Systematic Fuzzing and Testing of TLS Libraries

Juraj Somorovsky
Horst Görtz Institute for IT Security
Ruhr University Bochum
Hackmanit GmbH
juraj.somorovsky@rub.de

Abstract

We present TLS-Attacker, an open source framework for evaluating the security of TLS libraries. TLS-Attacker allows security engineers to create custom TLS message flows and arbitrarily modify message contents using a simple interface in order to test the behavior of their libraries.

Based on TLS-Attacker, we present a two-stage fuzzing approach to evaluate TLS server behavior. Our approach automatically searches for cryptographic failures and boundary violation vulnerabilities. It allowed us to find unusual padding oracle vulnerabilities and overflows/overreads in widely used TLS libraries, including OpenSSL, Botan, and MatrixSSL.

Our findings motivate developers to create comprehensive test suites, including positive as well as negative tests, for the evaluation of TLS libraries. We use TLS-Attacker to create such a test suite framework which finds further problems in Botan.

1. INTRODUCTION

Transport Layer Security (TLS) [27] is used to secure the connection to websites, Web services, or to create Virtual Private Networks (VPNs) and connect LANs from different locations. Different application scenarios and protocol extensions quickly raised the complexity of this standard. Its complexity led to various designs as well as implementation failures in various attack scenarios. In the last few years, we saw attacks targeting improper encryption algorithms and configurations [12, 13, 44], complex state machines [17, 25, 39], extension specifications [19, 47], or attacks targeting implementation failures with buffer overflows and overreads [49].

The large number of recent attacks has motivated researchers to provide further security analyses of TLS and to develop novel security evaluation tools. In recent scientific studies authors have considered the proper evaluation of TLS state machines [25, 17], and they have also developed tools for sending protocol messages in an arbitrary order [25] and even tools for modifying specific message fields [17]. The message field modifications provided by these tools are rather static, i.e., a developer only has the ability to explicitly define values of specific message fields; however, he cannot execute dynamic field modifications given precomputed values in TLS message flows, which is very important for dynamic fuzzing. In view of the shortcomings of previous approaches, and with intentions for extending the test coverage of existing TLS libraries, it was necessary to develop a new TLS testing framework – TLS-Attacker. TLS-Attacker is able to create *arbitrary TLS protocol flows* and execute *dynamic modifications* in TLS messages based on precomputed values.

Impact and applicability. Our approach allowed us to find new vulnerabilities in widely used TLS libraries. These include padding oracle vulnerabilities in OpenSSL [8], MatrixSSL [6] and Botan [1], or boundary violations in Botan and the first pre-release of OpenSSL-1.1.0. Furthermore, with this approach we exposed the fact that GnuTLS [10] does not verify specific message variables but instead silently proceeds the TLS handshake. The vulnerabilities have been reported to the developers and fixed in the newest versions.

TLS-Attacker and modifiable variables. Our framework relies on a construct called *modifiable variable*. A modifiable variable is a container for basic data types like integers or byte arrays. By accessing these data types, the modifiable variable can dynamically modify the content of the original variable value. For example, it is possible to increase or decrease an integer value or to execute an XOR operation on a given byte array. We use modifiable variables to construct TLS messages and TLS records. This allows us to dynamically modify any byte at any time in the TLS protocol flow.

The main goal of TLS-Attacker is to provide an easy-to-use framework with a simple user interface, allowing developers to create custom TLS protocol flows in order to discover state machine attacks [25, 17] or test countermeasures against cryptographic attacks [55, 23]. This is possible directly in Java, using the TLS-Attacker interface. Furthermore, it is even possible to define *custom protocol flows* and modifications using simple *XML messages*.

Two-stage fuzzing approach. To prove the practicability of TLS-Attacker, we use it to construct a two-stage fuzzing approach. In the first stage, we introduce cryptographic fuzzing for known vulnerabilities like padding oracle attacks [55] or Bleichenbacher attacks [23]. In the second stage, we then systematically modify protocol message vari-

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978411

ables and protocol flows to trigger specific implementation bugs or buffer boundary violations.

TLS test suite. The padding oracle vulnerability we discovered in OpenSSL [9] (CVE-2016-2107) was introduced by writing a constant-time patch that should have mitigated the Lucky 13 attack [13]. Unfortunately, a missing length check for sufficient HMAC length turned the OpenSSL server from a rather complex timing oracle to a direct padding oracle since the server responded with a different TLS alert. This issue went unnoticed for nearly three years even though OpenSSL became the primary TLS target library of the security research community. We observed a similar problem in the MatrixSSL library. The impact of the insufficient padding check, however, was worse than in OpenSSL; the developers introduced a buffer overflow vulnerability by attempting to patch the Lucky 13 attack.

These two cases clearly show that writing and maintaining critical cryptographic libraries is of huge importance. New security critical functionalities must be validated with proper test suites. These test suites should not only include positive tests, they must include *negative tests* verifying correct library behavior when sending invalid messages or incorrectly formatted data.

We use TLS-Attacker to create a test suite concept for validating TLS libraries. With our concept, the developers can create valid and invalid TLS message flows containing arbitrary messages. The TLS responses can be validated with predefined assertions which check for the correct message contents. For example, it is possible to validate the correctness of TLS alert message types or proper cryptographic properties. The first test cases in our test suite already showed insufficiencies in the Botan cipher suite support.

Contributions. This work makes the following contributions:

- **TLS-Attacker:** We provide a novel framework for the evaluation of TLS libraries, which can be used by security researchers or developers. The code is on GitHub: https://github.com/RUB-NDS/TLS-Attacker.
- **Novel fuzzing approach for TLS:** Based on TLS-Attacker, we implement a two-stage fuzzing approach for the evaluation of TLS servers. Our approach allows us to find different vulnerabilities in widely used libraries.
- **Modifiable variables:** We present a concept of modifiable variables which provide a high flexibility for the implementation of arbitrary cryptographic protocols *beyond TLS*.
- **TLS test suite:** We also create a concept for testing TLS libraries which is easily extensible with positive and negative tests.

In our work we do not attempt to claim TLS-Attacker is complete or that it detects every vulnerability. Our findings, however, show that such a tool is necessary for the development of secure TLS libraries. TLS-Attacker is currently being integrated into Botan and MatrixSSL test suites.

2. TRANSPORT LAYER SECURITY

In the TCP/IP reference model, the TLS protocol is located between the transport layer and the application layer.

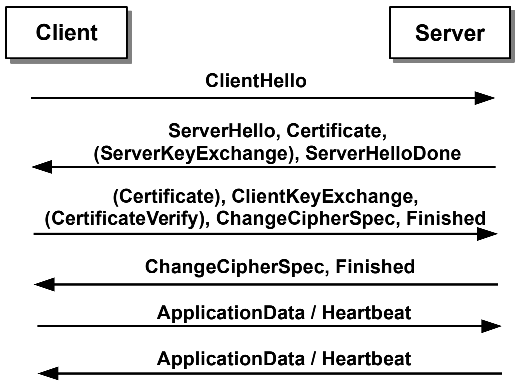

Client
Server
ClientHello
ServerHello, Certificate,
(ServerKeyExchange), ServerHelloDone
(Certificate), ClientKeyExchange,
(CertificateVerify), ChangeCipherSpec, Finished
ChangeCipherSpec, Finished
ApplicationData / Heartbeat
ApplicationData / Heartbeat

Figure 1: TLS protocol. After performing a TLS handshake, the peers can communicate securely and exchange `Application` or `Heartbeat` messages.

Its main purpose is to protect application protocols like HTTP or IMAP. The first (unofficial) version was developed in 1994 by Netscape, and was named *Secure Sockets Layer*. In 1999, SSL version 3.1 was officially standardized by the IETF Working Group and renamed *Transport Layer Security* [26]. The current version is 1.2 [27]. Version 1.3 is currently under development [28]. In addition to TLS, which acts over reliable TCP channels, the working group standardized DTLS [48] (Datagram TLS), which works on the top of UDP.

TLS is complex and allows communication peers to choose from a large number of different algorithms for various cryptographic tasks (key agreement, authentication, encryption, integrity protection). A *cipher suite* is a concrete selection of algorithms for the required cryptographic tasks. For example, `TLS_RSA_WITH_AES_128_CBC_SHA` defines RSA-PKCS#1 v1.5 public-key encryption in order to exchange a premaster secret, and it also defines symmetric AES-CBC encryption with a 128-bit key and SHA-1-based HMACs.

