session. These two major differences positively affect the security of TLS 1.3 and should be considered when designing new cryptographic protocols.

Our work has also shown that the freedom in the specification led to the usage of different session ticket structures and cryptographic algorithms. These changes can have an impact on the security of session tickets. Given the previous works on security bounds affecting indistinguishability and integrity of encryption schemes in different contexts [9,16,31,39], we would like to encourage implementors to be mindful when selecting different algorithms and to have these limits in mind.

Acknowledgments

We would like to thank the reviewers as well as Kenneth Paterson for their valuable feedback and comments, which helped to improve the quality of our paper. Sven Hebrok was supported by the research project "North-Rhine Westphalian Experts in Research on Digitalization (NERD II)", sponsored by the state of North Rhine-Westfalia – NERD II 005-2201-0014. This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 450197914, under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by the German Federal Ministry of Education and Research (BMBF) through the project KoTeBi.

References

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

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

[3] ZGrab 2.0. https://github.com/zmap/zgrab2.

[4] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013*.

[5] Amazon. Resolved: Application load balancer session ticket issue, Apr 2021. https://aws.amazon.com/security/security-bulletins/AWS-2021-002/.

[6] National Institute of Standards and Technology. Advanced Encryption Standard (AES). Technical Report Federal Information Processing Standard (FIPS) 197, U.S. Department of Commerce, November 2001.

[7] Ghada Arfaoui, Xavier Bultel, Pierre-Alain Fouque, Adina Nedelcu, and Cristina Onete. The privacy of the TLS 1.3 protocol. 2019.

[8] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[9] Mihir Bellare and Björn Tackmann. The multi-user security of authenticated encryption: AES-GCM in tls 1.3. In *6th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016*.

[10] Andrew Birkett. Win32 debug crt heap internals. https://www.nobugs.org/developer/win32/debug_crt_heap.html.

[11] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of bleichenbacher's oracle threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security)*, 2018.

[12] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[13] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In *Financial Cryptography and Data Security - 23rd International Conference, FC*, 2019.

[14] Marcus Brinkmann, Christian Dresen, Robert Merget, Damian Poddebniak, Jens Müller, Juraj Somorovsky, Jörg Schwenk, and Sebastian Schinzel. ALPACA: application layer protocol confusion - analyzing and mitigating cracks in TLS authentication. In *30th USENIX Security Symposium, USENIX Security 2021*.

[15] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*, 2003.

[16] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G. Paterson. The security of chacha20-poly1305 in the multi-user setting. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2021.

[17] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[18] M. Dolan. International Standard Audiovisual Number (ISAN) URN Definition. RFC 4246 (Informational), February 2006.

[19] Kristen Dorey, Nicholas Chang-Fong, and Aleksander Essex. Indiscreet logs: Diffie-hellman backdoors in TLS. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[20] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference, IMC*, 2014.

[21] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS certificate ecosystem. In *Proceedings of the 2013 Internet Measurement Conference, IMC*, 2013.

[22] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium (USENIX Security)*, 2013.

[23] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, 2007. Special Publication (NIST SP), https://tsapps.nist.gov/publication/get_pdf.cfm?\pub_id=51288.

[24] Enigmail Forum. Enigmail 1.7 completely broken. https://sourceforge.net/p/enigmail/forum/support/thread/3e7268a4.

[25] Robert Granger, Philipp Jovanovic, Bart Mennink, and Samuel Neves. Improved masking for tweakable blockciphers with applications to authenticated encryption. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2016.

[26] greenluigi1. How I hacked my car, May 2022. https://programmingwithstyle.com/posts/howihackedmycar/.

[27] Shay Gueron and Yehuda Lindell. GCM-SIV: full nonce misuse-resistant authenticated encryption at under one cycle per byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[28] D. Harkins. Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES). RFC 5297 (Informational), October 2008.

[29] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium (USENIX Security)*, 2012.

[30] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2015.

[31] Viet Tung Hoang, Stefano Tessaro, and Aishwarya Thiruvengadam. The multi-user security of gcm, revisited: Tight bounds for nonce randomization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[32] Ralph Holz, Johanna Amann, Olivier Mehani, Mohamed Ali Kâafar, and Matthias Wachs. TLS in the wild: An internet-wide analysis of tls-based protocols for electronic communication. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.

[33] R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (Internet Standard), September 2009.

[34] James P. Hughes. Badrandom: The effect and mitigations for low entropy random numbers in TLS, 2021. UC Santa Cruz, https://escholarship.org/uc/item/9528885m.

[35] Fiona Klute. Cve-2020-13777: TLS 1.3 session resumption works without master key, allowing mitm, June 2022. https://gitlab.com/gnutls/gnutls/-/issues/1011.

[36] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of TLS deployment. In *Proceedings of the Internet Measurement Conference 2018, IMC*, 2018.

[37] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[38] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. 2019.

[39] Atul Luykx and Kenneth G. Paterson. Limits on Authenticated Encryption Use in TLS, 2017. https://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf.

