UNIVERSITÄT PASSAU

Institute of IT-Security and Security Law

Master Thesis

Penetration Testing Framework for OCSP-Responders

Stefan Kunz

University of Passau
Faculty of Computer Science and Mathematics

Chair of IT Security
Prof. Dr. rer. nat. Joachim Posegga

Date: April 26, 2018

Supervisors: Prof. Dr. rer. nat. Joachim Posegga
Prof. Dr. Hans P. Reiser
Dr. Eric-Jean Knauel

Erklärung zur Master Thesis

Name, Vorname des
Studierenden: Kunz, Stefan

Universität Passau,
Fakultät für Informatik und Mathematik

Hiermit erkläre ich, dass ich die Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate auch als solche gekennzeichnet habe.

....
(Datum) (Unterschrift des Studierenden)

Supervisor Contacts:

Prof. Dr. rer. nat. Joachim Posegga
Chair of IT Security
University of Passau
Email: posegga@uni-passau.de
Web: `https://web.sec.uni-passau.de/`

Prof. Dr. Hans P. Reiser
Chair of Security in Information Systems
University of Passau
Email: hans.reiser@uni-passau.de
Web: `http://www.uni-passau.de/sis/`

Dr. Eric-Jean Knauel
ESCRYPT GmbH
Email: EricJean.Knauel@escrypt.com
Web: `https://www.escrypt.com/`

Abstract

Although the reliability of OCSP responders is extremely important for certificate revocation checks in PKIs, no standardized penetration testing solution for them exists yet. Therefore, the focus of this thesis is the search for OCSP's design and implementation related vulnerabilities.
For this purpose a custom black-box fuzzer based on boofuzz is developed. It enables automatic testing of OCSP responders to reveal potential exploitable vulnerabilities using its main generation-based input creation model or its additional mutation-based model. With the fuzzer's monitoring tools the fuzzing run can be monitored and logs are produced for later analysis.
This is successfully tested in a fuzzing campaign, which proofs the applicability of the custom fuzzer on arbitrary OCSP responders. The fuzzing campaign also shows, that the generation-based model is better applicable for a complex protocol like OCSP and it reveals potential exploitable implementation related security vulnerabilities within the audits created for the different targets.

Contents

1. Introduction

1.1. Motivation

OCSP responders are crucial building blocks of Public Key Infrastructures (PKIs), because they enable clients the verification of a certificate's current revocation status. In order to avoid information disclosure by revoked certificates, this must be possible without any interruption. Therefore, a high reliability is essential for OCSP responders.
The current standard are X.509 certificates issued and signed by trusted Certification Authorities (CAs) for their customers to establish trusted certificate chains. But if a certificate owner's private key gets compromised, its certificate has to be revoked immediately by its issuing CA to prevent the abuse of the compromised private key. That other devices in a network can be informed about revoked certificates, the X.509 standard suggests as mechanisms the Online Certificate Status Protocol (OCSP) and Certificate Revocation Lists (CRLs) [1].
CRLs are lists containing the serial number, revocation date and reason for every revoked certificate issued by a CA. That they cannot be altered, the lists are digitally signed by the CA or a CRL issuer assigned by the CA and made publicly available. Then a client can determine a certificate's revocation status by acquiring a sufficiently recent CRL and checking, if the checked certificate's serial number is not listed in the CRL. Disadvantage of this approach is, that revoked certificates can never be removed from a CRL. Their revocation information has to be kept in the CRL even when their validity duration is exceeded to enable for instance the verification of electronic signatures for which the certificate's revocation status at the time of the signature's computation is relevant. Thereby CRLs are not scaling well, because for checking a certificate's revocation status a client always has to parse the whole CRL, which grows with every revoked certificate.
The alternative is the Online Certificate Status Protocol (OCSP), which "is a lightweight stateless one-step request-response protocol" [2, p. 2]. Compared to CRLs, which always have to be parsed completely by the clients, it enables clients to query an up-to-date status of single certificates directly from their responsible OCSP responder. Again, the checked certificates are identified by their serial number and the OCSP responder's response is authorized by its signature, which also ensures the integrity of the response. OCSP reduces the network load required to check the status of single certificates, because the clients don't have to retrieve the whole CRL. It is sufficient for them to parse the OCSP response instead of the whole

CRL, which reduces their computational effort. This is especially in an environment with constraint devices, like the Internet of Things (IoT), an important aspect.
All in all for certificate revocation OCSP is important, because it enables different devices in a network to deal with compromised keys, while providing a better scalability than CRLs.

But CRLs and OCSP are not only essential to check a certificate's current revocation status. They are also mandatory for long-term storage of certificate based electronic signatures. The technical guideline 03125 of the German Federal Office for Information Security (BSI) prescribes, that for a permanent and traceable proof of an electronic signature's authenticity its verification data has to be preserved to be able to proof, that the user certificate used to sign the document was valid at the time of the signature's creation. This proof-relevant data consists of the signature, its time-stamp, the user certificate used to create the signature and all certificates in the user certificate's certificate chain up to the root certificate with their associated CRLs or OCSP responses at the time of the signature's creation.
This creation and validation of an X.509 certificate's certificate chain can also be outsourced to the Server-Based Certificate Validation Protocol (SCVP), which differs from OCSP in that it sends two messages. One to request the server's supported validation policies and a second one with the certificate ids to check, but SCVP is relatively new and therefore currently only supported by a few applications [3].

1.2. Problem Definition

The important role of OCSP responders in the certificate revocation process of PKIs was outlined in the previous section. This implies a high functional and fail safety, that a certificate's status can be retrieved reliably.
But Koschuch and Wagner figured out in 2015, that "almost 85% of HTTPS certificates don't contain any revocation information" [4, p. 32] based on a dataset of $66,335,624$ HTTPS certificates collected by Durumeric et al. [5] between 2012 and 2013. This means, that the `AuthorityInformationAccess` extension containing CRL or OCSP information was not defined for these certificates [6].
Also they found out, that most of their tested client applications for checking the revocation status of certificates just implemented a soft-fail approach at this time. Thereby every certificate is accepted, if the OCSP responder or the CRL are unavailable, which enables attacks just by blocking the communication with the OCSP responder. This additionally underlines, that the reliability of revocation information providers has to be further improved to increase the client applications' trust.

Especially if assumption 1 is considered, which was introduced by myself, it comes clear, that penetration testing is necessary to be able to investigate and proof assumption 1 by the identification of likely existing vulnerabilities before they can be detected and exploited by attackers. Thereby an application's security level can be increased and a better understanding of its risk is gained.

Assumption 1 *All non-trivial systems built by humans contain unknown bugs, which are exploitable vulnerabilities.*

Unfortunately for OCSP responders no standardized penetration testing solution exists yet. For this reason in this thesis a penetration testing framework specialized on OCSP responders was designed and tested with focus on fuzzing. Fuzzing was chosen as testing methodology, because it already proved in the past to successfully identify vulnerabilities. For example in 2017 Domas [7] wanted to validate a processor's instruction set to stop treating a system's most trusted component as trusted black-box. Due to the lack of publicly available and efficient introspection tools, he also built a tool based on black-box fuzzing. Its success was successfully demonstrated by exhaustively searching the x86 instruction set and revealing security-critical bugs in the x86 hardware.
Therefore to also verify the effectiveness and applicability of the framework developed in this thesis, a fuzzing campaign was rolled out on existing OCSP responders. Its results were used to proof assumption 1.

1.3. Thesis Structure

Chapter 2 explains the fundamentals of this thesis starting with a detailed explanation of ASN.1's Distinguished Encoding Rules (DER) in chapter 2.1. They are required to encode OCSP. Afterwards, chapter 2.2 introduces OCSP and its extensions, before fuzzing and already existing open-source frameworks to build a fuzzer are covered by chapter 2.3. Chapter 2.4 completes the fundamentals of this thesis with a threat model of OCSP's design related vulnerabilities.
Then in chapter 3 the fuzzing targets, the fuzzing approach and the fuzzing framework used as base to build the custom OCSP fuzzer are selected.
Based on these selections chapter 4 then describes the design and the implementation of the custom OCSP responder, which was developed in this thesis.
Afterwards, this fuzzer was used to roll out a fuzzing campaign, whose results are discussed in chapter 5.
Chapter 6 then tells, which works already were available in the domain of this thesis and how it can be continued.
Finally, chapter 7 summarizes the results of this thesis and mentions its contributions, as well as the limitations of the chosen approach.

2. Background

For a better understanding of the fundamentals of this thesis, they are clarified in this chapter. First ASN.1 and its Distinguished Encoding Rules (DER) are introduced, before the Online Certificate Status Protocol (OCSP) is explained. It is specified in ASN.1. Therefore an understanding of ASN.1's Distinguished Encoding Rules (DER) is mandatory to be able to encode the fuzzed input.
Afterwards follows an introduction to fuzzing and a threat model explaining OCSP's known and expected vulnerabilities.

2.1. ASN.1 DER

The Abstract Syntax Notation One (ASN.1) "was designed for modeling efficiently communications between heterogeneous systems" [8, p. 2]. This is achieved by the definition of an abstract data syntax for which each ASN.1 compiler implements a concrete language specific syntax, which it can encode into a bit-based transfer syntax. By mapping the abstract values to single byte[1] strings and vice versa the encoded data can be compared without knowing its ASN.1 syntax.
Especially for the use with digital signatures the Distinguished Encoding Rules (DER) were defined. They are a subset of ASN.1's Basic Encoding Rules (BER) giving every ASN.1 value a unique byte string encoding on which the digital signature can be computed [9].
Every ASN.1 value consists of a tag-length-value triplet (TLV triplet). Its encoding is explained in the following subsections.

[1]Bytes are similar to octets, therefore both designations are used in literature.

2.1.1. Tag

An ASN.1 tag's DER encoding is illustrated in fig. 2.1. Bit 7 and 6 are determined by the encoded ASN.1 value's class, which is `UNIVERSAL` if it was not specified differently. Bit 5 depends on the ASN.1 value's form, where string types are encoded in primitive form.
The remaining bits are destined for the tag number's encoding. Tag numbers up to 30 can be encoded using the low-tag-number form, which is the tag's binary value padded with leading 0-bits to a length of 5.
Greater tag numbers have to be encoded with the high-tag-number form for which the leading byte is filled with 1-bits and for the following bytes bit 7 is set to 1 down to the last one for which bit 7 is set to 0. The value of the single following bytes is computed by adding leading 0-bits to the tag number's binary value until its length is a multiple of seven and then dividing it into 7-bit blocks.

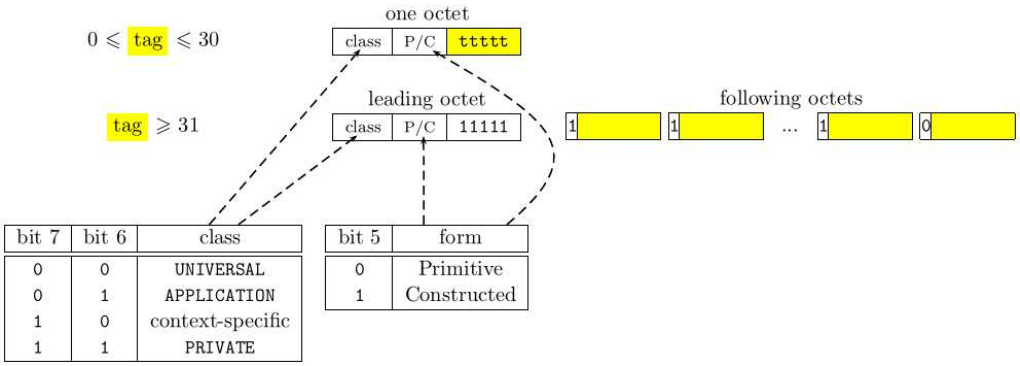


Figure 2.1.: ASN.1 DER tag encoding (image adapted, source: [8, p. 424])

IMPLICIT and EXPLICIT Tags

A special case of tags are the additional `IMPLICIT` and `EXPLICIT` tags, which can be specified for an element like in listing 2.1. They are required to unambiguously identify elements contained in non-ordered constructed types[2] (`SET`) and meta types (`CHOICE`), or if ambiguities arise due to `OPTIONAL` elements contained in a structure. Per default these additional tags are always `context-specific`, if no class is specified, but `context-specific` tags are only allowed in constructed or `CHOICE` types. Also an additional tag's type (`IMPLICIT` or `EXPLICIT`) can be omitted in an ASN.1 specification, if a default tagging method was defined.
`IMPLICIT` tags replace an element's default tag determined by its type and are rendered in the form of the element's default tag. This can destroy information and therefore they are not allowed for meta types like `CHOICE` or `ANY`.
`EXPLICIT` tags on the other hand complement the element's original tag. They cre-

[2]Constructed types are also sometimes mentioned as structured types in literature.

ate a TLV wrapper structure around it with the original TLV triplet set as value and are always rendered in constructed form [9].

```
1 [[class] number] [IMPLICIT | EXPLICIT] Type
2 class = UNIVERSAL | APPLICATION | PRIVATE
```

Listing 2.1: `IMPLICIT` and `EXPLICIT` ASN.1 tag notation

2.1.2. Length

An ASN.1 value's length field contains the number of bytes required to encode its value field. Dependent on its value the length field is encoded in one of the following forms:

- **short length notation** ($0 \leq \text{length} \leq 127$): The length is encoded in one byte like displayed in fig. 2.2. Bit 7 is set to 0 and the remaining bits identify the length's value.