2.1 The Handshake Protocol

In order to establish a TLS connection between two peers and exchange application data, a *TLS handshake* is executed (cf. Figure 1). A TLS handshake is initiated by a TLS client with a `ClientHello` message. This message contains information about the TLS version and a list of supported cipher suites. The server now responds with a `ServerHello` message containing the selected cipher suite. Furthermore, it sends its certificate in the `Certificate` message and indicates the end of transmission with the `ServerHelloDone` message. The client then sends a `ClientKeyExchange` message, which contains an encrypted premaster secret. Based on the premaster secret, all further connection keys are derived. Finally, both parties send the `ChangeCipherSpec` and `Finished` messages. The former notifies the receiving peer that subsequent TLS messages will use the newly negotiated cipher suite. The `Finished` message contains an HMAC computed over all the previous handshake messages based on a key derived from the premaster secret. Thereby, both peers are authenticated and can exchange application data or `Heartbeat` messages [50].

Note that this is an example of a TLS handshake with an RSA cipher suite. The specification also supports (EC)DH

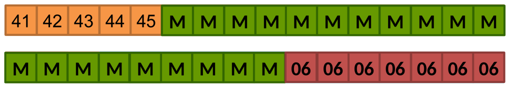

41 42 43 44 45 M M M M M M M M M M M
M M M M M M M M M 06 06 06 06 06 06 06

Figure 2: When processing five plaintext bytes with AES-CBC and HMAC-SHA, the encryptor needs to append 20 bytes of the HMAC-SHA output and seven bytes of padding.

cipher suites or usage of preshared keys. In addition, it is possible for the client to authenticate with a certificate or to use a session resumption to resume previous sessions. These methods result in TLS handshakes with slightly different structures with additional protocol messages. For example, if a client-authentication is used, the client additionally sends `Certificate` and `CertificateVerify` messages.

The exchanged messages have strict structures. For example, an RSA-based `ClientKeyExchange` message consists of a handshake type identifier (`0x10`), 3-byte long length indication and an encrypted premaster secret.

Different messages and flows result in a complex protocol making the design of *TLS state machines* and the proper verification of *protocol message structures* very challenging.

2.2 The Record Layer

The record layer is used to transmit protocol messages. Basically, it wraps the protocol messages and adds the information about the TLS protocol version, message type, and message length. The contents of the TLS records are encrypted after `ChangeCipherSpec` messages are exchanged.

In order to encrypt TLS records, it is possible to use different cryptographic primitives. One of them is a MAC combined with AES in CBC mode of operation. It uses the *MAC-then-Pad-then-Encrypt* mechanism [27]. This means that the encryptor first computes a MAC over the plaintext, then it pads the message to achieve a multiple of block length, and finally it uses AES-CBC to encrypt the ciphertext. For example, if the encryptor attempts to encrypt five bytes of data and uses HMAC-SHA (with 20 bytes long output), we end up with two blocks. The second block needs to be padded with seven bytes `0x06`, see Figure 2. Note that the encryptor can also choose a longer padding and append 23, 39, ...or 247 padding bytes.

For the description of our work, it is also crucial that one record message can include *one* protocol message or *several* messages at once. It is even possible to *split one protocol message into several records*. This adds to the complexity of the TLS standard.

2.3 TLS Extensions

Different application scenarios and cryptographic usages have resulted in definition of various extensions to the TLS protocol. The extensions can add cryptographic abilities of TLS peers (supported elliptic curves [22, 42]), define new protocol messages (heartbeat [50]), or change the maximum number of bytes transmitted in one TLS record (Maximum Fragment Length [31]). Protocol extensions are negotiated in the `ClientHello` and `ServerHello` messages and their correct processing is also crucial for securing a TLS protocol execution.

2.4 TLS Libraries

There are several widely used libraries supporting TLS, ranging from open source libraries like OpenSSL to closed source libraries like Microsoft's SChannel (Secure Channel).

In this paper, we analyze the following TLS server implementations: Botan [1], GnuTLS [10], Java Secure Socket Extension [5], MatrixSSL [6], mbedTLS [7], and OpenSSL [8].

3. ATTACK CATEGORIZATION

Recent years have shown that despite the wide usage of TLS, TLS libraries suffer from severe security vulnerabilities. In this section we concentrate on the description of the attacks relevant to our paper since they are necessary to describe our findings. Further attacks and their categorization can be found in [43, 51].

We have organized the relevant attacks into three basic categories.

3.1 Cryptographic Attacks

Padding oracle attacks. One of the main design failures in SSLv3 and TLS is the specification of the *MAC-then-Pad-then-Encrypt* scheme in CBC cipher suites. In 2002 Vaudenay showed that this scheme makes security protocols potentially vulnerable to padding oracle attacks [55]. These attacks are based on the malleability of the CBC mode of operation. CBC allows an attacker to flip specific bits in the plaintext without knowing the secret key. If a server allows the attacker to distinguish between valid and invalid padding bytes (e.g., by sending different error messages), the attacker can decrypt the message as follows. He starts with decrypting the last message byte. To this end, he iteratively flips bits in this byte and sends the message to the server. Once he receives a valid padding response, he knows he has correctly guessed the `0x00` byte. This allows him to decrypt the last plaintext byte. Afterwards, he can proceed with further padding bytes and decrypt the whole message [55].

In order to mitigate this attack, the implementation must not allow an attacker to distinguish valid from invalid padding structures in the decrypted messages. In 2013 AlFardan and Paterson presented the Lucky 13 attack [13] and showed that implementing countermeasures against padding oracle attacks in TLS is very challenging. Lucky 13 exploits a timing side-channel arising from the countermeasures described in the TLS recommendation [27].

TLS implementations attempt to implement various countermeasures to make padding oracle attacks impossible. However, recent evaluations and scientific studies show that TLS implementations still contain insufficient padding verifications [24, 41] or are vulnerable to variants of the Lucky 13 attack [11, 35].

Bleichenbacher's attack. One of the most important attacks in TLS history is Bleichenbacher's million message attack [23]. The attack targets the RSA PKCS#1 v1.5 encryption scheme, which is used in the TLS protocol to encrypt a shared secret between both TLS peers. Essentially, Bleichenbacher's attack is also a padding oracle attack. The attack is based on the malleability of the RSA encryption scheme and assumes the existence of an oracle that responds with "valid" or "invalid" according to the PKCS#1 v1.5 validity of the decrypted message.

A server defending against this attack must not allow for the distinction between valid and invalid ciphertexts. However, recent studies show insufficiencies in the application of this countermeasure, in the Java TLS implementation (JSSE) and the Cavium accelerator chips [44]. Further studies show how to improve efficiency of this attack or how to apply it to different protocols and standards [16, 15, 36].

3.2 State Machine Attacks

TLS is a complex protocol containing different message flows. This results in complex state machine implementations which can contain severe security bugs. The first relevant security vulnerability was discovered in 2014 and was named Early CCS, or CCS injection vulnerability [39]. This vulnerability allows an attacker to inject an early `ChangeCipherSpec` message into the TLS handshake and force the TLS peer to derive a shared key based on a null secret. If a Man-in-the-Middle (MitM) attacker and both TLS peers use vulnerable OpenSSL versions, the attacker can force both peers to establish a connection using a null secret and thus read the whole communication.

The Early CCS vulnerability prompted researchers to search for state machine vulnerabilities. They found different unexpected state transitions in widely used TLS libraries [17, 25]. For example, the Java TLS implementation contained a serious vulnerability which allowed one to finish the TLS handshake without `ChangeCipherSpec` messages. This resulted in a plaintext communication between the client and the server.

3.3 Overflows and Overreads

The Heartbleed bug in OpenSSL [49] has shown cryptography engineers how critical a simple buffer overread can be. Heartbleed allowed an attacker to read random bytes from a server's memory, for example, private cryptographic keys [53]. The reason was a buffer overread vulnerability in the OpenSSL heartbeat processing implementation. It forced major servers to renew their private keys and certificates.

In the recent years, additional problems in various TLS libraries like buffer overflows or integer overflows have appeared [29, 9, 2]. These buffer boundary violations motivated us to execute further security evaluations of TLS implementations.

4. REQUIREMENTS AND RELATED WORK

The recent development in the area of TLS and the high number of memory and state machine attacks motivated us to construct an enhanced evaluation of TLS implementations. In the following section we first describe requirements for a flexible TLS library. We analyze different approaches to achieve these requirements, discussing their advantages and disadvantages.