[40] Wilfried Mayer, Aaron Zauner, Martin Schmiedecker, and Markus Huber. No need for black chambers: Testing TLS in the e-mail ecosystem at large. In *11th International Conference on Availability, Reliability and Security, ARES*, 2016.

[41] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In *30th USENIX Security Symposium, USENIX Security*, 2021.

[42] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. Scalable scanning and automatic classification of TLS padding oracle vulnerabilities. In *In 28th USENIX Security Symposium (USENIX Security)*, 2019.

[43] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[44] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 4507 (Proposed Standard), May 2006.

[45] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), January 2008.

[46] SEC Consult Blog. Fake crypto: Microsoft outlook s/mime cleartext disclosure (cve-2017-11776). https://sec-consult.com/blog/detail/fake-crypto-microsoft-outlook-smime-cleartext-disclosure-cve-2017-11776/.

[47] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[48] Drew Springall, Zakir Durumeric, and J. Alex Halderman. Measuring the security harm of TLS crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC, 2016.

[49] Falko Strenzke. An analysis of openssl's random number generator. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2016.

[50] George Arnold Sullivan, Jackson Sippe, Nadia Heninger, and Eric Wustrow. Open to a fault: On the passive compromise of TLS keys via transient errors. In *31st USENIX Security Symposium (USENIX Security)*, 2022.

[51] Nick Sullivan. TLS session resumption: Full-speed and secure, February 2015. https://blog.cloudflare.com/tls-session-resumption-full-speed-and-secure.

[52] Petr Svenda, Matús Nemec, Peter Sekan, Rudolf Kvasnovský, David Formánek, David Komárek, and Vashek Matyás. The million-key question - investigating the origins of RSA public keys. pages 893–910. USENIX Association, 2016.

[53] Erik Sy, Christian Burkert, Hannes Federrath, and Mathias Fischer. Tracking users across the web via TLS session resumption. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC, 2018.

[54] Ye Tang, Huiyun Li, and Guoqing Xu. Cache side-channel attack to recover plaintext against datagram tls. In *2015 5th International Conference on IT Convergence and Security (ICITCS)*, 2015.

[55] Tim Taubert. Botching forward secrecy - the sad state of server-side tls session resumption implementations, November 2014. https://timtaubert.de/blog/2014/11/the-sad-state-of-server-side-tls-session-resumption-implementations/.

[56] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[57] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of CurveSwap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P*, 2018.

[58] Filippo Valsorda. Ticketbleed (CVE-2016-9244), February 2017. https://filippo.io/ticketbleed/.

[59] Filippo Valsorda. We need to talk about session tickets, September 2017. https://blog.filippo.io/we-need-to-talk-about-session-tickets.

[60] Serge Vaudenay. Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques*, 2002.

[61] Wikipedia. Magic number (programming), Apr 2022. https://en.wikipedia.org/w/index.php?title=Magic_number_(programming)&oldid=1084703437#Debug_values.

[62] David Ziemann. Analysis of the gnutls session ticket bug (cve-2020-13777), July 2020. https://www.hackmanit.de/de/blog/118-analysis-of-the-gnutls-session-ticket-bug-cve-2020-13777.

A Appendix

	T100k		**IP100k**	
TLS	71,200	71.20%	80,972	80.97%
TLS 1.0	32,064	45.03%	36,484	45.06%
Issues Ticket	19,182	59.82%	26,534	72.73%
Resumes Ticket	18,050	94.10%	25,805	97.25%
TLS 1.1	35,112	49.31%	39,753	49.09%
Issues Ticket	21,362	60.84%	30,256	76.11%
Resumes Ticket	20,116	94.17%	29,408	97.20%
TLS 1.2	70,648	99.22%	72,118	89.07%
Issues Ticket	54,111	76.59%	52,851	73.28%
Resumes Tickets	51,152	94.53%	51,382	97.22%
TLS 1.3	38,630	54.26%	30,700	37.91%
Issues Ticket	33,631	87.06%	26,221	85.41%
Resumes Ticket	31,624	94.03%	25,335	96.62%
Issues Ticket (≠ 32B)	30,177	78.12%	22,770	74.17%
Resumes Ticket (≠ 32B)	28,878	95.70%	22,252	97.73%

Table A.4: Servers supporting session tickets by protocol version. For TLS 1.3, we also filtered servers that did not offer session tickets of 32 bytes length. These are most likely Session IDs.

A.1 Tested Algorithms

We used the following Algorithms when testing for weak keys:

DES in ECB and CBC modes.