Figure 2.2.: ASN.1 DER short length notation (image adapted, source: [8, p. 424])

- **long length notation** ($0 \leq \text{length} \leq 256^{126} - 1$): Multiple bytes are required to encode the length like displayed in fig. 2.3. Bit 7 of the first byte is set to 1 and the first byte's remaining bits identify the number of bytes required to store the length's binary value.
  Just 1-bits in the first byte are not allowed. Thereby the maximal number of bytes to encode the length's binary value is 126, resulting in $256^{126} - 1$ bytes as upper limit for the encoded value field's length [8, 10].

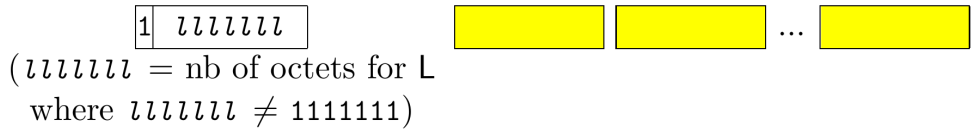


Figure 2.3.: ASN.1 DER long length notation (image adapted, source: [8, p. 424])

2.1.3. Value

Apart from the meta types, every ASN.1 type is associated with a `UNIVERSAL` tag number and optionally `SIZE` constraints can be applied to it. In the following different OCSP types are introduced, which are required to model an OCSP request. They are grouped by their encoding's form, sorted by their `UNIVERSAL` tag number in decimal form and their encoding is explained with an example.

Primitive Types

BOOLEAN values have the **tag value 1**. Their two possible encodings are displayed in listing 2.2 [11].

```
01 01 ff ; BOOLEAN, length: 1 byte, True
01 01 00 ; BOOLEAN, length: 1 byte, False
```

Listing 2.2: ASN.1 DER `BOOLEAN` encoding

INTEGER values have the **tag value 2**. An example of their encoding is given in fig. 2.4. Their value is directly encoded to a byte string, if it is positive, otherwise the value's two's complement is encoded. Therefore like in fig. 2.4 for the encoding of positive values, where the high order bit is 1, a leading `0x00` byte is added to mark the value as non-negative [12].

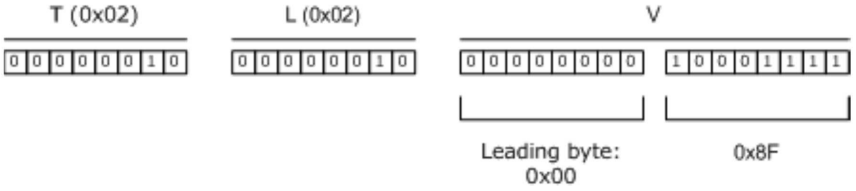


Figure 2.4.: ASN.1 DER `INTEGER` encoding [12]

BIT STRING values have the **tag value 3**. An example of their encoding is given in fig. 2.5. Their encoding's first byte contains the unused bits in the final content byte [13]. Trailing 0-bits are just encoded, if `SIZE` constraints for the type exist [8].

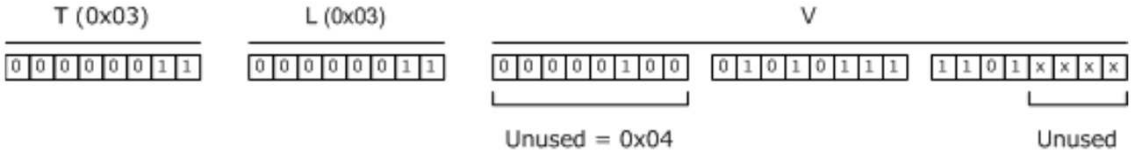


Figure 2.5.: ASN.1 DER `BIT STRING` encoding [13]

OCTET STRING values have the **tag value 4**. An example of their encoding is given in listing 2.3. They are encoded like `BIT STRING` values expect, that their trailing byte cannot have unused bits. Therefore no leading byte containing the unused bits is encoded.

```
04 0a                                   ; OCTET STRING, length: 10 bytes
|  1e 08 00 55 00 73 00 65 00 72 ; ...U.s.e.r
```

Listing 2.3: ASN.1 DER `OCTET STRING` encoding [14]

NULL values have the **tag value 5**. They have no value byte. Therefore they are always encoded like in fig. 2.6.

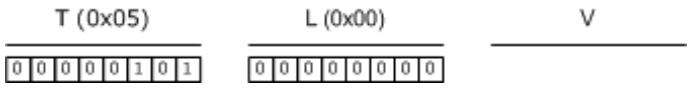


Figure 2.6.: ASN.1 DER `NULL` encoding [15]

OBJECT IDENTIFIER values have the **tag value 6**. An example of their encoding is given in listing 2.4.

```
06 05             ; OBJECT IDENTIFIER, length: 5 bytes
|  2b 0e 81 b0 1a ; 1.3.14.22554
```

Listing 2.4: ASN.1 DER `OBJECT IDENTIFIER` encoding

The `OBJECT IDENTIFIER`'s sub identifiers (SIDs) consist of unsigned integers, which are separated by a '.'. The first two of them are encoded together, therefore the following additional restrictions apply for them [16]:

$$\forall\ \mathrm{SID}_0 : \mathrm{SID}_0 \in [0, 1, 2]$$
$$\forall\ \mathrm{SID}_0 \in [0, 1] : \mathrm{SID}_1 \leq 39$$

According to the sample in listing 2.4 the composed value of the first two SIDs is computed like this:

$$\mathrm{SID}_{0+1} = \mathrm{SID}_0 \times 40 + \mathrm{SID}_1 = 1 \times 40 + 3 = 43$$

Dependent on their value the single SID values $(\mathrm{SID}_{0+1}, \mathrm{SID}_2, \mathrm{SID}_3)$ then are encoded into one or multiple bytes [17]:

- **1 byte** (SID ≤ 127): $\mathrm{SID}_{0+1} = 43\mathrm{d} =$ `0x2b`
- **multiple bytes** (SID > 127): $\mathrm{SID}_3 = 22554\mathrm{d}$

  1. Convert 22554 to base 128:

$$\begin{aligned} 22554 \ /\ 128 &= 176 \quad \mathrm{R}\ 26 \\ 176 \ /\ 128 &= 1 \quad \mathrm{R}\ 48 \\ 1 \ /\ 128 &= 0 \quad \mathrm{R}\ 1 \end{aligned}$$

  $\rightarrow$ termination on quotient $= 0 \rightarrow$ result: 1 48 26 = `0x01 0x30 0x1a`

  2. XOR all resulting bytes except the last one with `0x80` $\rightarrow$ `0x81 0xb0 0x1a`

String Types

The following string types are also encoded in primitive form. Their alphabets' character ranges and also the bytes required to encode a character differ. Examples of their encodings, which require a different number of bytes per character, are given in listing 2.5.

```
16 04 ; IA5String, length: 4 bytes
|  55 73 65 72                                        ; User
1e 08 ; BMPString, length: 8 bytes
|  00 55 00 73 00 65 00 72                           ; .U.s.e.r
1c 10 ; UniversalString, length: 16 bytes
|  00 00 00 55 00 00 00 73 00 00 00 65 00 00 00 72 ; ...U...s...e...r
```

Listing 2.5: ASN.1 DER string encoding

UTF8String values have the **tag value 12**. They can represent every universal character with a variable-length character encoding of 1-4 bytes [16].

PrintableString values have the **tag value 19**. They are limited to the character set of mainframe input terminals (`A-Za-z0-9'()+,-./:=?<space>`), which is encoded with 1 byte per character [18].

TeletexString values have the **tag value 20**. Their alphabet contains 308 characters, which are encoded in 1 byte or 2 bytes for composed characters [8].

IA5String values have the **tag value 22**. Their International Alphabet number 5 (IA5) is similar to the 128 character ASCII alphabet (`0x00-0x7f`), but can contain additional regional language specific characters [18].

UniversalString values have the **tag value 28**. They cover all 17 Universal Character Set (UCS) planes in which characters are referenced by the quadruple {`group, plane, row, cell`}, resulting in 4 bytes required to encode a single character. Currently just the first plane is used, therefore the first two bytes are normally rendered to null-bytes [8].

BMPString values have the **tag value 30**. They cover the first 0 plane of the UCS resulting in 2 bytes required to encode a character and an alphabet size of $65,536$ characters (`0x0000-0xffff`) [16]. Because currently only the first UCS plane is allocated, this value type should be preferred over `UniversalString`s due to its more compact encoding [8].

GeneralizedTime values have the **tag value 24** and represent time values, which are DER encoded into character strings of this format: `yyyyMMDDHHmmss.SSSZ`
An example of possible encodings is given in listing 2.6. Special for DER is, that the seconds (`ss`) are not compulsory. Also meaningless zeros in the fractions of seconds (`SSS`) must not be encoded and the decimal dot's encoding is only allowed, if fractions exist. Additionally DER requires the time value to be always in Universal Time Coordinate (UTC) (`Z`) [8].

```
18 13 ; GeneralizedTime, length: 19 bytes                    ; yyyyMMDDHHmmss.SSSZ
|  32 30 31 37 31 32 32 32 31 35 32 33 33 33 2e 31 30 33 5a ; 20171222152333.103Z
18 12 ; GeneralizedTime, length: 18 bytes
|  32 30 31 37 31 32 32 32 31 35 32 33 33 33 2e 31 33 5a    ; 20171222152333.13Z
18 0f ; GeneralizedTime, length: 15 bytes
|  32 30 31 37 31 32 32 32 31 35 32 33 33 33 5a             ; 20171222152333Z
18 0d ; GeneralizedTime, length: 13 bytes
|  32 30 31 37 31 32 32 32 31 35 32 33 5a                   ; 201712221523Z
```

Listing 2.6: ASN.1 DER GeneralizedTime encoding

Constructed Types

Constructed types have structured values consisting of on or more ASN.1 types. If single contained types are specified as `OPTIONAL`, the ASN.1 compiler is allowed to omit them during encoding.
Also `DEFAULT` values can be defined for the types contained in the structure. When a specified type is not contained in the structure's encoding, then this `DEFAULT` value is assigned to it. Therefore according to DER a field must not be encoded, if its default value is assigned to it [9].

A **SEQUENCE** has the **tag value 16** and is "an ordered collection of one or more types" [9]. Its value is rendered by concatenating the contained fields' rendered values in the order of their definition.

A **SEQUENCE OF** also has the **tag value 16** and is an ordered collection of zero or more occurrences of the same type [9]. Its value is rendered in the same way as the `SEQUENCE`'s value.

A **SET** has the **tag value 17** and is "an unordered collection of one or more types" [9]. For rendering its value the contained fields' rendered values can be concatenated in any order.

A **SET OF** also has the **tag value 17** and is an unordered collection of zero or more occurrences of the same type [9]. Its value is rendered in the same way as the `SET`'s value.

Meta Types

Meta types are not encoded themselves. Only their contained or resulting values are encoded. Because of this they are special and have no tag and length values. Therefore additional implicit tags are not allowed for them.

A **CHOICE** is "a union of one or more alternatives" [9]. Its encoding can be the encoded TLV triplet of any of its alternatives, whose encoding can be of primitive or constructed form.

An "**ANY** type denotes an arbitrary value of an arbitrary type" [9]. This type normally depends on an associated `INTEGER` or `OBJECT IDENTIFIER` value.

2.2. Online Certificate Status Protocol (OCSP)

The Online Certificate Status Protocol (OCSP) is specified in in RFC 6960 of the Internet Engineering Task Force (IETF) [19]. It is a lightweight stateless network protocol, allowing clients to request a X.509 certificate's current revocation status of an OCSP responder [2]. This chapter first explains the protocol's request-response flow followed by its extensions.

2.2.1. Request-Response Flow

The protocol's one-step request-response flow is visualized in figure 2.7. Its exchanged data is specified in ASN.1 and can be looked up in appendix A.
If a client wants to verify a certificate's status its OCSP responder's location has to be configured locally at the client or it has to lookup the OID of the `accessMethod` out of the certificate's `AuthorityInformationAccess` extension. The OID `id-ad-ocsp` is used, if revocation information is provided by OCSP. Then the OCSP responder's Uniform Resource Identifier (URI) to request can be read from the extension's `accessLocation` [6, 19].