4.1 Requirements for a Flexible TLS Testing Framework

Given the recent TLS attacks, we can summarize the following requirements for a new, flexible analysis tool.

Flexible stateless TLS handshake execution. Flexible protocol execution is necessary for state machine validations as well as for different fuzzing strategies. It is important that the framework does not derive the current state from the exchanged protocol messages. The security developer has to be able to set the protocol execution state at an arbitrary point of time, after an arbitrary number of messages have been exchanged. For example, he is able to define specific state transitions, or the exact point when the messages will be encrypted.

Flexible modification of arbitrary TLS variables. The security developer has to be able to execute arbitrary modifications to TLS variables. This includes various variables used in the TLS protocol flow, for example, length variables indicating the length of TLS messages and extensions, as well as specific cryptographic keys and secrets. It is not only necessary to simply set these variables, but the security developer has to perform *variable modifications* at runtime. For example, by evaluating the POODLE vulnerability [45] in a TLS server library, a security engineer analyzes whether a correct padding validation has been implemented. For this purpose, he needs to execute modifications in specific plaintext bytes *before* the message gets encrypted in the record layer. More precisely, if the plaintext message consists of $data||MAC||pad$ (data concatenated with MAC and padding), the security developer is only interested in the modification of specific padding bytes (e.g., he wants to apply an XOR operation on the first padding byte). The rest of the plaintext message has to stay untouched, otherwise a MAC failure can be triggered.

Systematic fuzzing of message variables. In order to trigger specific vulnerabilities, the TLS testing framework has to provide a list of variables given in the TLS protocol and be able to systematically modify these variables.

Easy to use interface. The TLS testing framework has to provide a simple interface to create new protocol flows and modifications. The security developer has to be able to define new protocol sets or protocol fuzzings and execute them automatically with the tested TLS library.

Detection of crashes and invalid protocol flows. The TLS testing framework has to provide an ability in order to detect TLS server crashes or invalid protocol flows. If a failure is detected, the protocol flow has to be recorded and the developer has to be able to execute the same protocol flow to re-analyze the discovered issue (e.g., after triggering a buffer overflow).

4.2 Approaches and Related Work

The first trivial approach is to patch an existing TLS library. However, patching every variable in the TLS protocol flow could be complicated. The code would need to be modified at different places, nested in several function levels. This would result in a huge overhead, especially when considering the complex code of open source libraries like OpenSSL.

Another possible approach is to use a TLS library and control its flow using a debugging interface. For example, the Java Platform Debugger Architecture (JPDA) [4] provides a programming interface that allows a developer to create software agents which can monitor and control Java applications. The developer could use the JSSE library and write a fuzzing extension within a Java agent. The agent would modify only the needed variables directly during TLS protocol flow. This would allow one to execute correct TLS message flows and modify only specific variables. This approach, however, has a significant drawback. TLS libraries are constructed to execute *correct* TLS protocol flows. In order to trigger the Early CCS vulnerability or other state machine attacks, an invalid TLS protocol flow has to be constructed.

A possible approach to execute fuzzing attacks is to take a fuzzing framework (e.g., Peach Fuzzer[1] or American Fuzzy Lop[2]) and initialize it with an intercepted TLS protocol flow. The fuzzer generates its messages based on the given protocol messages and sends them to the server. This approach could be applied to several plaintext messages in the TLS handshake. However, the TLS protocol flow also contains messages that are encrypted and authenticated using freshly generated keys. For example, in order to send a `Heartbeat` message, the complete TLS handshake must be executed first [50].[3]

De Ruiter and Poll implemented a customized tool to evaluate TLS state machines [25]. Their tool, however, does not allow one to modify custom protocol message variables. Berdouche et al. designed a novel TLS tool – FLEXTLS [18]. FLEXTLS was previously used to develop many prominent attacks (e.g., Triple Handshake [19]) and to discover state machine attacks [17]. This tool allows one to construct new protocol message flows and set custom variable values. However, in FLEXTLS the variables can only be initialized *with explicit values*. The tool is not intended to perform dynamic variable modifications. This means that more difficult modifications of variables with XOR or ADD operations are not supported. Furthermore, no TLS fuzzing is supported.

Very recently, two novel TLS testing frameworks have been developed: tlsfuzzer and Scapy-SSL/TLS [38, 46]. These TLS frameworks support stateless protocol executions and variable modifications. They are useful for developing new attacks and protocol modifications. Currently, they do not contain any consistent fuzzing strategies, nor dynamic variable modifications.

5. TLS-ATTACKER: DESIGN AND IMPLEMENTATION

The state-of-the-art of TLS evaluations motivated us to develop a novel flexible TLS framework. The main goal of this framework is to offer developers and security engineers a simple and accessible approach to evaluate their TLS implementations.

TLS-Attacker is implemented in Java with the support of the Maven project management tool.[4]

5.1 Modifiable Variables

At the heart of our framework, we implement a construct called `ModifiableVariable`. A `ModifiableVariable` is a wrapper for simple data types like integers or byte arrays. This wrapper contains the original value of a specific variable and provides its value by a `getter` method. While accessing the variable, the `ModifiableVariable` container is able to apply predefined modifications.

[1]http://peachfuzzer.com
[2]http://lcamtuf.coredump.cx/afl
[3]We note that the Heartbleed bug could have been triggered by sending an unencrypted `Heartbeat` message during the TLS handshake since the vulnerable OpenSSL version incorrectly accepted the `Heartbeat` message before the `ChangeCipherSpec` message.
[4]https://maven.apache.org

A simplified example of this construct provided for integer data types gives the following listing:

```java
public class ModifiableInteger {
  private int origValue;
  private Modification modification;

  public int getValue() {
    if (mod != null) {
      return modification.modify(origValue);
    }
    return origValue;
  }
}
```

`ModifiableInteger` contains two variables: a wrapped `origValue` and a `modification`. The `origValue` is used to hold the original integer value computed during a program execution. While accessing the variable over `getValue()`, this method is able to execute specific modifications before the original value is returned.

Usage of modifiable variables. The best way to present the functionality of this concept is by means of a simple example:

```java
ModifiableInteger i = new ModifiableInteger();
i.setOriginalValue(30);
i.setModification(new AddModification(20));
System.out.println(i.getValue());  // 50
```

In this example, we define a new `ModifiableInteger` and set its value to 30. Next, we define a new modification – `AddModification` – which simply returns a sum of two integers. We set its value to 20. If we execute the above program, the result 50 is printed. This is because by accessing the original integer value, its value gets increased by 20.

A similar concept is applied for all basic data types: integers, bytes, or byte arrays. `ModifiableInteger` contains, for example, these modifications: `add`, `explicitValue`, `xor`, `shift`, or `subtract`. Similar modifications are accessible to other numeric types. `ModifiableByteArray` is a container for byte arrays and contains, for example, the following modifications: `delete`, `insert`, or `xor`. It is worth mentioning that the modifications can be concatenated and executed successively.

Modifiable variables in protocol messages. All relevant protocol messages and record data are stored in modifiable variables. An example is shown in the following `ClientHello` protocol message:

```java
public class ClientHelloMessage {
  ModifiableInteger compressionLength;
  ModifiableByteArray compressions;
  ModifiableInteger cipherSuiteLength;
  ModifiableByteArray cipherSuites;
  ...
}
```

The `ClientHello` message is used in the TLS protocol flow and the variable values are computed dynamically. Before executing the protocol flow, the modifiable variables allow us to set arbitrary variable modifications or to define explicit variable values. The variables are then dynamically modified during the protocol execution. For example, the developer can use 2 cipher suites and set the explicit value of `cipherSuitesLength` to 5. TLS-Attacker then uses an invalid length value while serializing the `ClientHello` message, which could possibly trigger an overflow.

5.2 High-Level Overview

TLS-Attacker is divided into several Maven modules (cf. Figure 3). The concept of modifiable variables is located in a separate `ModifiableVariable` module so that further applications can profit from its functionality. The `Transport` module contains transport handling utilities for TCP and UDP. The core module is `TLS`, which contains a TLS protocol implementation. This is divided into further packages. The `protocol` package contains protocol messages and their handlers. The `workflow` package contains a flexible protocol flow implementation which allows one to define arbitrary message order. Further relevant packages are depicted in Figure 3. The `Attacks` and `Fuzzer` modules are based on the TLS functionalities and define several TLS attacks and fuzzing techniques. The `TestSuite` module defines an extensible TLS test suite.

