USENIX'23 Artifact Appendix: 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

A Artifact Appendix

A.1 Abstract

We performed a large-scale analysis of TLS session tickets. To this end, we extended TLS-Scanner with further tests which detected a variety of weak keys being used in the wild. To support developers and server administrators when evaluating their session ticket configurations, we publish our extension for TLS-Scanner. This extension is able to detect the vulnerabilities discussed in our paper. Within this artifact evaluation, we show that our extension is capable of detecting the vulnerabilities from our paper.

A.2 Description & Requirements

TLS-Scanner is an established tool to scan TLS servers for weaknesses. We implemented new probes to scan for weaknesses related to session tickets. These perform the tests outlined in our paper. Additionally, we provide a test server to test the scanner against (not part of the artifact, but used to evaluate it). For both tools we provide the source code, Dockerfiles, and Docker images. We recommend using the Docker images, as they ensure a reproducible environment.

A.2.1 Security, privacy, and ethical concerns

We are not aware of any exploitable issues in the tool. It should be secure to run on your machine. To evaluate a TLS server, TLS-Scanner needs to connect with that server, effectively revealing your IP address. The scanner also sends an HTTP request to the server which includes TLS-Attacker as the user agent.

The scan initiates multiple TLS connections to the server. Depending on the number of threads (option `-threads`) you could overwhelm the server. In our scans, we ensured this does not happen by scanning multiple servers at once with a shared limited threadpool (done by TLS-Crawler[1]). When using the scanner, use a low number of threads. The test server we provide is single threaded, so only a single (connection) thread should be used there.

Any data our probes exfiltrate from the ticket is already known to the scanner. However, if it is able to exfiltrate data from a ticket, this indicates a severe issue (as outlined in our paper) that you should report to the server administrator.

Test Server The test server uses an outdated version of BoringSSL and adds further parameters to the included server. The ticket decryption is implemented such that it is vulnerable to padding oracle attacks if the authenticity of the ticket is not ensured (e.g., no MAC). We recommend running it inside an isolated environment.

A.2.2 How to access

We provide a repository summarizing our artifact at `https://github.com/tls-attacker/We-Really-Need-to-Talk-About-Session-Tickets/tree/ad64fe34f41894f1aa5bbec65cf0446cdb0ad3f8`. This includes the TLS-Scanner source code[2] and a testserver. Further, this also contains the Dockerfiles to build Docker images of both components yourself. Alternatively, you can get the Docker image from Docker Hub.[3] Within this document, we describe how to run the scanner using maven and the server using docker. Additional topics, such as using the scanner with Docker or the server from source, are covered in the artifact repo's readme.

[1]`https://github.com/tls-attacker/TLS-Crawler`
[2]`https://github.com/tls-attacker/TLS-Scanner/commit/a0d4c1a910f0eb3e5ee3e28c9435818820c67919`
[3]snhebrok/tls-scanner-ae and snhebrok/vulnerable-bssl

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

For TLS-Scanner a Java 11 development kit and maven are required.[4] To run the server using Docker, Docker needs to be installed. You can also build the image yourself using the Dockerfile contained in our repository or pull it from Docker Hub.

A.2.5 Benchmarks

None.

A.3 Set-up

We provide multiple ways to set up the scanner and test server. Within this document we cover building the scanner from source using maven.

A.3.1 Installation

Using Source and Maven You need to clone the code and then compile it using maven. This automatically fetches all dependencies.

```
git clone --recurse-submodules
↪ https://github.com/tls-attacker/
↪ We-Really-Need-to-Talk-About-Session-Tickets
cd We-Really-Need-to-Talk-About-Session-Tickets
git checkout c71fb839bd4ad2dc00cbb1a578d7d4254f8aec17
mvn clean package
```

A.3.2 Basic Test

To verify that the scanner is initialized correctly run the scanner without any arguments:

```
cd TLS-Scanner/apps
java -jar TLS-Server-Scanner.jar
```

If everything works correctly this should print an error stating that the provided parameters could not be parsed (including a stack trace). This error message should state that the option `-connect` is required and also include a stack trace. The available options should be printed afterward.

If you pass the `-connect` flag followed by a host, the specified host should be scanned.

If an error occurs, ensure you are using Java 11 to build and run the project. Other versions usually fail.

[4]More information about the TLS-Attacker projects and their requirements can be found under https://github.com/tls-attacker/TLS-Attacker-Description.

A.3.3 Test Server

The Docker image is available at Docker Hub as `snhebrok/vulnerable-bssl`[5] with the tag `sessionticket-ae`. Running the image will automatically pull it. You can still pull it explicitly with `docker pull snhebrok/vulnerable-bssl:sessionticket-ae`.

A.4 Evaluation workflow

A.4.1 Major Claims

To evaluate the ecosystem in our paper, we used TLS-Scanner. We claim to be able to detect different vulnerabilities related to TLS session tickets:

(C1): We can detect a variety of default keys for encryption and authentication. This is shown in experiment (E1).