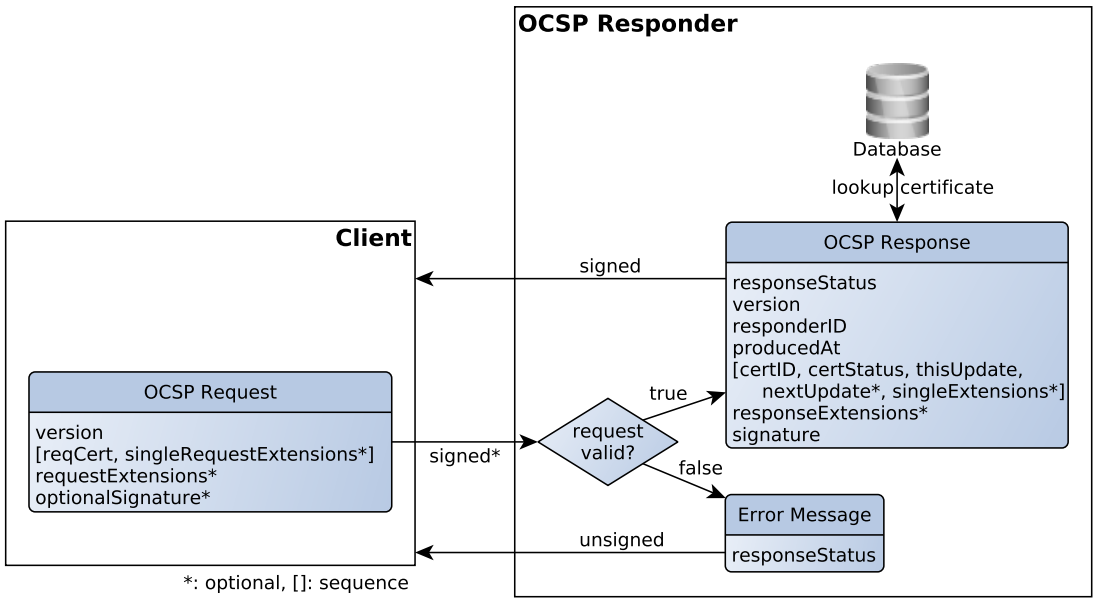


Figure 2.7.: OCSP Sequence Diagram

OCSP Request

An overview of the `OCSPRequest` is given in its class diagram in fig. 2.8. Its `tbsRequest` explicitly contains the used protocol `version` and in the `requestList` the single certificate status requests. Each of them contains an identifier (`CertID`) of its requested target certificate (`reqCert`) consisting of the certificate's `serialNumber`, which is unique within its issuer also specified in the `CertID`. Additionally, optional extensions can be defined for the surrounding `TBSRequest` (`requestExtensions`) and per certificate `Request` (`singleRequestExtensions`), but it is not guaranteed, that they are processed by the OCSP responder. Optionally the `OCSPRequest` can also be signed by including its `optionalSignature`. It is computed on the `tbsRequest` after it was encoded using ASN.1's DER [19].

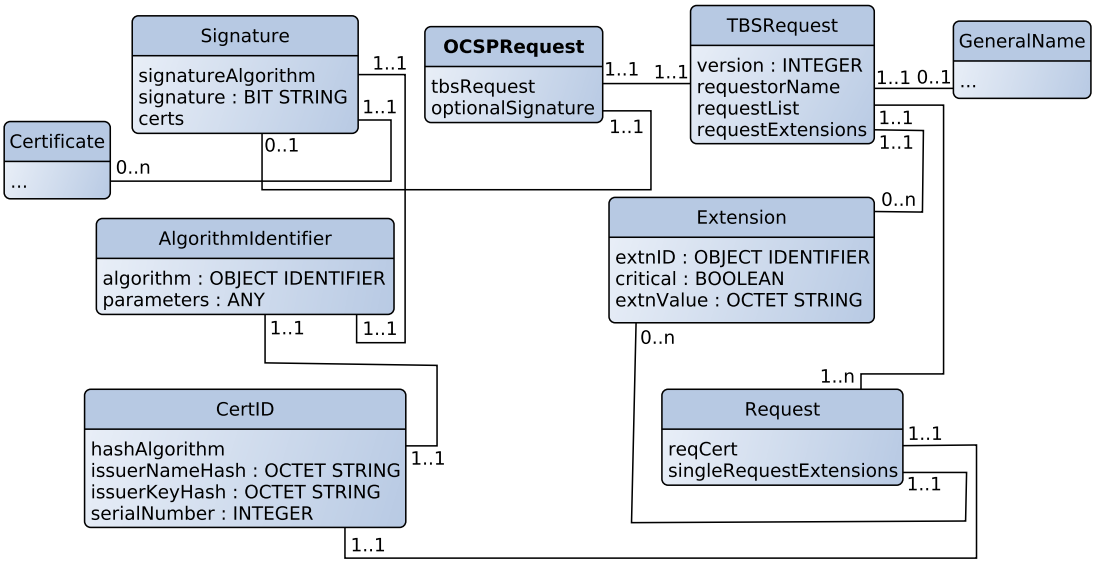


Figure 2.8.: OCSP Request Class Diagram

OCSP Response

When the OCSP responder receives a request, it first checks its validity as shown in the sequence diagram in fig. 2.7. If the syntax is correct, the requested service is provided and all required information is contained in the request, it can be answered. Otherwise an error response is sent. An overview over of both response types gives the `OCSPResponse`'s class diagram in fig. 2.9.

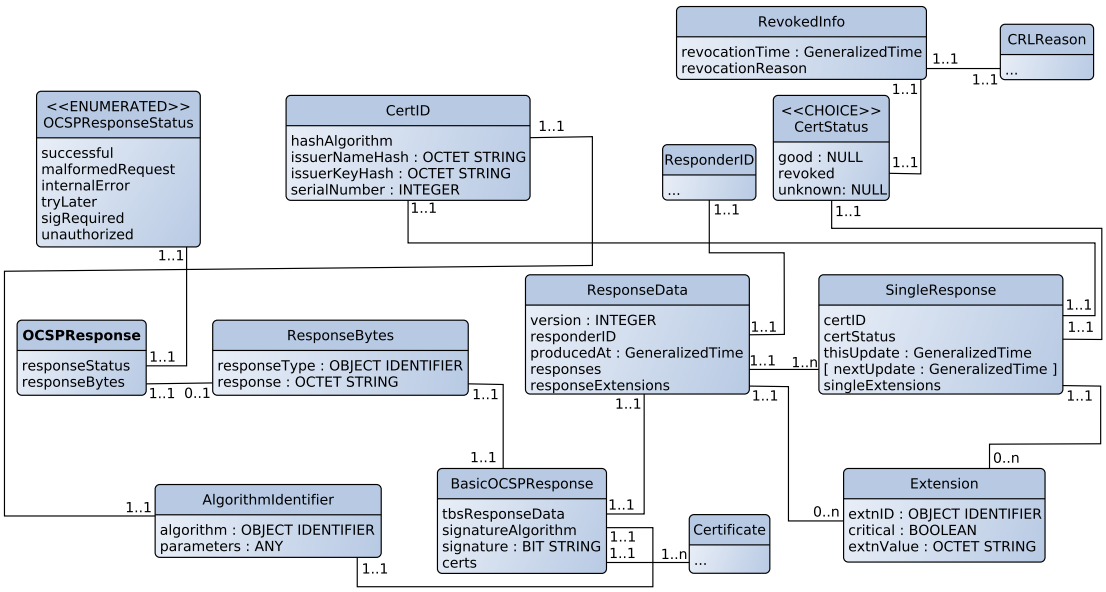


Figure 2.9.: OCSP Response Class Diagram

Error responses are unsigned, which means they do not contain a `BasicOCSPResponse`, which is always signed. Also they have an appropriate error `responseStatus` set in the `OCSPResponse`, which is different from `successful`.
For answering a request, the responder looks up the requested certificates in its database to generate the `ResponseData`. Then the `responseStatus` `successful` is set for the `OCSPResponse` and within the `ResponseData` the used response syntax `version`, the responder's identifier (`responderID`) and the time at which the response was signed (`producedAt`) is specified.
The `responses` contained in the `ResponseData` must include for every requested certificate a `SingleResponse`, which contains the certificate's identifier (`certID`) and its current revocation status (`certStatus`). The status `good` indicates, that the requested certificate was not revoked during its validity period. `Revoked` is used, when the certificate identified by its `certID` was revoked or never issued by the associated CA. To distinguish both cases the `RevokedInfo` has to be defined for this status specifying `revocationTime` and `revocationReason`. Additional CRL extensions are not allowed for this status. The last status is the `unknown` status. It means, that the certificate's issuer is unknown or not served by the responder.
The validity interval is also contained for every `SingleResponse` in form of the most recent time at which the status was known by the responder (`thisUpdate`) and the time at which newer information will be available (`nextUpdate`). `NextUpdate` is optional. If it is not set it is assumed, that newer revocation status information is available every time. And again optional extensions can be defined for the surrounding `ResponseData` (`responseExtensions`) and for every requested certificate's `SingleResponse` (`singleExtensions`).

But unlike for error responses, the `BasicOCSPResponse` just defined for complete responses includes a signature. It is computed on the `tbsResponseData` after it was encoded using ASN.1's DER. For this task the key of the CA, who issued the questioned certificates, a locally configured trusted responder certificate or a designated responder whose certificate was issued directly by the CA shall be used. In the last case then the `OCSPSigning` extension[3] shall be defined for the responder's certificate [19].

2.2.2. Benefits over CRLs

All in all compared with CRLs OCSP responders provide more timely revocation status information and also additional status information can be obtained [19]. Furthermore, OCSP consumes less bandwidth and its clients can be more constraint, because they do not need to retrieve and parse the full CRL [4].

2.2.3. Extensions

The request-response flow of OCSP introduced in section 2.2.1 and the more detailed ASN.1 specification of OCSP's messages in appendix A show, that for the exchanged messages extensions can be defined optionally per certificate request, certificate response and for the surrounding OCSP messages. Important is the `nonce` extension, which appends a nonce in form of an `OCTET STRING` to the message sent to prevent replay attacks. Its ASN.1 specification and also the ones of additional extensions can be found in RFC 6960 [19]. The extensions' OIDs are listed in appendix A.7. Table 2.1 gives an overview about them:

[3] `OCSPSigning` extension: OID=`id-kp-OCSPSigning`, see appendix A.7

OID=id-pkix-ocsp-...	Purpose
nonce	Prevent replay attacks.
crl	Response extension indicating the CRL on which the revoked certificate was found.
response	Request extension enabling the client to specify its supported response types.
archive-cutoff	Response extension containing the archive cutoff date. For certificates expired after this date, revocation information is still available.
service-locator	Response extension to identify the certificate's authoritative OCSP server.
pref-sig-algs	Request extension enabling the client to specify its non-binding list of supported signature algorithms to increase the probability of compatibility.
extended-revoke	Response extension mandatory for the whole response, if the `revoked` status is returned for non-issued certificates.

Table 2.1.: OCSP Extensions

Furthermore, the German Common PKI Profile defines also the OCSP CertHash extension[4], which is a `SingleResponse` extension including the hash of the requested certificate as evidence for the client, that the requested certificate is known to the responder. This is necessary, because the `certStatus good` just indicates, that the requested certificate was not revoked during its validity period, but it not tells if it ever has been issued [20, 21].
To concern scalability issues also a lightweight OCSP profile and OCSP stapling were specified. They are introduced in the following two sections, before OCSP's new Certificate Transparency (CT) extension is explained.

Lightweight OCSP Profile

The lightweight OCSP profile minimizes communication bandwidth and client-side processing for cost efficiency and to improve scalability. This is accomplished by allowing clients to request only one certificate per request, forcing them to use SHA1 as hash function and advising them to append as less extensions as possible to their requests, as well as not to sign them. Also the clients are only allowed to request a certificate's status, if they successfully verified its signature before. Because OCSP responders are allowed to omit request extensions and they should not append response extensions to their responses, clients should be able to process responses also, if not all requested extensions are included.

[4]CertHash extension OID: id - commonpki - at - certHash OBJECT IDENTIFIER ::= id - commonpki - at 13 =id-commonpki-at-certHash, see appendix A.7

Additionally the responders should only include one `SingleResponse` in their answers. Multiple `SingleResponse`s are only allowed, if the response was pre-generated to improve performance and cache efficiency. Furthermore, this profile allows the removal of expired certificates from the database. If no authoritative record is available anymore, the `responseStatus` of the answer will be `unauthorized` or the request will be forwarded to a responder covering it. Forwarding the request is also allowed, if not all requested extensions are supported by the responder.
To further improve efficiency the fractional seconds of time values are removed and response caching is enabled to reduce the network traffic. Therefore the clients are only allowed to accept answers, if the current time is between `nextUpdate` and `thisUpdate`. DER encoded `OCSPRequests` with up to 255 bytes have to be sent via HTTP-GET by appending them base-64 encoded to the OCSP responder's URI. Larger requests are sent DER encoded using HTTP-POST.
For caching the OCSP responses should have a HTTP header of the form displayed in listing 2.7 and the DER encoded `OCSPResponse` as content [22].

```
content-type: application/ocsp-response
content-length: <OCSP response length>
last-modified: <thisUpdate> -> time of response's last modification>
ETag: "<string identifying the associated OCSPResponse (e.g. hash)>"
expires: <nextUpdate>
cache-control:
    max-age=<n>,     -- thisUpdate < n < nextUpdate
    public,          -- make response cacheable by shared/non-shared caches
    no-transform,    -- forbid proxy cache to change content type, length, encoding
    must-revalidate -- prevent caches from returning stale responses
date: <time of HTTP response generation>
```

Listing 2.7: HTTP-POST header of the OCSP response

OCSP Stapling

OCSP stapling supports the delivery of a server's authoritative OCSP responses to its clients directly within their Transport Layer Security (TLS) handshake. This is enabled by the TLS feature extensions `CertificateStatusRequest` (`status_request`) and `MultipleCertificateStatus` (`status_request_v2`), which explicitly have to be requested during certificate issuance. The `CertificateStatusRequest` extension requests only the revocation status of the server's certificate, whereas the `MultipleCertificateStatus` extension requests the revocation status for all certificates in its certificate chain.