TLS-Attacker currently implements the TLS 1.0, 1.1, 1.2 and DTLS 1.2 protocol versions and the client-side functionality, including client-side authentication. Furthermore, it implements these features:

- Key exchange algorithms: RSA, ECDH(E), DH(E)
- Encryption algorithms: AES-CBC, 3DES-CBC
- Extensions: EC, EC point format [22], Heartbeat [50], Maximum fragment length [31], Server name indication [31], Signature and hash algorithms

This allows us to cover the majority of the relevant TLS attacks. Further features are in development.

5.3 The TLS Module

`TLS` is the core module of TLS-Attacker. It implements the complete TLS functionality, using modifiable variables and the transport handlers from the `Transport` module. It solely relies on the cryptographic functionality provided by the standard Java cryptographic providers and the Bouncy Castle library (version 1.54).

The `TLS` module is divided into several packages. The `config` package contains classes for TLS protocol configuration. TLS constants (cipher suites, key exchange algorithms, alerts) are defined in the `constants` package. The `crypto` package contains TLS specific cryptographic functionalities which extend the basic behavior of Java cryptographic providers.

The `protocol` package implements the TLS protocol messages and their handlers. Each protocol message is defined by a handler (responsible for message processing) and a message state (representing the current state of the TLS message). For example, the `handshake` package contains the `HandshakeMessageHandler` and `HandshakeMessage` classes (see Figure 3). The abstract `HandshakeMessage` class contains general handshake variables used by all handshake messages, e.g., the handshake message type. `ClientHelloMessage` extends `HandshakeMessage` and includes `ClientHello` specific variables like an array of cipher suites or the cipher suite length. Note that all these variables are modifiable to achieve high flexibility. Processing of handshake messages is provided by the `HandshakeMessageHandler` classes. Each handler implements two functions: `prepareMessage` used while sending TLS message and `parseMessage` used while parsing incoming messages.

A similar concept of handlers and messages is implemented for the TLS record handling defined in the `record` package.

The `workflow` package contains TLS protocol executors. The TLS protocol execution solely depends on the defined TLS messages. A TLS executor takes a list of protocol messages as an input, searches for proper message handlers, and lets them process the defined messages. This approach presents two notable advantages:

1. It makes it very convenient for a TLS-Attacker user to define custom TLS protocol flows with arbitrary message ordering just by setting specific TLS messages.
2. The processed TLS protocol flows can be stored for further analysis or even for a repeatable execution.

5.4 Using TLS-Attacker Interfaces

In the following section we present how TLS-Attacker interfaces can be used to construct a custom protocol message flow detecting vulnerability to Bleichenbacher's attack. We chose this attack as an example since it requires a deep intervention in the TLS protocol functionality, including modification of the plain padded premaster secret.

Detecting Bleichenbacher's oracle with ≈ 10 lines of code. The design of TLS-Attacker supports simple definitions of new attacks. The following example shows a custom protocol flow by explicitly setting the plaintext value of a padded premaster secret. This code can be used to evaluate the correctness of countermeasures against Bleichenbacher's attack [23].

```
TlsContext context = initializeTlsContext(config);
WorkflowExecutor executor =
    initializeWorkflowExecutor(context);

// Setting explicit modification of the premaster
    secret
RSAClientKeyExchangeMessage rsa = new
    RSAClientKeyExchangeMessage();
ModifiableVariable<byte[]> pms = new
    ModifiableVariable<>();
pms.setModification(new ExplicitValueModification(
    VALUE));
rsa.setPlainPaddedPremasterSecret(pms);

// Constructing protocol message flow
List<ProtocolMessage> m = context.
    getProtocolMessages();
m.add(new ClientHelloMessage());
m.add(new ServerHelloMessage());
m.add(new CertificateMessage());
m.add(new ServerHelloDoneMessage());
m.add(rsa);
m.add(new ChangeCipherSpecMessage(ConnectionEnd.
    CLIENT));
m.add(new FinishedMessage(ConnectionEnd.CLIENT));
m.add(new Alert(ConnectionEnd.SERVER));

// Protocol execution
executor.executeWorkflow();
```

By setting a custom premaster secret, the security engineer enforces TLS-Attacker to execute a TLS handshake with this custom value. The execution of TLS handshakes with different premaster secret values can trigger different server behaviors and thus reveal an oracle which can be used to perform Bleichenbacher's attack.

Note that constructing such a protocol flow with a common TLS library would need a deep knowledge of the library. On the other hand, with TLS-Attacker, security engineers can easily use the TLS-Attacker interfaces to define new protocol flows with specific message modifications and embed them into their test suites.

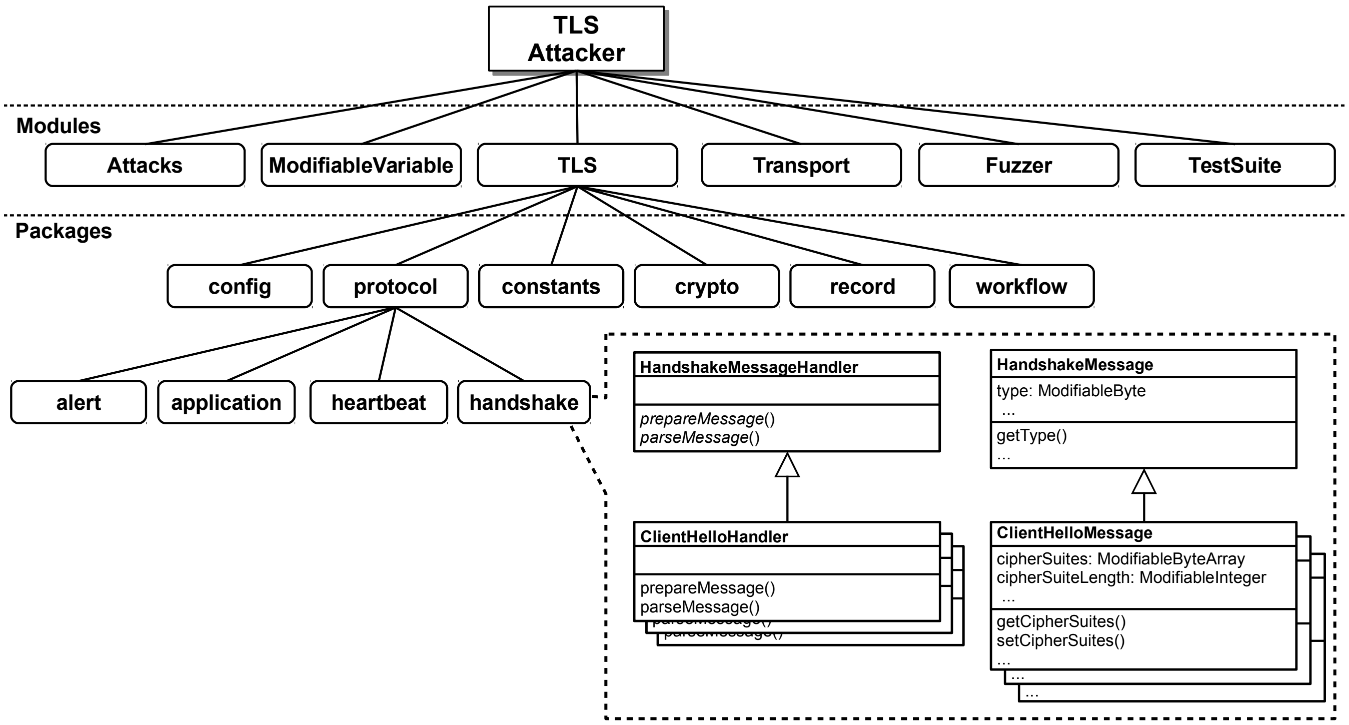


Figure 3: TLS-Attacker is divided into several Maven modules. The core module is TLS implementing the protocol functionality and flexible protocol execution. All protocol messages use modifiable variables.