(C2): We can detect padding oracle vulnerabilities. This is shown in experiment (E2).

(C3): We can detect missing ticket authentication. This is shown in experiment (E2).

A.4.2 Experiments

TLS-Scanner scans a single server at a time. For our experiments we propose to scan our test server. You can also scan other servers, but these might not be vulnerable to our proposed vulnerabilities. All time estimates were created using a laptop with an i7-1165G7 and 32G of RAM. Each scan should take about five to ten minutes with the parameters we recommend below.

Basic Test Execution For each experiment we describe which parameters to pass to the test server for this experiment. After the test server is started, you need to run TLS-Scanner against the server. The scan might take some minutes and finishes by outputting the results. This also contains results not related to session tickets. The results related to session tickets are located in the section with the heading `SessionTicketEval`. We describe what this section should contain for each experiment. For all experiments, there are some parameters that you should always set, which we outline below.

Running the Scanner For all tests, you need to execute the scanner against the testserver as follows:

```
java -jar TLS-Server-Scanner.jar -connect
 ↪ [host] -scanDetail NORMAL
```

`[host]` is the host to scan (including port). When using the docker test server as specified, this should be `172.17.0.1:8000`.[6] The detail affects how many test vectors

[5]https://hub.docker.com/r/snhebrok/vulnerable-bssl

[6]Check whether docker assigns this IP to one of your network interfaces. You can also use any other IP assigned to your device.

are tested against the server. For our paper we used `DETAILED`, but all experiments work with `NORMAL` (the default value) as well.

Running the Test Server For the test server we recommend running it as follows:

```
docker run --rm  -it -p8000:8000
 ↪  snhebrok/vulnerable-bssl:sessionticket-ae
 ↪  s_server -accept 8000 -loop -www
```

Depending on the experiment, further parameters need to be added.

(E1) Detecting default keys:
[5 human-minutes + 5 compute-minutes]
Scan a server whether it uses one of our proposed weak keys. The scanner outputs the detected key and format of the ticket.
Test Server Preparation: The test server needs to use a weak key. Example parameters are

```
-ticketEnc AES-128-CBC
-ticketEncKey 00
-ticketHMac SHA256
-ticketHMacKey 00
-ticketHMacKeyLen 32
```

Results: The summary should contain the following lines:

```
Ticket use default STEK (enc) : true
Ticket use default STEK (MAC) : true
```

Further down is a section `Default STEK` which contains details about the detected keys for encryption and HMAC (if you set both groups of parameters). This includes the detected format, algorithm, and key. For encryption, this also contains which secret is included in the ticket.
No padding oracle or MAC check issues should be found as an Encrypt-then-Mac scheme is used (albeit with a weak key).
Within the repository's readme we describe how to manually verify this attack using OpenSSL.

(E2) Missing Authentication and Padding Oracles:
[5 human-minutes + 15 compute-minutes]
Scan a server whether it is not properly authenticating its tickets and even has a padding oracle vulnerability.
Test Server Preparation: The test server must not use authentication (HMAC) for the ticket. To detect a padding oracle vulnerability, a CBC cipher must be used. Example parameters are

```
-ticketEnc AES-256-CBC
-ticketHMac None
```

Results: The summary should contain the following lines:

```
No (full) MAC check : true
Vulnerable to Padding Oracle : true
```

Further down is a section `Manipulation`. This summarizes the behavior the server showed when inducing `bitflips` into a ticket. Several behaviors are preclassified:

- **A** The modified ticket was accepted. The authenticity of the ticket was hence not verified (completely).
- **#** The modified ticket was accepted, but key material unknown to the scanner was used. That is, the server recovered some corrupted key material from the modified ticket.
- **_** The modified ticket was rejected, and a normal handshake was performed. This should be the case if the authenticity of the ticket is properly ensured.
- Other characters are explained in the results.

Further down is a subsection `Padding Oracle` which contains details stating at which position the oracle was found. This also includes the recovered plaintext, as well as what value was XOR-ed at which position to recover the plaintext. Further down is a summary of the observed behavior difference per offset (when modifying the last byte). Note that multiple offsets might show different behavior, but not all are necessarily caused by a valid padding oracle vulnerability. This is internally verified by trying to recover the second byte. As the `Overall Result` is `TRUE`, this second byte was found.

A.5 Notes on Reusability

We believe the source code can help other researchers to more easily detect default keys in encrypted blobs with a possibly unknown format (classes `SessionTicketEncryptionFormat`, `SessionTicketMacFormat`, and `DefaultKeys`). This approach could be applied to other protocols where one party chooses key material to protect a payload.

The code is currently under internal review and will be merged into the main versions of TLS-Scanner and TLS-Anvil[7] in the future. This allows researchers to more easily scan for issues related to session tickets. In combination with TLS-Crawler[8], this also allows for performing larger scans.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

[7] Maehren et al. at Usenix 22

[8] We used `https://github.com/tls-attacker/TLS-Crawler/commit/d0c6e1e3d6a7168da2181ab74fa0d33b13b426f2` in our scans.