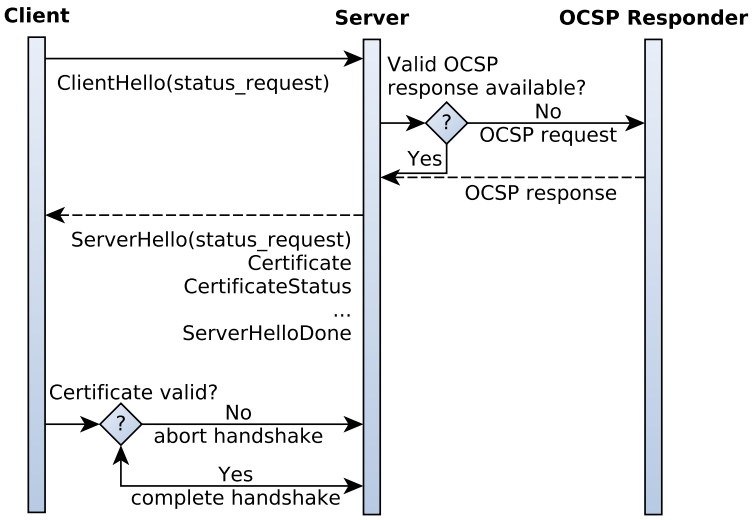


Figure 2.10.: TLS handshake with OCSP stapling

Figure 2.10 illustrates how the OCSP stapling mechanism is integrated in the TLS handshake. When a client wants to establish a TLS connection to a server and it supports one of these extensions, this is indicated to the server by defining the corresponding extension for the `ClientHello` message sent from the client to the server, to which a new connection is established.
Only if the server supports stapling and receives a `ClientHello` message with one of these extensions defined, it is allowed to deliver the revocation status for its certificate. Then it checks, if it already has a valid cached OCSP response with the `OCSPResponseStatus successful` for its certificate. Is this not the case it has to be fetched from the OCSP responder, before the `ServerHello` message can be sent to the client with the corresponding extension defined. The `CertificateStatus` message is sent directly after the `Certificate` message with the server's identifying certificate and before the remaining TLS messages to complete the `ServerHello` are sent.
Afterwards, the client has to validate the certificate and the OCSP response received from the server. Is one of them invalid, the TLS handshake has to be aborted by the client with a `bad_certificate_status_response(113) alert` to the server, otherwise the TLS handshake can be completed.
The handshake also should be aborted in the case, that the client requested an OCSP response by defining the corresponding extension and the server's response did not contain any certificate status information. Then the client should retrieve it from the OCSP responder by itself or it must immediately abort the handshake, if the corresponding feature extension was specified in the server's certificate indicating, that certificate status information has to be provided by the server [23, 24].
All in all OCSP stapling enables the clients to perform offline certificate verification. This brings performance improvements in form of a reduced Round-Trip Time (RTT)

and prevents a Denial of Service (DoS). With the reduced number of OCSP requests also the load on the OCSP responders is reduced enabling more constraint devices and a better scalability. Additionally, the client's privacy is increased by preventing the ability of the OCSP responder to track certificate usage [4, 22, 25].

Certificate Transparency (CT)

A new extension for OCSP is Certificate Transparency (CT). It is an open framework for monitoring and auditing of SSL certificates to detect mistakenly or maliciously issued certificates by otherwise trusted CAs. Thereby security attacks like website spoofing, server impersonation or man-in-the-middle attacks can be prevented [26].

CT is realized with independent publicly available append-only CT logs, which can be run by anyone. Issued certificates are submitted to a log and appended to its Merkle Tree, which enables auditors to cryptographically proof its consistency. Anyone then can submit a certificate to any log for public auditing, but the lock may not accept every root certificate. If the submitted certificate was accepted a Signed Certificate Timestamp (SCT) is returned, which is the log's promise to incorporate the certificate and the root certificate used to verify the certificate's chain in its Merkle Tree within the Maximum Merge Delay (MMD).
CT is currently an important topic, because from April 2018 on Chrome will enforce CT and reject every newly issued public certificate, if it cannot present a SCT [27]. This will force certificate owners or their CAs to publicly announce newly issued certificates by submitting them to one or multiple accepted CT logs, which increases the transparency of the system.
A SCT can already be obtained by a CA prior to certificate issuance, if a Precertificate is submitted to a log, which was constructed of the certificate to be issued. This Precertificate requires a special critical poison extension to ensure it cannot be validated by a standard X.509v3 client. After receiving the SCTs from the logs to which the certificate or Precertificate was submitted, the following **three mechanisms** exit for presenting the SCTs to a TLS client. They are also illustrated in fig. 2.11.

1. The SCTs can be embedded directly into the TLS server's certificate by encoding the `SignedCertificateTimestampList` as ASN.1 `OCTET STRING` and inserting it into an **X.509v3 certificate extension** (OID: 1.3.6.1.4.1.11129.2.4.2). Therefore the CA submits a Precertificate to the logs and embeds the received SCTs into the certificate finally issued for the TLS server.

2. The SCTs can be embedded into a **TLS extension** (`signed_certificate_timestamp`), but this extension must only be sent to TLS clients supporting it. Therefore the clients have to indicate their support for this TLS extension by including it into their `ClientHello` message during

the TLS handshake with empty `extension_data`.

3. The SCTs can be delivered to the TLS client using **OCSP stapling**. In this case they are encoded as ASN.1 `OCTET STRING` and embedded by the OCSP responder into its response using the extension specified in listing 2.8. This OCSP response is then cached by the TLS server and delivered to its clients during the TLS handshake like described in the previous section about OCSP stapling.

```
1         extnID    OBJECT IDENTIFIER ::= { 1.3.6.1.4.1.11129.2.4.5 }
2         extnValue OCTET STRING      ::= SignedCertificateTimestampList
```

Listing 2.8: ASN.1 specification of OCSP's CT extension

Advantage of embedding the SCTs into a TLS extension or OCSP response is, that SCTs for a certificate can be added dynamically and their retrieval does not delay the certificate's issuance.
Also if the SCTs are presented to the client into a TLS extension no modification of the CA is required, but with the disadvantage, that compared to the other two mechanisms a modification of the TLS server is necessary.

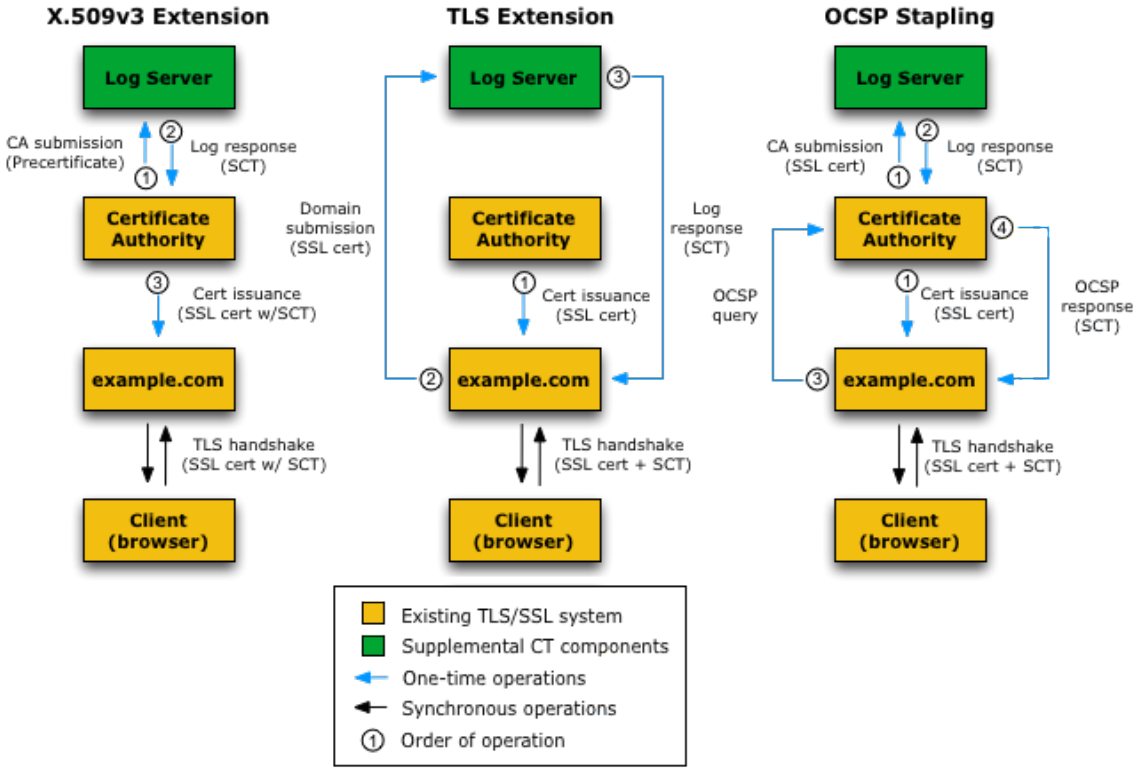


Figure 2.11.: Certificate Transparency Mechanisms (image adapted, source: [28])

Independent of the used mechanism a TLS server should always present to its clients SCTs retrieved from multiple logs for the case, that a client not accepts on of them.

For validating the received SCTs a TLS client computes the signature of their data and certificate to verify it with the log's public key. Also clients must reject SCTs with a timestamp in the future.
To ensure a log's consistency its behavior has to be monitored. Therefore partial information is checked against other partial information or the whole log is retrieved to check its consistency. But certificate misuse detection and distrusting logs is not performed by the logs itself, it relies on interested parties like domain owners or CAs. They monitor a whole log or just certificates for their domains of interest and react accordingly, if they detect suspicious certificates.
Auditing of the SCTs and their corresponding certificates is performed by interested auditors like browsers or researchers, which check the SCTs' validity and if they were correctly appended to their log [28, 29].

2.3. Fuzzing

Fuzzing is an effective software testing technique to identify security vulnerabilities by feeding a tested software target with a large number of varying input and monitoring its reaction. This is performed with the intention to cause unexpected program behavior to reveal possible exploitable vulnerabilities, which could be abused to manipulate the software target or trigger a DoS. Because fuzzing can be performed on every target parsing input, it is a popular vulnerability testing technique to improve the robustness and usability of software, as well as an important part of the Secure Development Lifecycle (SDL) [30, 31, 32].
White-box fuzzing has full access to the target's source code enabling the construction of an execution tree. Thereby the covered code paths can be traced and used to influence the further creation of fuzzed data. This enables the verification of hit target code paths to achieve a higher code coverage and quality.
Black-box fuzzing in contrast, ignores the tested software target and concentrates on the target's input data format. This makes the testing procedure more efficient and reusable for other software targets with a similar input data format, but with the disadvantage, that the hit code paths cannot be verified. Thereby complex application logic or code paths may be missed [33].

2.3.1. Phases

Following Eddington [32], the fuzzing process can be divided into the following six phases:

Investigation

First the fuzzing target, its data input method and format have to be investigated to be able to select a fuzzer.
Dependent on the target's input data format fuzzers can be classified into file, network, general and custom fuzzers. Network fuzzers target network protocols, while general fuzzers can target files, network protocols and other input data formats for custom I/O interfaces. Custom fuzzers in contrast target just one specific input data format or network protocol [32].

Modeling

The second phase is the modeling phase. The data and states of the target system have to be modeled, that the fuzzed input can be created. This is one of the most important phases, because the result of the fuzzing run highly depends on the quantity and quality of its inputs. To speed this process up templates or automatic tools to analyze the target and its network captures can be used.
For the construction of input two different methods are available. The input can be created mutation-based starting with known good mutation templates as input, which are mutated by random bit-flips dependent on the previous input.

The other method is generation-based. First the fuzzed input format or network protocol has to be modeled, which is a time-consuming process. Then, after the model representing the protocol's elements and their relationships was built, the fuzzed data can be generated based on it.
Because the parsing code is the primary target of fuzzing, fuzzers produce input, which likely triggers code-defect patterns, by applying three different levels of anomalies on the model. The first anomaly are abnormal field values like for example boundary values and values close or equal to zero for integers. Also the inserting of format strings into strings or setting them null belongs to it, as well as increasing field sizes to provoke buffer overflows.
The second anomaly concerns the message structure multiplying or deleting message elements to detect vulnerabilities in message parsers.
Whereas the third anomaly targets the message sequence by deleting and multiplying entire messages to provoke dead-locks and out-of-memory errors.