Not only for Java developers. Convenient XML serialization. It is not necessary to develop Java in order to construct arbitrary protocol flows with custom modifications. Instead, we decided to use JAXB (Java Architecture for XML Binding) [3] for serialization and deserialization of TLS protocol flows. This allows a developer to define the same protocol flow as above using a simple XML document:

```
<workflowTrace>
 <protocolMessages>
  <ClientHello>
   <messageIssuer>CLIENT</messageIssuer>
   <supportedCompressionMethods>
     <CompressionMethod>NULL</CompressionMethod>
   </supportedCompressionMethods>
   <supportedCipherSuites>
    <CipherSuite>TLS_RSA_WITH_AES_128_CBC_SHA</
        CipherSuite>
    <CipherSuite>TLS_RSA_WITH_3DES_EDE_CBC_SHA</
        CipherSuite>
      ...
   </supportedCipherSuites>
  </ClientHello>
  <ServerHello>
   <messageIssuer>SERVER</messageIssuer>
  </ServerHello>
  <Certificate>
   <messageIssuer>SERVER</messageIssuer>
  </Certificate>
  <ServerHelloDone>
   <messageIssuer>SERVER</messageIssuer>
  </ServerHelloDone>
  <RSAClientKeyExchange>
   <messageIssuer>CLIENT</messageIssuer>
   <plainPaddedPremasterSecret>
    <byteArrayExplicitValueModification>
     <explicitValue>
00 02 8B 2B FC 66 ED 21  07 9A F8 5F 92 8E 34 38
                    ....
81 51 B5 91 E6 B3 1D F8  BB 98 48 31 8F 73 A2 CE
60 0A 31 CC 34 D7 0D 42  F6 3F D1 59 20 53 71 5E
     </explicitValue>
    </byteArrayExplicitValueModification>
   </plainPaddedPremasterSecret>
  </RSAClientKeyExchange>
  <ChangeCipherSpec>
   <messageIssuer>CLIENT</messageIssuer>
  </ChangeCipherSpec>
  <Finished>
    <messageIssuer>CLIENT</messageIssuer>
  </Finished>
  <Alert>
   <messageIssuer>SERVER</messageIssuer>
  </Alert>
 </protocolMessages>
</workflowTrace>
```

6. FUZZING WITH TLS-ATTACKER

TLS-Attacker provides a suitable framework for performing fuzzing attacks. Based on our observations about the previous TLS attacks and vulnerabilities, we designed an extensible approach for performing a systematic evaluation of TLS libraries. Our approach is divided into two stages, see Figure 4.

In the following sections we first describe the basics behind our fuzzing strategies and vulnerability detection, and then we describe both fuzzing stages.

6.1 Fuzzing Strategies

We employed several fuzzing strategies from the mutators used in Peach Fuzzer[5] and American Fuzzy Lop,[6] which are relevant for TLS fuzzing. We implemented these strategies using our modifiable variables.

[5]http://community.peachfuzzer.com/v3/Mutators.html
[6]http://lcamtuf.coredump.cx/afl/technical_details.txt

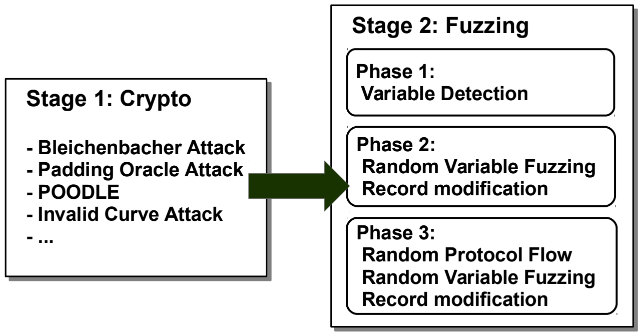

Stage 1: Crypto
- Bleichenbacher Attack
- Padding Oracle Attack
- POODLE
- Invalid Curve Attack
- ...
Stage 2: Fuzzing
Phase 1: Variable Detection
Phase 2: Random Variable Fuzzing Record modification
Phase 3: Random Protocol Flow Random Variable Fuzzing Record modification

Figure 4: Fuzzing with TLS-Attacker is divided into two stages; first various cryptographic attacks are executed, then a systematic protocol fuzzing is started.

For example, TLS-Attacker generates the following modifications during fuzzing with integers. The original integer value can be XORed with random bits, shifted left or right, and increased or decreased by a random value. In addition, specific values can be returned based on a dictionary consisting of a zero value and values causing overflows in specific number representations. Similar strategies are employed by modification of further numeric data types.

Byte arrays are modified by applying additional strategies. TLS-Attacker automatically generates modifications which duplicate arrays, remove or insert specific bytes, or shuffle the given byte array. The dictionary with explicit values contains an empty array or arrays consisting of `0x00` and `0xFF` values.

Note that the design of modifiable variables allows TLS-Attacker to chain generated modifications as well.

6.2 Vulnerability Detection

In order to detect buffer boundary violations, integer overflows, or other memory corruptions [34, 33], the runtime behavior of the TLS library has to be observed. For this purpose, we use AddressSanitizer (ASan).[7] ASan is a memory error detector which can be enabled at compile time in recent versions of LLVM or GCC compilers. It is typically used while fuzzing C and C++ applications. If a fuzzer finds a memory error in an application compiled with ASan, the application crashes, prints an error message, and exits with a non-zero code.

We use Asan to compile C and C++ TLS libraries before we start TLS-Attacker fuzzing. If a memory error or a different bug in a TLS server compiled with ASan is triggered, the server crashes and outputs an error message describing the cause of the detected boundary violation.

ASan is of course not suitable for TLS applications developed in different languages, like Java. For anomaly detection in Java servers and other servers which cannot be compiled with ASan, we analyze the protocol flows with a TLS context analyzer. The TLS context analyzer investigates whether a TLS protocol flow has been executed correctly, contains an invalid protocol flow with an additional protocol message, or whether a message in a valid protocol flow is modified by a specific modification.

In case a runtime error or an invalid protocol flow in the above cases is detected, TLS-Attacker stores the protocol flow in an XML file. This file can later be used for further analysis.

[7]http://clang.llvm.org/docs/AddressSanitizer.html

6.3 Two-Stage Fuzzing

Cryptographic Attacks.

In the first stage, we investigate the cryptographic behavior of the analyzed TLS server. We attempt to trigger different error messages by sending invalid padding bytes or forcing the server to accept invalid elliptic curve points [21, 37]. TLS-Attacker can dynamically collect the server responses and store them for further analysis.

Currently, TLS-Attacker implements checks for Bleichenbacher's attack [23], padding oracle attacks [55], invalid curve attacks [37], and POODLE [45].

Fuzzing for Buffer Boundary Violations.

TLS-Attacker allows one to execute random variable modifications or to construct invalid messages. We use these features in the second stage where we attempt to trigger an invalid server behavior and find buffer boundary violations. This stage is divided into three phases.

The starting point for each phase is a set of known TLS protocol flows. This includes correct TLS protocol flows as depicted in Figure 1, as well as several invalid TLS protocol flows identified in the previous years [39, 17, 25]. At the beginning of the fuzzing process, TLS-Attacker attempts to execute these protocol flows and stores correctly executed, complete protocol flows for further executions in the next phases. These are described below.

Phase 1: Searching for "influencing variables". A TLS protocol flow and its messages contain a huge amount of variables: message length values, derived keys, or certificates. Not all of these variables are suitable for fuzzing. For example, it is not necessary to fuzz random values (e.g., client random) or variables which are not validated. Therefore, our framework allows one to use an explicit blacklist of variables that should be omitted during the fuzzing process. In addition, in the first phase, TLS-Attacker iteratively changes all variables from the TLS protocol flow and analyzes whether those variables influence the correct protocol flow.

For example, we found that some TLS libraries do not validate specific length variables at all. See the following section.

Phase 2: Fuzzing with variable modifications. In the second phase, we continue only with the variables from the first phase that were identified as influencing the TLS protocol flow. We execute *correct* protocol flows with randomly modified variables. In this phase, more variables can be modified at once or the modifications can be chained.

As discussed in Section 2.2, a TLS record can contain one protocol message, several protocol messages (this is the default behavior of TLS-Attacker), or even one protocol message can be sent in several records. The distribution of protocol messages in a different number of records could potentially trigger an invalid server behavior. Therefore, in addition to the variable modifications, in this phase we attempt to split protocol messages into different numbers of TLS records with randomly chosen record lengths.

Phase 3: Fuzzing with random protocol flows. In the last phase, we continue the fuzzing process with additional randomized protocol flows. For this purpose, we add or remove random protocol messages from the configured protocol sequences.

7. TLS FUZZING EVALUATION

The number of fuzzing attempts in Stage 2 can be configured by the developer. Depending on the performance of the tested library, the number of TLS protocol flows and the resulting duration varies. For example, for OpenSSL-1.1.0-pre3, we were able to execute 166,000 flows in one hour, resulting in about 46 protocol flows per second. The tests were executed on a laptop with an Intel Core i7 5600U CPU.