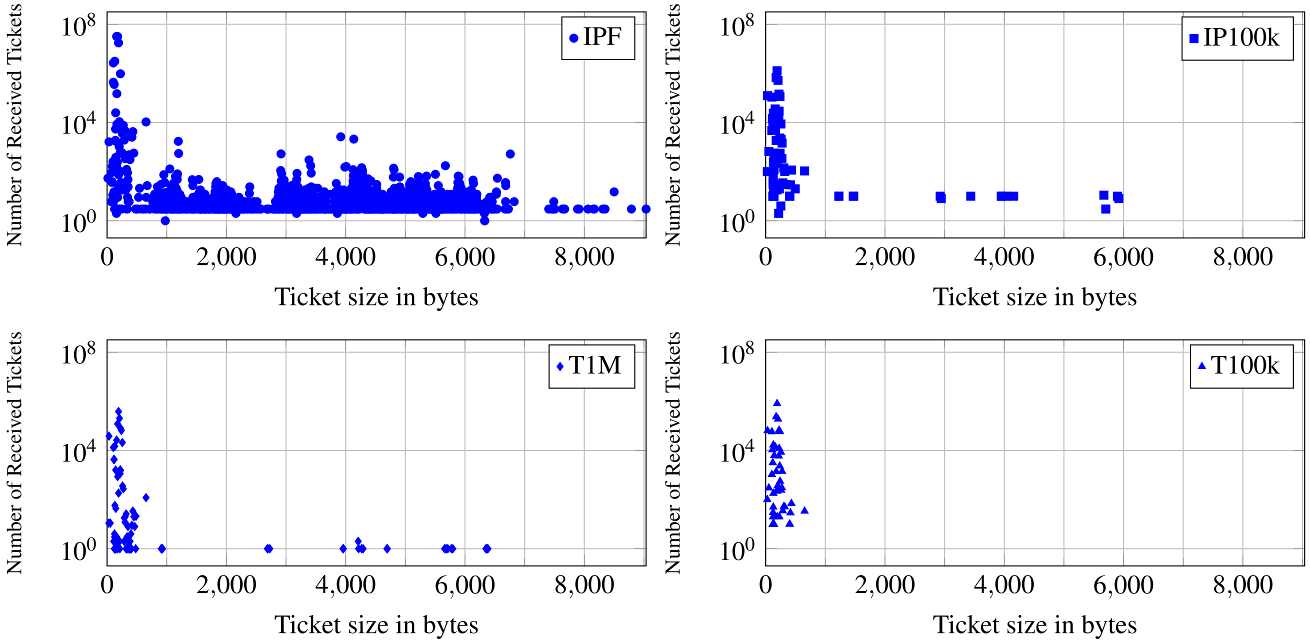

Figure A.8: Observed ticket sizes in our scans.

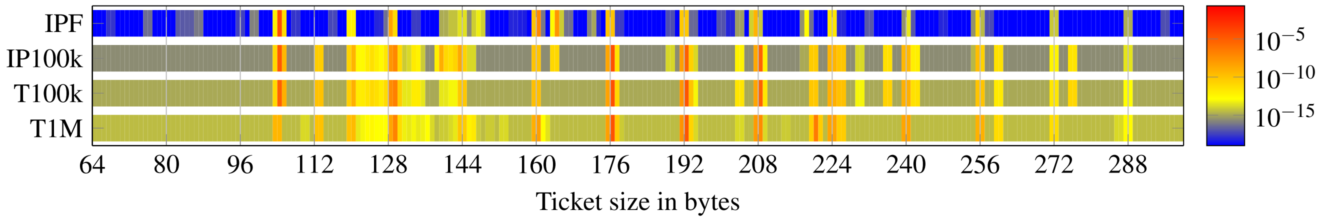

Figure A.9: Most commonly observed ticket sizes in our scans. The color is logarithmic and denotes the frequency observed in our scan, where 100% is the sum of all tickets.

3DES in ECB and CBC modes.

AES with 128bit and 256bit keys. In ECB, CBC, CTR, CCM, and GCM modes. Note that for CCM and GCM we have ignored the authentication tags.

ChaCha20 on its own.

When testing for weak authentication keys we always assume that an HMAC is used. For the underlying hash algorithms we test MD5, SHA1, SHA256, SHA384, SHA512.

A.2 Tested Default Keys

We now describe which keys we generate. Inspired by keys found in code examples, we chose a mixture of educated guesses and known values.

48× Constant Bytes We consider keys consisting of the same byte repeated (e.g., `0x00000000`). For the values we chose multiples of 16 ± 1 (e.g., 0, 1, 15, 16, 17, ...).

48× Increasing Bytes We also chose keys consisting of consecutively increasing bytes values (e.g., `0x00010203`). As a starting point, we again chose multiples of 16 ± 1.

3× Special Values We further include two special cases of keys with consecutively increasing bytes: Starting at 0 or 1 with a step size of 16 (`0x00102030` and `0x01112131`). Additionally, we included `0x00112233445566778899aabbccddeeff` if the key size is 16 bytes.

41× Debug Values We used 41 known debug values [61], some of which are used to denote uninitialized memory. We repeated these values to fill the desired key length.

4× NIST Keys Inspired by [26], we also included keys that are used as example keys in the NIST specification of AES [6]. However, we only use them if the desired key length is appropriate.

This results in up to different 144 keys due to overlap and some keys only being tested in specific key sizes.