Comparing the two methods introduced for input creation, the generation-based method using heuristics is more efficient and thereby better suited to test complex

protocols, than the other method based on random mutations. The syntactic and semantic knowledge of the generated model can be used to reduce the number of test cases and focus on protocol areas with a higher susceptibility to vulnerabilities. Thereby the inner components of complex parsing code can be reached faster. This also was observed by Miller and Peterson [31], who compared both methods and reached a 76% higher code coverage with a generation-based fuzzer, than with a mutation-based one [30, 33].

Validation

After the model was built it has to be validated to ensure it is correct and that the fuzzer can communicate with the system. For this purpose automated tools, network monitors or debuggers can be used.

Monitoring

Then the target has to be monitored to detect hangs and crashes, that they can be reported and the target can be reset to keep the fuzzing process running. Important for monitoring is, that the produced error states are logged with enough detail to be later able to reproduce them.

Run

If the monitoring is established, the fuzzing process can be executed. This should be performed in a stress like manner using parallelization to increase the probability to detect a fuzzing bug.

Review

After an appropriate number of iterations the results have to be reviewed to identify duplicates and to estimate, if an occurred bug can be exploited. This can be accelerated by the support of crash analysis tools.

2.3.2. Frameworks and Tools

For the construction of a custom fuzzer, already different popular frameworks and tools exist. Criteria for the compared tools listed in table 2.2 was their applicability for network fuzzing and that they are released under an open-source license.

Name	Target	Language	Model	Validation	Monitor	Last Release
Scapy	Network	Python	Python			2016
Peach CE	General	C#	XML	GUI	x	2014
Sulley	Network	Python	Python	Netmon	x	2014
boofuzz	Network	Python	Python	Netmon	x	2017

Table 2.2.: Fuzzing Frameworks and Tools

Scapy is a powerful packet manipulation tool without other fuzzing related features. It builds packets based on packet definitions in Python and facilitates their reuse. Scapy already contains various packet definitions, which include also packet definitions for OCSP[5] [34].
The Peach Community Edition (Peach CE), on the other hand, is a full-fledged general black-box fuzzing framework built in C#, which produces the fuzzed data generation-based on the basis of a data model built of an XML template or partially generated by automatic tools capable of converting XML, ASN.1 or Wireshark captures. To ensure the model is correct and the fuzzer can communicate with the fuzzed target a validation tool with a Graphical User Interface (GUI) is available and Peach CE has the ability to parallelize the fuzzing process. This brings significant performance improvements. Besides that, Peach CE is also capable of monitoring the fuzzed target by network capturing, debugging, controlling the Virtual Machine (VM) or reverting it to a known good state. Additionally it speeds up reviewing by grouping duplicates and providing crash analysis. But it has the disadvantage, that only Peach CE is available under an open-source license, which is not maintained anymore since 2014, when Peach was forked to closed source [32, 35].
Sulley is another frequently used generation-based black-box fuzzing framework. It is a network fuzzer written in Python, which is also used to define its data model. Sulley provides a better usability than Peach CE, because less overhead is required to define its data model compared to Peach CE's XML approach. Its architecture is visualized in figure 2.12. The fuzzed network protocol's `Primitives` are grouped into `Blocks` to represent the packets of the protocol's individual requests. User-defined `Primitives` can be added using `Legos` and a `Session` is built by linking the blocks together to a graph, which is traversed for fuzzing of the target. Like Peach CE Sulley offers parallelization, which is accomplished by defining multiple fuzzing `Targets` for a single session. Its monitoring capabilities are similar to the

[5]OCSP packet definitions of Scapy, `https://github.com/secdev/scapy/blob/master/scapy/layers/x509.py`, Accessed: 2017-12-15

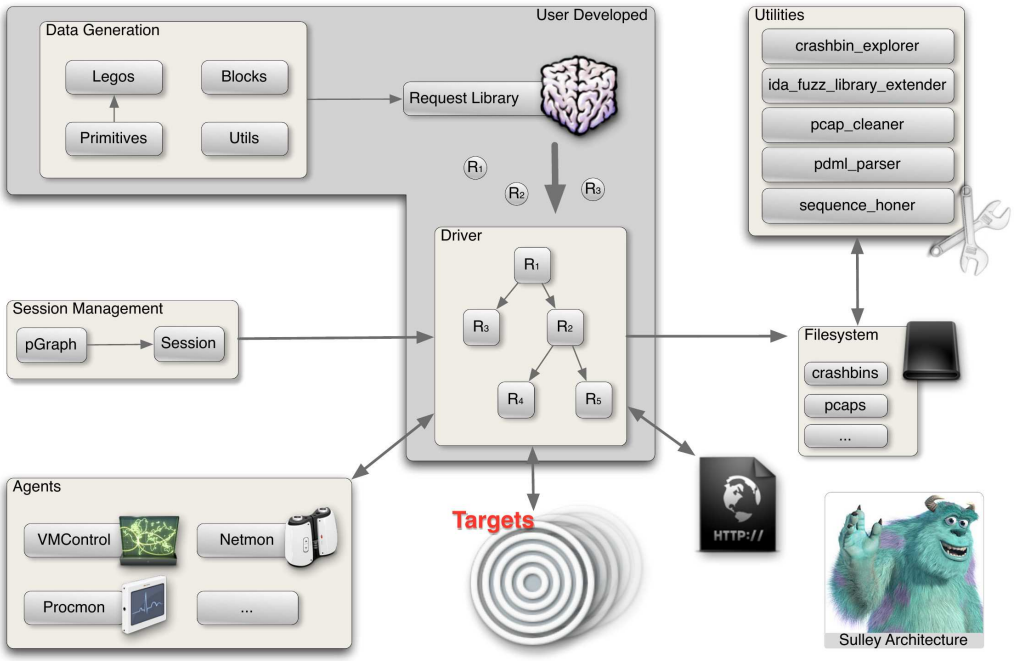


Figure 2.12.: Sulley Architecture Diagram [36, p. 19])

ones of Peach CE and their implementation is realized by `Agents`, which are added to the `Session`'s `Targets`. A web monitoring interface is available to observe the fuzzing run's progress and its results. To verify the data model's validity and the fuzzer's capability to communicate with the `Target` the captured packets of the `Netmon Agent` can be investigated. But compared to Peach CE, Sulley just offers categorization and replay of the detected faults and no additional crash analysis. In addition it is less stable and also unmaintained since 2014 [32, 35, 37].
Comparing the latest published versions of both frameworks, Sulley 1.0 provides a better usability than Peach CE 3.1, but is less stable. Therefore Pereyda [35] forked Sulley in 2015 to implement various bug fixes and refactor it. This successor of Sulley was published by him under the name boofuzz and is still maintained. Additionally he added extra text- and CSV-loggers to make Sulley's `Netmon Agent` redundant and improve the recording and evaluation of the results.

2.4. Threat Model

According to the Committee on national security systems (CNSS) a vulnerability is a "Weakness in an information system, [...] or implementation that could be exploited by a threat source" [38, p. 131], which triggers the vulnerability's exploitation intentionally or by accident.
For OCSP different vulnerabilities exist affecting the protocol's design and its implementation, which depends on the OCSP responder in use.

Vulnerabilities resulting from the protocol's design are that, as in section 1.2 already explained, many applications relying on OCSP as revocation information provider just implement a soft-fail approach or they process CRLs as fall-back solution. This is an incident at the latest, if both solutions are not accessible due to a DoS. Therefore to circumvent a DoS in case of a flood of queries OCSP allows unsigned error responses, because the cryptographic signature's computation significantly affects the time required for response generation. However, this enables spoofing attacks. Attackers can use for example the error response status `tryLater` to send false error responses and get OCSP clients to soft-fail or use their fall-back solution.
Another way to prevent a DoS is response pre-computation to improve an OCSP responder's performance and cache efficiency. But OCSP responses can also be cached by TLS clients, TLS servers or proxy servers, if the responses' HTTP headers were set correctly or OCSP stapling is used. This enables replay attacks after a certificate was revoked, until the cached response's expiration date is reached. For example the OCSP responder responsible for answering OCSP requests for the certificate of the domain `www.google.de` pre-generates responses with a validity interval of 1 week[6]. Only after this validity interval is exceeded the response expires and a new one will be generated by the responder to answer further OCSP requests. The OCSP server can prevent this by defining the nonce extension, if it was included in the client's request, but it is not guaranteed, that every server processes this extension. In the example of the OCSP responder responsible for the domain `www.google.de`, the nonce extension was ignored by the responder.
One more replay attack is possible, because OCSP requests do not contain the addressed responder. Thereby it is possible for an attacker to replay a client's request to another responder not knowing, that the requested certificate was revoked.
Furthermore, the `pref-sig-algs` extension allows a client to specify its preferred signature algorithms. This can be exploited by an attacker for a man-in-the-middle downgrade attack. But it is not a significant security incident, because OCSP responders must not process this extension and a client can reject responses, which not satisfy its security policies [19].
Another more complex attack is possible due to the predictability of the responses. When they are computed in real-time, the defined extensions and the requested certificate's status don't change, only the time values differ during multiple requests,

[6]Measured on 14.03.2018 using OCSP Checker (`https://www.pkicloud.com/tools.html`)

which normally just have an accuracy of one second. Thereby the value of the `tbsResponseData` can be predicted dependent on the point in time, when the request was sent and its hash can be calculated in advance to search for a hash collision. If a hash collision was found, the responder then can be forced to sign even valid certificates, if the CA's certificate is used to sign OCSP responses and the request was submitted at the correct point in time. This even can be scaled, if the nonce extension is processed by the responder. Because of the minimal interaction between the attacker and the responder, this attack cannot be detected. Therefore weak hash algorithms and the CA's certificate should not be used to sign OCSP responses. How this attack works in detail is explained in section 6.1.
One more vulnerability with low impact is the missing cryptographic protection of the response's HTTP header. Its time values can be arbitrarily manipulated by an attacker. Therefore an OCSP client should only rely on the header for caching and according to OCSP's lightweight profile certificate status validation should only take the response's content into consideration, which contains the signed `OCSPResponse` [22].

Implementation related vulnerabilities are software faults or weaknesses in authentication, which can be directly exploited or whose consequences can be exploited. Consequences can be hangs or crashes. Hangs cause a DoS and can be the result of infinite loops, many nested loops or deep recursions. Crashes are the result of unhandled exceptions like buffer overflows, segmentation faults, unvalidated inputs, race conditions or null pointer dereferences. Both consequences are targeted by fuzzing and especially crashes should be investigated with more detail, when they occur during a fuzzing run, to assess whether they can be exploited.
For OCSP implementations vulnerabilities are mainly expected in the ASN.1 parsing code. An ASN.1 integer's value range is just limited by ASN.1's long length notation[7]. Problematic in this case can be big positive or negative numbers and other values generated by applying the anomalies mentioned in section 2.3.1 on the model. Also Unicode strings or big requests can cause potential vulnerabilities, as well as OCSP's various extensions, whose value is just defined as octet string containing the extension's data.
These implementation specific vulnerabilities are well suited for fuzzy testing. Dependent on the time available a fuzzer tries to achieve the greatest possible coverage of possible input values to be fed to the tested target. Because of this fuzzing was chosen as the testing methodology to investigate in this thesis.

[7] $maxValueLength = 256^{126} - 1 \rightarrow maxIntValue = 2^{maxValueLength*8-1} - 1$ (see section. 2.1.2)

3. Methodology Selection

The construction of a penetration testing framework for OCSP responders with focus on fuzzy testing starts with the investigation phase. First the fuzzing target has to be selected, before the approach for generating the fuzzed input can be chosen. Afterwards a suitable fuzzing framework can be selected as base to construct the custom OCSP fuzzer.

3.1. Fuzzing Target

Goal of this thesis is to build a custom network fuzzer for protocol-based fuzzing of OCSP responders. Therefore as fuzzing target any OCSP responder can be considered.
The OCSP responder, which comes with the cryptography and SSL/TLS toolkit OpenSSL 1.0.2l[1] was selected as first target, because it can be installed out of the package sources of Ubuntu 16.04 64-bit, which enable a fast and easy setup. Among other tools the OpenSSL library provides tools for cryptography and enables to build a CA for issuing X.509 certificates. These certificates can also be revoked and CRLs can be issued, as well as an OCSP responder is included for clients to check a certificate's current revocation status. This OCSP responder was selected as first target, because with its fast setup it is well suited for testing the produced fuzzer already during development to be able to improve it and ensure its stability. Because I assume, that this open-source toolkit already was fuzzed during its SDL it should be only used to test and improve the fuzzer to be able to more extensively fuzz a second target.