Note that it is not our intention to fully analyze the impact of the detected vulnerabilities or describe complete attacks. Our contribution is to prove the practicability of our fuzzing approach by finding novel vulnerabilities and their sources. The presented vulnerabilities have been reported and patched by the library developers, proving their relevance.

7.1 Padding Oracle Attacks

In recent years we observed several scientific results proving the padding oracle exploitation possibilities in widely used TLS libraries. Irazouqui et al. showed how to exploit cache access times in co-located virtual machines in cloud environments to gain sufficient timing differences for executing the Lucky 13 attack [35]. Almeida et al. and Albrecht and Paterson showed that an extended version of the Lucky 13 timing attack is still applicable to the s2n library provided by Amazon [14, 11].

Surprisingly, it is not always necessary to execute complex timing attacks. As we show in the following section prominent TLS libraries are vulnerable to direct padding oracle attacks, where servers respond with different alert messages. The vulnerabilities result from incorrect sanity checks of the decrypted CBC ciphertexts [13].

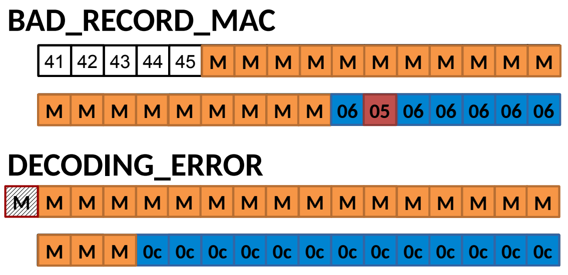
Unusual padding oracle in Botan.

By evaluating Botan 1.11.21, we observed different response messages sent by the analyzed TLS server. Further analysis revealed that directly after the record data is decrypted, the implementation evaluates the length of the unpadded data `record_len` and whether this data has enough length for HMAC validation (e.g., while using `TLS_RSA_WITH_AES_128_CBC_SHA`, the HMAC is 20-byte long):

```
if(record_len < mac_pad_iv_size)
  throw Decoding_Error("Record␣sent␣with␣invalid␣
      length");
```

If there is not enough data for MAC validation, the server responds with a `DECODE_ERROR` alert. This alert differs from a typical case when an invalid padding is used (cf. Figure 5).

In order to trigger the `DECODE_ERROR` alert for a `TLS_RSA_WITH_AES_128_CBC_SHA` cipher suite (20-byte long HMAC output, 16-byte long AES block cipher), the decrypted message has to consist of at least 32 bytes (two blocks), and it has to contain at least 13 valid padding bytes. This means that the attacker has to set at least 13 bytes to trigger the alert and use padding oracle attacks. This is possible in scenarios where the attacker knows parts of the plaintext, for example, if the victim uses a browser which sends known HTTP headers to the website [30].



BAD_RECORD_MAC
41 42 43 44 45 M M M M M M M M M M M
M M M M M M M M M 06 05 06 06 06 06 06
DECODING_ERROR
M M M M M M M M M M M M M M M M M
M M M 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c

Figure 5: Direct padding oracle provided by Botan 1.11.21. In case of an invalid padding, Botan responds with `BAD_RECORD_MAC`. In case of a valid padding, Botan attempts to process the HMAC. It responds with a `DECODE_ERROR` if the number of remaining bytes is insufficient for HMAC validation.

The vulnerability was fixed after our disclosure in version 1.11.22 and has been labeled CVE-2015-7824.

Unlucky patch of Lucky 13 in OpenSSL.

OpenSSL relies on two versions of AES: software implementation provided directly by OpenSSL and hardware implementation relying on AES-NI [32].[8] For each of these AES versions, a different CBC functionality in OpenSSL is implemented. The following vulnerability was present only in the CBC functionality implemented for the hardware version of AES (AES-NI). This version is enabled by default on machines with recent CPUs. We performed our tests with OpenSSL 1.0.2g.

In order to patch the Lucky 13 attack, OpenSSL developers have introduced new code validating padding bytes and HMAC in constant-time. The code attempts to implement a very delicate CBC sanity checking countermeasure provided by AlFardan and Paterson [13]. The OpenSSL validation is implemented in methods `aesni_cbc_hmac_sha1_cipher` and `aesni_cbc_hmac_sha256_cipher` and works follows:

1. It verifies whether the ciphertext can provide enough data for HMAC and minimal padding. For example, ciphertexts consisting of 16 bytes are directly rejected by OpenSSL because they do not provide enough data for HMAC validation.

2. It decrypts the AES-CBC ciphertext into a message m of length l_m and interprets the last byte of the message as a padding byte l_{pad}. Furthermore, it internally computes the maximum padding value: $maxpad = l_m - l_{hmac}$ (where l_{hmac} represents the HMAC output length).

3. For validation purposes, it computes an HMAC $hmac'$ over the TLS record header and $l_m - l_{pad} - l_{hmac} - 1$ bytes of the decrypted message m. It constructs $hmac'||pad'$ where pad' consists of $l_{pad} + 1$ padding bytes.

4. At the end, it validates whether the last $(maxpad + l_{hmac})$ bytes of the message m are equal to $hmac'||pad'$.

[8]AES-NI is an extended set of instructions available in recent Intel and AMD CPUs.

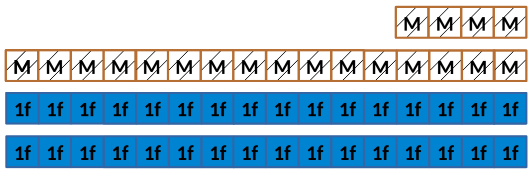


Figure 6: When OpenSSL 1.0.2g decrypts two full padding blocks, it creates an internal structure for constant-time HMAC and padding validation: $hmac'||pad'$**. However, only the padding bytes are validated.**

Unfortunately, the code is missing one important sanity check. According to [13], directly after decrypting the ciphertext in the second step, the implementation must check whether there is enough plaintext data. If $l_{hmac} + l_{pad} + 1 > l_m$, there are not enough plaintext bytes to validate both HMAC and padding. Therefore, the padding validation should run over 256 dummy padding bytes and the HMAC should be validated over the record header and the first $l_m - l_{hmac}$ bytes of m [13].

Because the OpenSSL code does not validate the decrypted message length, it is possible to completely skip the HMAC validation. Consider a vulnerable server using HMAC-SHA that processes a CBC ciphertext. The CBC ciphertext decrypts to 32 valid padding bytes `0x1F`. After the message is decrypted in the second step, the server computes $maxpad = l_m - l_{hmac} = 12$. In the third step, it computes $hmac'$ over the TLS record header and an empty message. It constructs $hmac'||pad'$ where pad' consists of 32 `0x1F` bytes. In the fourth step, it uses $hmac'||pad'$ to validate contents of the decrypted message. The validation is, however, performed only over $(maxpad + l_{hmac}) = 32$ decrypted bytes. Even though OpenSSL internally computes an HMAC value $hmac'$, this value is completely ignored during the validation process, the HMAC validation step succeeds, and the decrypted message is processed further.[9] See Figure 6.

The above described functionality obviously results in a different timing behavior. However, further message processing results even in a different TLS alert message. The decrypted message is processed in the function **`ssl3_get_record`**, which attempts to remove the padding and HMAC bytes and computes the new plain message length **`rr->length`** $= l_m - l_{hmac} - l_{pad} - 1$, which is declared as an unsigned integer. Note that in our case, the decrypted message consists of 32 bytes, $l_{hmac} = 20$ and $l_{pad} = 31$. Thus, the computation results in an integer underflow of **`rr->length`**. This underflow is further caught by the validation of the maximum plaintext length in the same OpenSSL method:

```
if (rr->length > SSL3_RT_MAX_PLAIN_LENGTH) {
        al = SSL_AD_RECORD_OVERFLOW;
        SSLerr(SSL_F_SSL3_GET_RECORD,
               SSL_R_DATA_LENGTH_TOO_LONG);
        goto f_err;
}
```

This *if branch* results in a different alert message, namely: `RECORD_OVERFLOW`. TLS-Attacker detected that this alert message differed from the typical `BAD_RECORD_MAC` alert, and reported a problem after executing the first evaluation stage.

In order to trigger the `RECORD_OVERFLOW` alert, the attacker needs to send a ciphertext which decrypts to 32 equal bytes. The attacker can exploit this behavior in specific BEAST scenarios [30] by controlling 31 bytes of the plaintext data. In comparison to the previous Botan vulnerability, the attacker is only able to recover at most 16 subsequent plaintext data at most because of the CBC mode of operation properties [52, 54].