[1] OpenSSL, `https://github.com/openssl/openssl/releases`, Accessed: 2018-03-05

As second target the proprietary commercial OCSP responder contained in the backend of ESCRYPT's Key Management Service (KMS) was selected, because it is a new product, which was not yet intensively fuzzed. Thereby it is the more interesting target for extensive testing. Its architecture is displayed in fig. 3.1 and is

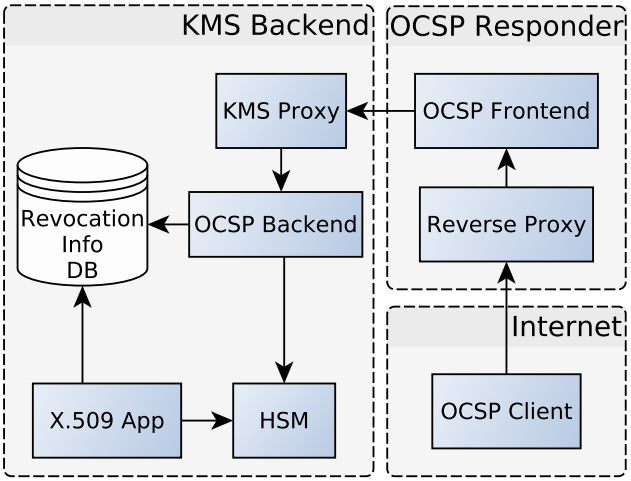


Figure 3.1.: ESCRYPT KMS OCSP Architecture Diagram

divided into an `OCSP Frontend` and an `OCSP Backend`, which are both located in different trust zones and accessed via RESTful web services. `OCSP Clients` submit OCSP requests to the `OCSP Frontend` over the `Reverse Proxy`, which simultaneously prevents further access. The `OCSP Frontend` is responsible for validating incoming OCSP requests and submitting the valid ones to the `OCSP Backend`, which is contained in the `KMS Backend` and accessed over the `KMS Proxy` again limiting further access. The `OCSP Backend` looks up the requested certificates' current revocation status in the `Revocation Info DB` and returns signed responses using the `Hardware Security Module (HSM)` also contained in the `KMS Backend`. To keep the `Revocation Info DB` up-to-date the `X.509 App`, which issues and revokes certificates, pushes new revocation information to the `Revocation Info DB`.
To ensure the communication participants' authentication, the communication between the `OCSP Responder`'s `OCSP Frontend` and its `OCSP Backend` is secured using authenticated TLS. Additionally the `OCSP Responder`'s components are packed into separate docker images. The one containing the `OCSP Frontend` is based on Ubuntu 16.04 64-bit. Furthermore the `OCSP Frontend` is implemented in Java, which is a managed language hiding the memory management from the programmer. Thereby the risk of memory errors like segmentation faults is reduced, but unhandled exceptions, deadlocks or memory spikes resulting in a DoS or information-disclosure bugs can still be detected by fuzzing [33].

3.2. Fuzzing Approach

Because the OCSP responder is fuzzed protocol based, it is obvious to build a generation-based fuzzer, because an exact ASN.1 specification of the protocol is available. Thereby the syntactic and semantic knowledge of the generated model can be used to reduce the number of test cases required for reaching the inner components of complex parsing code. Miller and Peterson [31] reached a 76% higher code coverage with generation-based fuzzing compared to the mutation-based approach.
In order to be able to also compare both input creation methods, both should be implemented with focus on the generation-based model and compared in a subsequent fuzzing campaign based on the second target.

3.3. Fuzzing Framework

To create the custom OCSP fuzzer the different options introduced in section 2.3.2 have to be considered to be able to select to best base configuration.
Scapy is just a packet manipulation tool without other fuzzing related features. Therefore it cannot be used as standalone solution, but it already contains packet definitions for OCSP[2], respectively X.509. But after examining them more closely it turned out, that just packet definitions for OCSP responses are included yet. They cannot be used for fuzzing of an OCSP responder. Therefore it is not worth the effort to integrate Scapy in another fuzzing framework or write a complete own fuzzing framework utilizing the packets generated by Scapy. The OCSP request related packet definitions required for fuzzing of an OCSP responder are still missing.
Peach CE and Sulley on the other side are full-fledged general black-box fuzzing frameworks. Peach CE is based on C# and its packet definitions are written in XML. Sulley is based completely on Python, which requires less overhead for the packet definitions and enables an easier customization.
But because both frameworks are not maintained anymore since 2014 and Sulley is not stable, boofuzz was chosen as base to build the custom OCSP fuzzer. It provides the advantages of Sulley and also compared to Sulley its code was refactored. The `Legos` were replaced by a special interface (`IFuzzable`) to ease the framework's extendability and also a new text-logger, as well as a CSV-logger were added to improve the quality of the logs.

[2]OCSP packet definitions of Scapy, `https://github.com/secdev/scapy/blob/master/scapy/layers/x509.py`, Accessed: 2017-12-15

4. Design and Implementation

After the investigation phase the methodology is known and the model for OCSP can be created. Therefore first the required components of the fuzzer have to be specified, before they can be implemented.

4.1. Design

The custom OCSP fuzzer's base is boofuzz, which is written in Python 2.7. To avoid migration problem, the custom OCSP fuzzer should also be based on this Python version. The main components of boofuzz are presented in the following section, before the additional components required to model an OCSP request and fuzz OCSP responders are introduced.

4.1.1. Boofuzz

Boofuzz is based on Sulley, therefore Sulley's architecture for the most parts is also valid for boofuzz. This section introduces the most important components of boofuzz and its improvements compared to Sulley. The architecture of boofuzz is displayed in fig. 4.1.

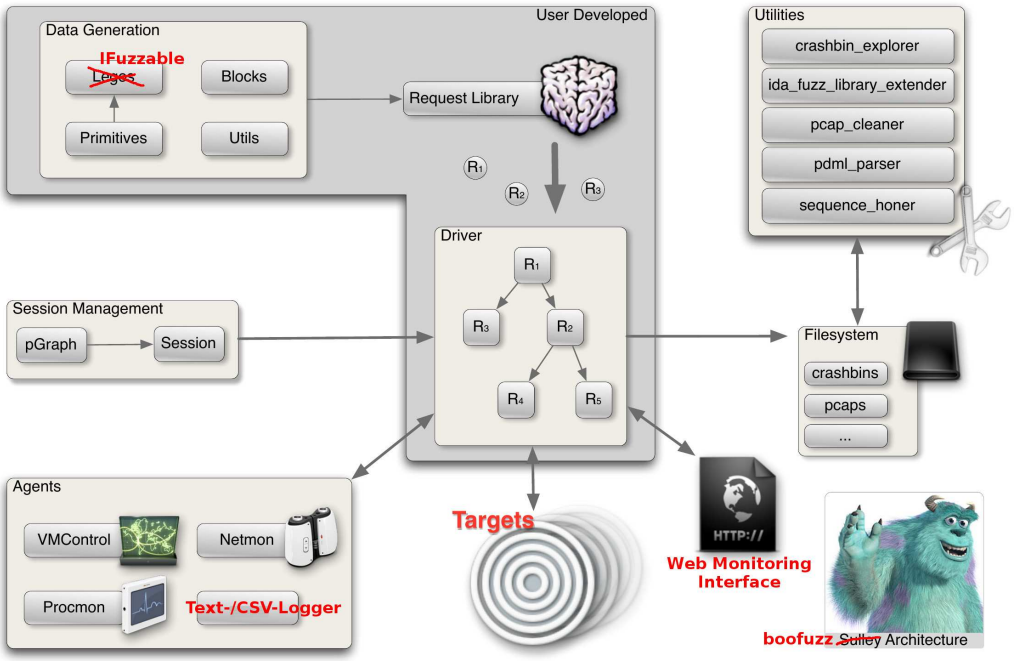


Figure 4.1.: Boofuzz Architecture Diagram (image adapted, source: [36, p. 19])

Agents

Monitoring of the fuzzing run is mainly realized by agents. The **process monitor agent (procmon)** runs on the target's machine and is responsible for starting the target, crash detection and stopping it after a hang or crash was detected to be able to restart it again. Boofuzz includes two different versions of procmon. One (`process_monitor.py`) based on pydbg[1], which is a pure python debugger. It is attached to the target's process and supports dumping of crash synopses, which can later be used during post mortem analysis to build a graph of all crash paths. But unfortunately pydbg depends on a 32-bit Windows environment and cannot be ported to the 64-bit Linux environment on which the second fuzzing target provided by ESCRYPT relies [37].
Therefore its alternative (`process_monitor_unix.py`) supporting Unix operating systems has to be used for the custom fuzzer. Its crash details are limited to the exit code of the process, if it terminates, or if the process is stopped, the signal that caused the death and the core dump created by the operating system (OS) [39].

The **network monitor agent (netmon)** is based on pcap[2]. It must be launched at a location where it can monitor the network traffic produced during the fuzzing run to be able to capture it. Afterwards the capture files created per test case can

[1]pydbg, `https://github.com/OpenRCE/pydbg`, Accessed: 2018-03-07
[2]Python pcap module, `https://github.com/dugsong/pypcap`, Accessed: 2018-03-07

be analyzed using Wireshark's GUI[3].
But boofuzz also includes a text- and a CSV-logger, which both log the requests sent by the fuzzer and the responses received from the target, making netmon obsolete. Therefore netmon shall be used to validate the model and after it was successfully validated, the final decision about the agents and log handlers to establish monitoring should be made [37, 39].

The **web monitoring interface** is not an agent it is a minimal web server displaying the progress of the fuzzer and its results. It reports the test case number for detected hangs and crashes, as well as their synopsis. Additionally it enables to pause and resume the fuzzer [37].

A **VMware control agent (vmcontrol)** is also available. It can control the VM in which the fuzzed target runs and is able to handle snapshots to revert the VM to a known good state, if a fault was detected or the tested target is not available anymore [37].
This agent is not applicable for the custom fuzzer to build, because the OCSP responder of ESCRYPT runs in docker containers and not in a VM. Also an OCSP responder should normally just fetch a certificate's current revocation status and not update its database. Therefore the necessity of occasionally resetting the target and its persistent data to their initial state does not exist.

Session

Like in Sulley in boofuzz the network protocol's single primitives are grouped into `blocks` to represent individual requests, which are linked together to build a `Session` [37]. But linking multiple requests together is not necessary for the custom OCSP fuzzer, because OCSP is just a request-response protocol with no further following messages.
To improve the extendability of boofuzz, Sulley's `Legos` were removed. Additional primitives and blocks in boofuzz are created by implementing the provided `IFuzzable` interface [39].
And after the model is build it can be used for fuzzing. Therefore boofuzz creates the test cases by mutating the model's single primitives. This is either realized by mutating their values randomly or by iterating over the fuzz libraries of the fuzzable primitives containing so called "smart" values instead of fuzzing the primitive's full value range. These values are created based on the first anomaly explained in section 2.3.1.

[3]Wireshark, `https://www.wireshark.org/`, Accessed: 2018-03-07

4.1.2. Custom Components

Because HTTP parsers are commonly used I assume, that they were already extensively fuzzed and they are robust enough, that no further fuzzing of them is necessary. Therefore only the content of the OCSP request should be fuzzed and not its HTTP header. The header should be modeled statically using the built-in primitives of boofuzz.

Mutation-based Model

In section 3.2 was decided, that the OCSP fuzzer should be fuzzed mutation- and generation-based to be able to later compare both approaches. For mutation-based fuzzing known good and valid OCSP requests are selected as mutation templates and mutated for every new test case until they are exhausted. This should be realized by reading-in the mutation templates byte by byte and building the mutation-based model with the built-in `Byte` primitive of boofuzz.

Generation-based Model

The building of the generation-based model is more complex, because boofuzz has no built-in ASN.1 types. As explained in chapter 2.1 an ASN.1 type is a TLV triplet. Every type has its own tag and the length depends on the value. Therefore a general ASN.1 type shall be implemented, which is responsible for fuzzing and DER encoding of the tag and the length, that for every new type just the decimal value of its universal tag has to be specified and a primitive responsible for fuzzing and DER encoding its value.
This general ASN.1 type should also support combinatorial fuzzing of its length and value field, as well as the options listed in table 4.1. Based on it the remaining ASN.1 types listed in the table and required to model the OCSP request should be implemented to make the fuzzer applicable for fuzzing of an OCSP responder.

primitive types:	`INTEGER, BIT STRING, OCTET STRING, OBJECT IDENTIFIER, NULL, BOOLEAN`
string types:	`IA5String, TeletexString, PrintableString, UniversalString, UTF8String, BMPString, GeneralizedTime`
constructed types:	`SEQUENCE, SEQUENCE OF, SET, SET OF`
meta types:	`CHOICE, ANY`
options:	`DEFAULT, EXPLICIT, IMPLICIT, OPTIONAL, SIZE`

Table 4.1.: ASN.1 types and options to implement

Additionally to improve the efficiency of the generation-based model and speed up testing, test case skipping should be implemented, if a single ASN.1 type's mutation type reaches a certain crash threshold. Then the specific mutation type is exhausted and its remaining test cases should be skipped.
Mutation types, which should be implemented for ASN.1 types are optional, tag, length and value mutations, as well as combinatorial mutations of the type's length and its value.

4.2. Implementation

The following chapter tells how the custom components of the generation-based model were implemented and which improvements were necessary to overcome unexpected challenges during their implementation. Afterwards it explains how monitoring was established.
Due to the triviality of the mutation-based model's implementation, it is not further discussed in this chapter.

4.2.1. Modeling and Validation

In order to be able to recognize challenges as early as possible, the model was build incrementally and during development mutations were disabled to be able to continuously validate the packets generated by the fuzzer using Wireshark and a JavaScript tool[4].
First a complete ASN.1 specification of the request's single components introduced in section 2.2 was required to be able to define the model. This could not be found in one document. Therefore multiple sources had to be considered, to be able to retrieve a complete specification of an OCSP request. It can be looked up in appendix A.1.