The vulnerability was fixed after our disclosure in OpenSSL 1.0.2h / 1.0.1t. It has been labeled CVE-2016-2107.[10]

Unlucky patch of Lucky 13 in MatrixSSL.

A similar problem with patching the Lucky 13 attack could be observed in the MatrixSSL library. The developers of MatrixSSL, however, introduced a much more serious buffer overflow vulnerability while attempting to implement a Lucky 13 countermeasure.

The vulnerability has been patched after our disclosure in MatrixSSL 3.8.2.

7.2 Bleichenbacher's Attack on MatrixSSL

In 2014 Meyer et al. [44] analyzed vulnerabilities of TLS libraries to Bleichenbacher attacks. Most libraries were only vulnerable to timing attacks at that time.

In this work we could find a direct Bleichenbacher vulnerability in MatrixSSL. The vulnerable server responds with a different TLS alert (`ILLEGAL_PARAMETER`) in case the decrypted `ClientKeyExchange` message is correctly formatted but contains an invalid TLS version number. Otherwise, the server responds with a `DECRYPT_ERROR` alert. This kind of vulnerability was in a fact first described by Klima et al. in 2003 [40].

The vulnerability has been patched after our disclosure in MatrixSSL 3.8.2.

7.3 Missing Length Checks

Our analysis in phase 1 of the second stage revealed interesting results regarding the checks of different length variables. For example, GnuTLS 3.4.9 does not strictly check the length variables in the following extensions: max fragment length, elliptic curves, EC point format extension, and signature and hash algorithms extension.

If an invalid length variable is contained in one of these fields in the `ClientHello` message, GnuTLS just proceeds with the TLS handshake without further message parsing.

We assume this behavior is caused by performance optimizations included in the evaluated library. An attacker could use it for fingerprinting of TLS server library. Moreover, this behavior becomes interesting in light of the very recent SLOTH attack [20]. By performing this attack, the attacker attempts to find hash collisions for a transcript of protocol messages. Thereby, he tampers selected handshake message fields so that the messages remain valid. Not validating specific message fields gives the attacker more modification freedom and improves the attack.

[9] Note that the same behavior can be observed by sending 32 equal padding bytes *pad* where $l_{pad} >$ `0x1F`. Even though the padding is incomplete, the implementation only validates the equality of the 32 decrypted message bytes.

[10] Commit 70428eada9bc4cf31424d723d1f992baffeb0dfb: https://github.com/openssl/openssl/commit/70428eada9bc4cf31424d723d1f992baffeb0dfb

7.4 Overflows and Overreads

By fuzzing the variables in the first and second phases, we were also able to find array boundary vulnerabilities.

Stack overflow in OpenSSL-1.1.0-pre1. By fuzzing the first pre-version of the OpenSSL 1.1.0 library (OpenSSL-1.1.0-pre1), ASan reported a stack overflow vulnerability and the server crashed. Our analysis of the vulnerability revealed that the stack overflow is caused by sending an overlong DH parameter in the `DHClientKeyExchange` message. TLS-Attacker triggered this bug by a left shift of the original BigInteger value, as we found in the resulting protocol flow configuration document:

```
<workflowTrace>
 <protocolMessages>
  ...
  <DHClientKeyExchange>
   <y>
    <bigIntegerShiftLeftModification>
     <shift>41</shift>
    </bigIntegerShiftLeftModification>
   </y>
  </DHClientKeyExchange>
  ...
 </protocolMessages>
</workflowTrace>
```

When we found this vulnerability (February 2016), OpenSSL has already published a new version, OpenSSL-1.1.0-pre2, which did not include this vulnerability. The OpenSSL security team mentioned in our email correspondence that they knew about this vulnerability internally.[11] Since the security vulnerabilities in OpenSSL pre-releases are not public, we were not aware of this fact.

The vulnerability is not present in any of the newest OpenSSL release versions (1.0.2g).

Potential buffer overread in Botan 1.11.28. Furthermore, we found a buffer overread vulnerability in Botan 1.11.28. This vulnerability was found by executing the second phase of the fuzzing stage by modifying bytes in the underlying record layer:

```
<workflowTrace>
 <protocolMessages>
   ...
  <ChangeCipherSpec>
   <messageIssuer>CLIENT</messageIssuer>
   <records>
    <record>
     <maxRecordLengthConfig>0</
          maxRecordLengthConfig>
    </record>
   </records>
  </ChangeCipherSpec>
   ...
 </protocolMessages>
</workflowTrace>
```

The resulting protocol flow revealed that by sending an empty TLS record, the server attempts to use an invalid array index. The failure is triggered by a randomly selected configuration of the maximum record length, which is set to zero with `maxRecordLengthConfig`.

Even though Botan 1.11.28 does not directly verify the length of the incoming TLS records, further handshake handlers can successfully reject the message and throw an error. These subsequent verifications turn the resulting vulnerability into a rather harmless buffer overread, which only confirms the functionality of TLS-Attacker.

We reported the issue to Botan developers. It has been patched in Botan 1.11.29.

[11]The vulnerability has been fixed in the following commit: https://github.com/openssl/openssl/commit/e2b420fdd708e14a0b43a21cd2377cafb0d54c02

8. BUILDING A TLS TEST SUITE

The uncovered vulnerabilities have strongly motivated developers to build TLS test suites with negative tests to validate correct TLS behavior in specific cases. TLS negative tests could then have mitigated several vulnerabilities found in this paper. For example, a padding oracle test suite could have sent encrypted records with modified padding contents to trigger different TLS alerts. Developers introducing new functionality (e.g., Lucky 13 countermeasures) would have been warned about invalid message processing before releasing new library versions.

We have managed to extend TLS-Attacker with a `TestSuite` module allowing TLS developers to easily build positive and negative test suites. In the following section we describe the usage of assertions with our framework and present an experimental test suite for cipher suite usage across TLS protocols. Note that the TLS test suite is a work-in-progress and this section aims at describing the suitability of TLS-Attacker for this purpose.

8.1 Usage of Assertions

In order to build comprehensive TLS test suites, we have extended the TLS-Attacker functionality with assertions. In particular, we have extended modifiable variables with asserting values that allow developers to validate resulting contents of modifiable variable fields after the TLS protocol is executed.

The following listing gives an example of a TLS protocol flow containing assertions which detect the OpenSSL vulnerability:

```
<workflowTrace>
 <protocolMessages>
   ...
  <Finished>
   <messageIssuer>CLIENT</messageIssuer>
   <records>
    <Record>
     <plainRecordBytes>
      <byteArrayExplicitValueModification>
       <explicitValue>
3F 3F 3F 3F 3F 3F 3F 3F  3F 3F 3F 3F 3F 3F 3F 3F
3F 3F 3F 3F 3F 3F 3F 3F  3F 3F 3F 3F 3F 3F 3F 3F
       </explicitValue>
      </byteArrayExplicitValueModification>
     </plainRecordBytes>
    </Record>
   </records>
  </Finished>
  <Alert>
   <messageIssuer>SERVER</messageIssuer>
   <level>
    <assertEquals>2</assertEquals>
   </level>
   <description>
    <assertEquals>20</assertEquals>
   </description>
  </Alert>
 </protocolMessages>
</workflowTrace>
```

In this listing we can see a TLS protocol flow setting the explicit value of the plain padded TLS record to 32 `0x3F` bytes. Thus, it potentially triggers a `RECORD_OVERFLOW` alert in vulnerable OpenSSL implementations (CVE-2016-2107).

After executing the protocol flow, we expect to receive a TLS alert message. This TLS alert message contains assertions for the level (2) and description (20) values. This ensures that the protocol flow only succeeds if the server correctly responds with a `BAD_RECORD_MAC` alert message.

Similar test cases can be defined for other cryptographic attacks or protocol behaviors.

8.2 Experimental Test Suite for Correct Cipher Suite Handling

We have created a proof-of-concept test suite for the validation of correct cipher suite support in TLS protocols. The test suite takes the available cipher suites and checks whether they are available for correct protocol versions. For example, `TLS_RSA_WITH_AES_256_CBC_SHA256` must only be accepted in TLS 1.2 but rejected in other protocol versions.