General ASN.1 Type

After the specification was complete, the DER encoded general ASN.1 type could be constructed using boofuzz `Block`s. The TLV triplet's length was realized with a boofuzz `Size` block, which was assigned to the triplet's value and whose encoding was adapted to ASN.1. But it turned out after most required value types of the OCSP request were modeled, that this solution was not efficient enough due to the many recursive function calls required to render the OCSP request. For the computation of the length values using the adapted `Size` block, the associated values' rendered length was computed separately for every TLV triplet. Especially for constructed

[4]asn1js, `https://github.com/lapo-luchini/asn1js`, Accessed: 2018-03-16

types, where the whole structure had to be temporarily rendered to determine its length, this was very expensive in terms of computation time. Because of this the time required to render an OCSP request with every modeled field at the end took between 40 and 50 seconds per packet.
This performance was not acceptable and therefore the single boofuzz blocks were removed from the general ASN.1 type's implementation and replaced by a custom block directly rendering every TLV triplet's tag and length. It just requires an individual implementation to render a type's value, its tag and custom mutations.
Fig. 4.2 shows in the architecture diagram how this custom block and the different implemented ASN.1 value types correlate with each other.

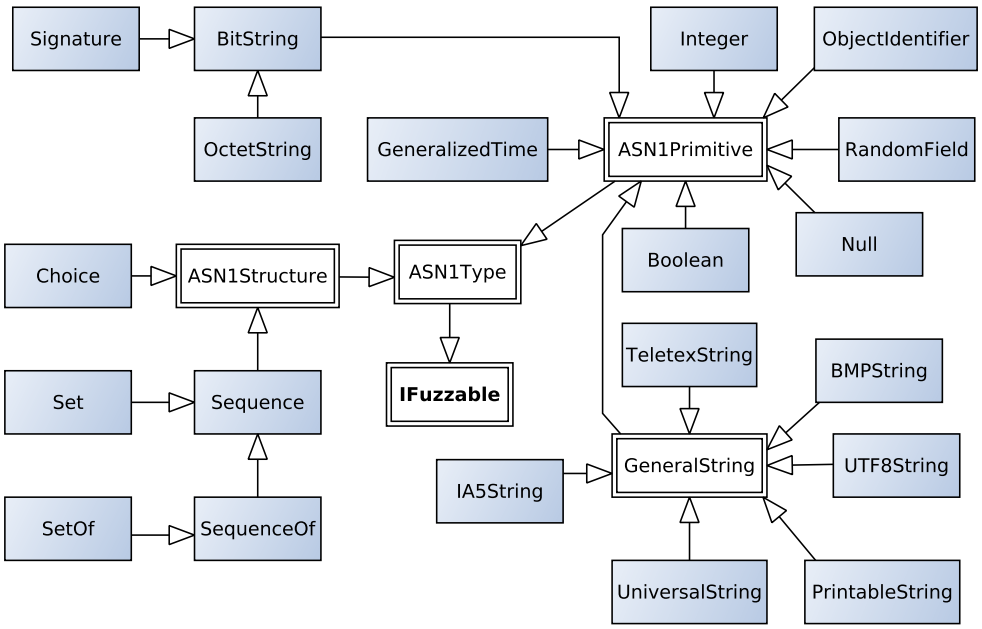


Figure 4.2.: Custom Component Architecture Diagram

The custom `ASN1Type` implements the `IFuzzable` interface of boofuzz required to define new blocks. It contains default implementations of the required methods and is responsible for rendering the TLV triplet's tag and length, as well as for mutating them and adding a mutation with the type's default value to every `ASN1Type`, which is marked as `OPTIONAL`. Otherwise if an `OPTIONAL` `ASN1Type` or a sub-type contained in its structure currently were not mutated and would all be rendered with their default values, the `OPTIONAL` `ASN1Type` is skipped during encoding. This approach was chosen to avoid the fuzzed packet being dropped early by the target's error handling logic, if it not supports an `OPTIONAL` part of the OCSP protocol. Then an increase of the target's coverage by the remaining test cases could be prevented.
The `ASN1Primitive` and the `ASN1Structure` extend the `ASN1Type`, to be able to implement values of primitive and constructed form by again extending them and

just implementing the encoding of the specific value and its custom value mutations. To improve the model's performance an `ASN1Primitive`'s encoded value is cached and only updated, if it was changed. Mutations are realized by defining for every implemented ASN.1 type a list of values to fuzz and iterating over this list for every type's instance. Additionally to the value mutations primitive types also support combinatorial fuzzing, which can be configured by giving the number of items to uniformly choose out of a type's length and the value fuzz library. These are then picked out of the fuzz libraries and combined with each other. A `DEFAULT` value can also be defined for an `ASN1Primitive`. Then according to ASN.1's DER the primitive is skipped during encoding, if its current value is equal to its default value. The `ASN1Structure` distinguishes itself of the `ASN1Primitive` by containing a stack of `ASN1Type`s as value instead of a single value. Therefore it not requires a fuzz library, because its mutations are a union of its structured values' mutations.

Specific Value Type Implementations

The encoding of the specific value types was implemented according to chapter 2.1, but during implementation of the specific ASN.1 types special implications had to be made.

ObjectIdentifier
As for most ASN.1 values also the value range of an `ObjectIdentifier` is just limited by the maximal value length of an ASN.1 value. Therefore to achieve the best possible coverage of interesting OID values within a reasonable amount of time, additionally to interesting values having a one and a multiple byte encoding per SID also a list of 1,021 known OIDs was extracted out of OpenSSL's fuzz library[5] and added to the `ObjectIdentifier`'s fuzz library.

Signature
A special primitive was required to model an OCSP request's `Signature`. It is encoded as ASN.1 `BIT STRING` and calculated from the encoded value of the `tbsRequest`, which has to be associated with it. Therefore the `Signature`'s implementation extends the `BitString` type and adds to the customized random mutations of the `BitString` additional mutations changing the hash algorithm in use[6]. The signature is computed always using RSA with a static key size of 2048 bit.

String Types
Additionally for efficiently implementing the different string types a `GeneralString` was introduced to be extended by the implementations of the specific string types. Thereby a specific string type can be defined by specifying its universal tag number and the number of bytes required to encode a single character. For simplicity it is

[5] OpenSSL's OID fuzz library, `https://github.com/openssl/openssl/blob/master/fuzz/oids.txt`, Accessed: 2018-01-02
[6] Supported hash algorithms: SHA512, SHA384, SHA256, SHA1 and MD5

not distinguished between ASCII and UTF-8 string encodings. Every string type uses the fuzz library of the `GeneralString` containing different ASCII, UTF-8 and binary strings. The fuzz library's values were taken over from the already existing string primitive of boofuzz and amended by additional Unicode strings of varying length covering the different byte numbers required for a single character's encoding. String types are encoded dependent in the type's specific number of bytes per character. If a single character's encoding is oversized and its encoding requires more than the specified number of bytes or its encoding is undersized, it is padded to the next byte boundary using zero bytes. Thereby for simplicity the alphabet of the string types with character sets similar to the ASCII alphabet (`IA5String`, `PrintableString`) is extended with universal characters having a single byte UTF-8 encoding. Also if a binary string out of the fuzz library is selected as current value mutation, it is always directly returned without additional encoding. Additionally, for string instances `SIZE` constraints can be specified. They are applied after encoding by properly cutting or padding the result value with zero bytes.
Problems during the implementation of the string types were caused by the integration of the Unicode characters, because Python 2.7 does not support strings of different encodings. Strings just represent byte strings and an appropriate conversion is necessary to integrate Unicode strings into them without raising exceptions. But more problematic was the size of the string types' fuzz libraries. Each of them had a size of around 70 MiB and with every modeled string instance the memory consumption of the fuzzer with the generation-based model increased up to 1.8 GiB for almost the complete model. Compared to a memory consumption of 74 MiB without the string fuzz libraries this has a huge impact. Therefore the fuzz libraries' implementations were made static for every ASN.1 type and the definition of additional random or instance specific values to fuzz was realized by extra private fuzz libraries containing just a field's instance specific mutations. This reduced the memory footprint of the fuzzer with the generation-based model and all modeled string instances to 137 MiB.

Structured Types
Special implications also had to be made previous to the implementation of the structured types. The `Set`'s implementation was realized by extending the `Sequence`. To take into account the non-existing order of a `Set`, compared to the `Sequence` the rendered values of its collection's fields just had to be randomly shuffled, before they could be concatenated and returned.
Additionally, the `SequenceOf` and the `SetOf` are collections of one or more occurrences of the same type. Therefore the implementation of the `SequenceOf` was implemented by extending the `Sequence`. To achieve multiple occurrences of the same type additional custom mutations of varying number were added to the `SequenceOf` with other mutations of the contained type or just by repeating the returned rendered value instances of the type contained in the collection. The `SetOf` again was implemented by extending the `SequenceOf` and randomly shuffling the single rendered values before returning and concatenating them.

Meta Types
The implementation of the meta types also was not trivial. The `Choice` was implemented as `ASN1Structure` without tag and length value. Therefore additional tags set for a `Choice` get forwarded to its alternatives, which were pushed to its value stack. When the `Choice` is rendered, just the rendered value of the last mutated alternative is returned or the one of a randomly chosen alternative, if none of the alternatives was mutated yet and all of them still have their original value.
This `Choice` type then also was used for modeling ASN.1's arbitrary `ANY` type. Often this `ANY` type's value is influenced by an associated `INTEGER` or `OBJECT IDENTIFIER`, but it is not exactly defined. Therefore it was modeled with a `Choice` containing in its value stack an ASN.1 `NULL` field and a `RandomField` as alternative. This `RandomField` renders random strings for which the initial tag and value can be specified, as well as the number of random values, which should be contained in its private fuzz library, and their encodings.

4.2.2. Monitoring

After the complete model was successfully implemented and directly validated as part of the implementation process, fuzzing of the single fields contained in it could be activated to be able to setup and select the boofuzz agents and tools for monitoring.

Process Monitor Agent
First procmon had to be configured properly, that it is able to control and monitor the target. To ensure procmon is able to detect unexpected behavior of the target during the fuzzing run an extra dummy OCSP responder was created in Python. It falls into an infinite recursive loop, when it receives a client's request until it is terminated by a runtime error. That the target's termination with exit code 1 due to the runtime error was successfully detected by procmon can be seen in the results of the monitoring interfaces contained in appendix B.1. There also the dummy responder's code snippet is listed.
To also check if core dumps are created correctly, a second dummy OCSP responder was created in C. If a request is submitted to it, it crashes due to a segmentation fault. Previous to this test it was necessary to enable core dumps by increasing their size limit set by the OS, because for Ubuntu 16.04 64-bit the size limit for core dumps is set to 0 bytes to disable them by default. The results of the web monitoring interface and the procmon log contained in appendix B.2 show, that the core dump was successfully created. There again also the dummy responder's code snippet can be looked up.

Loggers
Boofuzz includes text- and CSV-loggers for capturing the results of the fuzzing run. To be able to select the best performing one, a short fuzzy test was performed on the OCSP responder provided by OpenSSL, which was selected in section 3.1 as

target for testing the fuzzer.
During 10k of 100k test cases, the fuzzer produced a text-log of 6.5 GB and a CSV-log of 0.5 GB, as well as 5 crashes were detected. Therefore due to the same content of both files and their huge difference in size it was decided to just use the CSV-logger for further fuzzing runs. But because the output of the CSV-log is still really big a filter was implemented for it, which just keeps the case number and no further log messages for a test case, if it was not interesting and no crash was detected during its execution.
The 5 crashes, which were detected during the short fuzzing run, could be traced by investigating the procmon log, which can be seen in listing 4.1. The crashes were raised, because the maximal number of file handles allowed by the OS was exceeded. By default this limit is set to 1024 for the OS in use and due to the detected crashes it was decided to increase it for further tests to avoid this kind of crashes.

```
1 Error accepting connection
2 140707607934616:error:02008018:system library:accept:Too many open files:b_sock.c
      :859:
3 140707607934616:error:20065064:BIO routines:BIO_accept:accept error:b_sock.c:860:
```

Listing 4.1: Procmon log: OS file handle limit exceeded

Network Monitor Agent
Netmon was useful for validating the model during development of the fuzzer. To be able to estimate its usefulness also for further fuzzing runs and to test the fuzzer's robustness a test was performed on OpenSSL's OCSP responder. For the test just the value fields of the model's TLV triplets were fuzzed resulting in around 100k test cases, which took 12 hours to execute on a test machine (OS: Ubuntu 16.04 64-bit, CPU: i5 4x2.5 GHz, RAM: 12 GiB, Swap: 10 GiB). During the test netmon's memory consumption exploded and filled the complete remaining memory and swap space. Also a CSV-log of 12.9 GB was produced. Therefore it was decided, that just the CSV-logger should be used for further tests, because it also contains the packets sent and received by the fuzzer making netmon redundant.
To further reduce the CSV-log's size then the fuzzed ASN.1 type's default values were excluded from the log, because they are already contained in the model. Thereby log filtering can be disabled again for future fuzzing runs to not lose potentially important information.
Additional, as planned in the design of the fuzzer's custom components then also test case skipping based on a crash threshold was implemented to avoid triggering the same crash multiple times by similar input data and slowing down the fuzzing run's duration. If for a specific field of the OCSP request then the crashes exceed this threshold the current mutation type's remaining test cases are skipped.