We also executed the tests with the analyzed frameworks, detecting that Botan (1.11.30) is not standard-compliant and that it incorrectly accepts TLS 1.2 cipher suites in TLS 1.0 and 1.1 protocols. This does not pose a direct security risk. However, using secure cipher suites in older protocols could possibly undermine their security and should be correctly handled with the TLS framework.

We notified Botan developers about this issue. They intend to fix this issue in the next Botan version.

9. DISCUSSION

In this paper we showed that widely used TLS libraries can include critical vulnerabilities which can be exposed by systematic fuzzing. The case of the OpenSSL stack overflow shows that a critical vulnerability can be introduced in a specific library version, for example, by code restructuring. More interestingly, new vulnerabilities can be introduced by implementing countermeasures against known attacks. The cases of padding oracle attacks in OpenSSL and MatrixSSL showed that an incorrect countermeasure can turn a server offering a timing oracle into a direct oracle or even into a server with a buffer overflow vulnerability.

We conclude that similar fuzzing technologies, as introduced in our paper, should be included into library test suites and should be included into continuous integration, or should be run before the new library versions are released. MatrixSSL and Botan libraries are in a phase of including TLS-Attacker into their test suite.

In future work, TLS-Attacker fuzzing could be extended to include new fuzzing strategies or compile-time instrumentation to observe new code paths at runtime. The concepts of TLS-Attacker fuzzing could also be extended to validate TLS clients or even to test security protocols beyond TLS, e.g., IPSec or SSH.

Acknowledgments

We would like to thank Hanno Böck, Robert Gawlik, Thorsten Holz, Tibor Jager, René Korthaus, Felix Lange, Christian Mainka, Robert Merget, Daniel Neus, Florian Pfützenreuter, Philip Riese, Jörg Schwenk, and Aaron Zauner for their helpful discussions and additional contributions. We also extend our thanks to our reviewers for their insightful comments and suggestions.

This work was partially supported by the European Commission through the FutureTrust project (grant 700542-Future-Trust-H2020-DS-2015-1) and by the Federal Office for Information Security through project 197 (Sichere Implementierung einer allgemeinen Kryptobibliothek).

10. REFERENCES

[1] Botan: Crypto and TLS for C++11. http://botan.randombit.net/.

[2] Gnutls security advisory. http://www.gnutls.org/security.html.

[3] Java Architecture for XML Binding. https://jaxb.java.net/.

[4] Java platform debugger architecture. http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/.

[5] Java Secure Socket Extension (JSSE). https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html.

[6] matrixSSL. Compact Embedded SSL/TLS stack. http://www.matrixssl.org/.

[7] mbed TLS. https://tls.mbed.org/.

[8] OpenSSL – Cryptography and SSL/TLS Toolkit. https://www.openssl.org.

[9] OpenSSL security advisory. https://www.openssl.org/news/vulnerabilities.html.

[10] The GnuTLS Transport Layer Security Library. http://www.gnutls.org.

[11] Albrecht, M. R., and Paterson, K. G. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I* (2016), pp. 622–643.

[12] AlFardan, N., and Paterson, K. Plaintext-Recovery Attacks Against Datagram TLS. In *Network and Distributed System Security Symposium (NDSS 2012)* (Feb. 2012).

[13] AlFardan, N. J., and Paterson, K. G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *2013 IEEE Symposium on Security and Privacy 0* (2013), 526–540. http://www.isg.rhul.ac.uk/tls/TLStiming.pdf.

[14] Almeida, J. B., Barbosa, M., Barthe, G., and Dupressoir, F. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE* (2016).

[15] Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J. A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., and Shavitt, Y. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)* (Aug. 2016).

[16] Bardou, R., Focardi, R., Kawamoto, Y., Steel, G., and Tsay, J.-K. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO* (2012), Canetti and R. Safavi-Naini, Eds.

[17] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y., and Zinzindohoue, J. K. A messy state of the union: taming the composite state machines of TLS. In *IEEE Symposium on Security & Privacy 2015 (Oakland'15)* (2015), IEEE.

[18] Beurdouche, B., Delignat-Lavaud, A., Kobeissi, N., Pironti, A., and Bhargavan, K. FLEXTLS: A Tool for Testing TLS Implementations. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (Washington, D.C., Aug. 2015), USENIX Association.

[19] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Pironti, A., and Strub, P.-Y. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *IEEE Symposium on Security & Privacy 2014 (Oakland'14)* (2014), IEEE.

[20] Bhargavan, K., and Leurent, G. Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '16)* (Feb 2016).

[21] Biehl, I., Meyer, B., and Müller, V. Differential fault attacks on elliptic curve cryptosystems. In *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 2000), CRYPTO '00, Springer-Verlag, pp. 131–146.

[22] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and Moeller, B. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. Updated by RFCs 5246, 7027.

[23] Bleichenbacher, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998.

[24] Böck, H. A little POODLE left in GnuTLS (old versions), Nov. 2015. https://blog.hboeck.de/archives/877-A-little-POODLE-left-in-GnuTLS-old-versions.html.

[25] de Ruiter, J., and Poll, E. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 193–206.

[26] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507.

[27] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[28] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. draft-ietf-tls-tls13-04, Jan. 2015.

[29] Dietz, W., Li, P., Regehr, J., and Adve, V. Understanding integer overflow in c/c++. In *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE 2012, IEEE Press, pp. 760–770.

[30] Duong, T., and Rizzo, J. Here come the $\oplus$ Ninjas. Unpublished manuscript, 2011.

[31] Eastlake, D. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard), Jan. 2011.

[32] Gueron, S. Intel Advanced Encryption Standard (AES) New Instructions Set, Revision 3.01, 2012.

[33] Haller, I., Slowinska, A., Neugschwandtner, M., and Bos, H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 49–64.

[34] Haugh, E. Testing c programs for buffer overflow vulnerabilities. In *In Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2003).

[35] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, ACM, pp. 85–96.

[36] Jager, T., Schinzel, S., and Somorovsky, J. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML Encryption. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-14, 2012. Proceedings* (2012), S. Foresti and M. Yung, Eds., LNCS, Springer.

[37] Jager, T., Schwenk, J., and Somorovsky, J. Practical Invalid Curve Attacks on TLS-ECDH. *20th European Symposium on Research in Computer Security (ESORICS)* (2015).

[38] Kario, H. Testing TLS. Ruxcon, Oct. 2015. https://github.com/tomato42/tlsfuzzer.

[39] Kikuchi, M. CCS Injection Vulnerability, 2014. http://ccsinjection.lepidum.co.jp.

[40] Klíma, V., Pokorný, O., and Rosa, T. Attacking RSA-Based Sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, vol. 2779 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Sept. 2003.

[41] Langley, A. The POODLE bites again, Nov. 2014. https://www.imperialviolet.org/2014/12/08/poodleagain.html.

[42] Merkle, J., and Lochter, M. Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS). RFC 7027 (Informational), Oct. 2013.

[43] Meyer, C. *20 Years of SSL/TLS Research : An Analysis of the Internet's Security Foundation.* PhD thesis, Ruhr-University Bochum, Feb. 2014.

[44] Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., and Tews, E. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security Symposium, San Diego, USA* (August 2014).

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

[46] Moneger, A. Penetration Testing Custom TLS Stacks. ShmooCon, Feb. 2016. https://github.com/tintinweb/scapy-ssl_tls.

[47] Ray, M., and Dispensa, S. Renegotiating TLS. Tech. rep., PhoneFactor, Inc., Nov. 2009.

[48] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), Jan. 2012. Updated by RFC 7507.

[49] Riku, Antti, Matti, and Mehta. Heartbleed, cve-2014-0160, 2015. http://heartbleed.com/.

[50] Seggelmann, R., Tuexen, M., and Williams, M. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520 (Proposed Standard), Feb. 2012.

[51] Sheffer, Y., Holz, R., and Saint-Andre, P. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457 (Informational), Feb. 2015.

[52] Somorovsky, J. Curious Padding oracle in OpenSSL (CVE-2016-2107). http://web-in-security.blogspot.de/2016/05/curious-padding-oracle-in-openssl-cve.html.

[53] Sullivan, N. The results of the cloudflare challenge. https://blog.cloudflare.com/the-results-of-the-cloudflare-challenge.

[54] Valsorda, F. Yet Another Padding Oracle in OpenSSL CBC Ciphersuites. https://blog.cloudflare.com/yet-another-padding-oracle-in-openssl-cbc-ciphersuites.

[55] Vaudenay, S. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology — EUROCRYPT 2002*, vol. 2332 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Apr. 2002.