That the information logged by the CSV-logger is sufficient for the evaluation of the results, could be confirmed by the previously mentioned fuzzing run. Based on the CSV-log the detected crash of the fuzzer, which is visible in fig. 4.3 could be traced back and classified as false positive caused by a big request of around 8 MB, which had to be send in multiple packets and whose response was not detected due to a timeout.

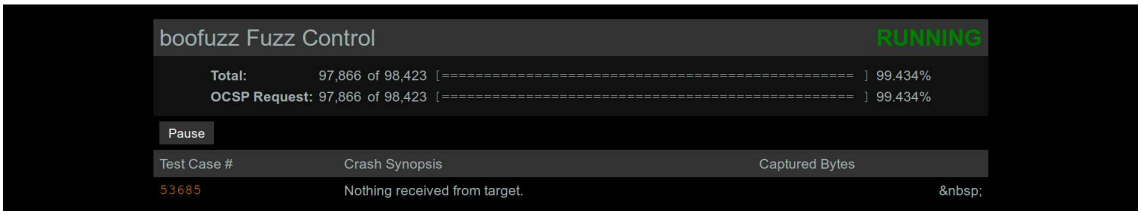


Figure 4.3.: Web monitor interface with the detected crash

Evaluation
To be able to efficiently evaluate the results of the fuzzing run additionally a CSV-log summarizer was implemented. It extracts the crash synopses and the requests, which caused them, from the produced log file and calculates statistics on the test cases' durations, request and response sizes.
To be able to replay and further investigate the requests, which caused crashes, also REST clients were added capable of reading in a request and sending it one or multiple times to the target.

5. Fuzzing Campaign

To evaluate the custom network fuzzer built in this thesis and verify assumption 1 a fuzzing campaign was rolled out on different targets.
During target selection in section 3.1 just the proprietary commercial OCSP responder of ESCRYPT was planned for this campaign, but it could not be set up in time. Therefore the OCSP responders of OpenSSL, OpenCA[1] and the Dogtag PKI[2] were considered as additional open-source targets for the fuzzing campaign from which the Dogtag PKI was excluded, because its setup was too complex in comparison with the other open-source targets.
To be able to compare the applicability of a generation- and a mutation-based model on a complex protocol like OCSP, both input creation approaches were applied on the targets and measures were selected to be able to compare them, as well as to evaluate the developed custom network fuzzer for OCSP responders. As standard for evaluation the U.S. American National Institute of Standards and Technology (NIST) proposes ISO 9241, which defines different evaluation criteria.
The usability is "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [40, p. 424]. The effectiveness is composed of the accuracy and completeness with which the specified goals are achieved and the efficiency is determined by the required resources in relation to the accuracy and completeness with which these goals were achieved. Thereby both the effectiveness and the efficiency can be measured using quantitative measures.
Qualitative measures like the satisfaction, which measures the "positive attitudes towards the use of the product" [40, p. 424] cannot be measured in "hard" data. They have to be time-consumingly measured using standardized post-task questionnaires [41]. Therefore for assessment of the fuzzer and the applicability of both models on OCSP just the following quantitative measures were selected:

[1] OpenCA OCSP Daemon, `https://github.com/openca/openca-ocspd`, Accessed: 2017-04-03
[2] Dogtag PKI, `http://www.dogtagpki.org/wiki/PKI_Main_Page`, Accessed: 2017-04-03

- **Effectiveness:**
  - **Goal of the fuzzing campaign:** Approval of assumption 1 by revealing implementation specific vulnerabilities of the selected fuzzing targets.
  - **Measures:**
    * amount of detected fuzzing bugs (hangs/crashes)
    * percentile of reproducible bugs within all detected fuzzing bugs
    * code coverage
- **Efficiency:**
  - total execution time of the fuzzing run
  - duration per test case
  - test duration per detected fuzzing bug

To be able to identify directly reproducible fuzzing bugs all requests causing bugs during the fuzzing run were extracted out of the produced CSV-log using the CSV-log summarizer of the fuzzer and replayed using its REST client like displayed in listing 5.1.

```
$ python2.7 csv_log_summarizer.py ../audits/openssl/log.csv
$ ./replay_requests.sh localhost 8088 openssl
../audits/openssl/requests/4.bin
10s timeout reached
../audits/openssl/requests/2468.bin
Duration: 0.001 seconds
```

Listing 5.1: CSV-log analysis and replay of detected fuzzing bugs for crash analysis

5.1. Audits

The audits for the different fuzzing targets were created using both input creation models. The generation-based model can be adapted with its configuration file. For the mutation-based model always the same known good mutation template was used containing a signed OCSP request requesting the revocation status of 4 certificates with 4 extensions defined per requested certificate and for the whole request. This template is mutated based on the built-in mutations of boofuzz applying byte- and bit-flips.

Therefore first for every target the configuration of its test environment is listed. Afterwards the results of the fuzzing run are presented followed by the crash analysis during which the detected fuzzing bugs were further analyzed and arranged into the template displayed in table 5.1. It groups duplicates and documents the results observed during replay of single test cases for which fuzzing bugs were detected. The review following the crash analysis then estimates, which bugs can be directly reproduced and exploited, as well as how the results of the audit have to be assessed.

Amount: < *number* >	< *mutant name* >	< *mutation type* >
Consequence:	< *description* >	
[**Result:**	< *description*, if the bug can be reproduced >]	

Table 5.1.: Crash analysis template to arrange fuzzing bugs

5.1.1. OpenSSL 1.0.2l

Test Environment:

- **Test machine:** OS: Ubuntu 16.04 64-bit, CPU: i5 4x2.5 GHz, RAM: 12 GiB, Swap: 10 GiB
- **Configuration of the generation-based model:**
  - Crash threshold: 5
  - Timeout: 10s
  - Fuzzable parts of the ASN.1 TLV triplet: Length, Value
  - Amount of combinatorial length × value mutations: **25** × **40**
  - Random mutations:
    - RandomField: initial tag = 22 (IA5String), mutations = **2000**
    - BitString: flip rate = **0.2%**, mutations = **2000**
- **Coverage measurement:** gcov, lcov for report generation

Results:

model	**generational**	**mutational**	
cases	204,475	109,984	
duration	29:52:21 h	5:52:29 h	
log size	13.2 GB	0.9 GB	
skipped	2,398	N/A	
success	202,061 (99.99%)	109,974 (99.99%)	
error	23 (0.01%)	10 (0.01%)	
fail	16 (0.01%)	114 (0.10%)	# test suite
covered lines	8.1%	6.4%	46.9%
covered functions	13.7%	11.5%	52.9%
coverage(ocsp.c)	36.9%	21.5%	19.9%
coverage(ocsp_asn.c)	26.7%	20.0%	20.0%

Table 5.2.: Results for fuzzing of OpenSSL 1.0.2l

Generation-based model:

measure	values	min	max	median	average	std	variance	sum
Durations in ms	202082	1	176919	881	532	785	616571	107540924
Request-Length in bytes	202061	146	4131308	2549	21108	94928	9011333964	4265153401
Response-Length in bytes	202045	84	1120	84	156	263	69097	31513752

Mutation-based model:

measure	values	min	max	median	average	std	variance	sum
Durations in ms	109983	0	24130	24	192	716	513035	21148147
Request-Length in bytes	109974	1062	1062	1062	1062	0	0	116792388
Response-Length in bytes	109860	84	1371	1354	1138	475	226083	125026214

Figure 5.1.: Statistics for fuzzing of OpenSSL 1.0.2l

Crash Analysis:

- **Generation-based model:**

Amount: 7	OCSPRequest (Sequence)	length
Consequence:	Responder Error: malformedrequest (ASN1_D2I_READ_BIO:not enough data)	
Result:	**Timeout**	

Amount: 1	type-id (ObjectIdentifier)	combinatorial
Consequence:	Responder Error: malformedrequest	

Amount: 6	othername_bmp (BMPString)	value
Consequence:	Successful response	

Amount: 10	ediPartyName_nameAssigner (UniversalString, BMPString)	value
Consequence:	Responder Error: malformedrequest	

Amount: 6	hashAlgorithm_algorithm (ObjectIdentifier)	value
Consequence:	Segmentation fault	
Result:	**Responder killed + core dumped**	

Amount: 5	singleRequestExtensions_sig-algs_id (ObjectIdentifier)	value
Consequence:	Responder Error: internalerror	

Amount: 4	requestExtensions_sig-algs_id (ObjectIdentifier)	combinatorial
Consequence:	Responder Error: malformedrequest	

- **Mutation-based model:**

Amount: 124	OCSP-Request (mutational)	mutation
Consequence:	Responder Error: malformedrequest (ASN1_D2I_READ_BIO:not enough data)	
Result:	**Timeout**	

Review:
The results of the fuzzing run displayed in table 5.2 and fig. 5.1 show, that with the generation-based model twice as many test cases were produced, but its fuzzing run took five times longer. This can be partly explained by the huge difference in size of the inputs generated for both fuzzing runs. As shown in fig. 5.1 the generation-based model produced requests with up to 4 MB, whereas the requests produced by the mutation-based model did not vary in size, because this model just applies byte-flips on the given mutation templates. Thereby the mutation-based model in total just produced 117 MB of input data, which can be faster processed as the 4.2 GB of input produced by the generation-based model in twice as many test cases. This also affected the size of the CSV-logs produced for both models visible in table 5.2, because they contain every request sent by the fuzzer.
The differences in the applicability of both models for a complex protocol like OCSP, which is defined in ASN.1 and encoded using ASN.1's DER, can be seen in the achieved code coverage. Using the generation-based model a 2% higher code coverage was achieved, than with the mutation-based model. The huge difference in coverage compared to OpenSSL's test suite is the result of the fuzzer being limited to the APIs offered by OpenSSL's OCSP responder application, which is just a small part of the whole SSL/TLS toolkit. Similar results also were observed by Au et al., who were fuzzing Android on the base of the APIs provided by its user interface (UI) [42].

In the relevant modules of OpenSSL (ocsp.c, ocsp_asn.c) both fuzzing runs were able to exceed the test suite's coverage. Here the generation-based model's better applicability could be proofed by achieving an even better coverage than the mutation-based model.

During crash analysis the single fuzzing bugs were analyzed in detail by replaying them using the fuzzer's REST client and documenting the results using the crash analysis template.
Both models were able to trigger a timeout by setting an invalid length for the outer `SEQUENCE` of the `OCSPRequest`. If it is set too high, the responder ignores the request's length set in the HTTP header and hangs while it waits for more data. This is a security vulnerability, because it can be used to trigger a DoS by replaying this type of request multiple times and not correctly terminating the connection to the OCSP responder. Then the responder not releases the acquired resources and will soon exceed its allowed file handle limit, as well as increase its memory footprint resulting in a crash similar to the one displayed in listing 4.1.
Additional fuzzing bugs just could be detected by the generation-based model. It was able to detect additional timeouts during fuzzing of `ObjectIdentifiers` and different universal string values. But during replay of single requests they could not be reproduced.
The most serious fuzzing bug was also detected with the generation-based model. During `ObjectIdentifier` fuzzing of a requested certificate's hash algorithm, a segmentation fault was detected by the OS causing it to kill the target and dump its core. This is a serious security incident, because it directly triggers a crash of the responder, which results in a DoS. The incident is demonstrated in fig. 5.2.

```
OCSP Request Data:
    Version: 0 (0xffffffffffffffff)
    Requestor Name: Registered ID2.39.65585.43.132.0.2.43.132.0.2.43
.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.1
32.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132
.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0
.2.43.132.0.2.43.132.0.2.43.132.0.2.43.132.0.2
    Requestor List:
        Certificate ID:
          Hash Algorithm: GOST 28147-89 MAC
          Issuer Name Hash: 5FFD580114477C1B40D91600EDB4857CF1AB8901
          Issuer Key Hash: E7C709486626586166E43DDF09B6756282A95A10
          Serial Number: 1000
./start_responder.sh: line 4:  4469 Segmentation fault      (core du
mped) openssl/apps/openssl ocsp -port 8088 -text -index CA/ca/demoCA
/index.txt -CA CA/ca/demoCA/cacert.pem -rkey CA/ca/ocsp-responder/pr
ivate_key.pem -rsigner CA/ca/ocsp-responder/certificate.crt -ignore_
err -reqout /tmp/ocsp-request.asn1
```

Figure 5.2.: OpenSSL 1.0.2l segmentation fault

A segmentation fault is the result of a memory access violation. Thereby the risk exists, that by specific modifications of the causing request also the target can be forced to execute remote code within its context [43]. Especially for open-source

projects with full source-code access this increases the seriousness of the vulnerability, because exploits can be prepared using a demo system, which prevents an early detection of the attack. But this requires further analysis of the target and the created core dump to identify the location of the access violation and gain information about it, which was not possible within the scope of this thesis.
Therefore the vulnerabilities detected for OpenSSL were communicated to OpenSSL's project leaders, that they can further analyze and fix them, before attackers are able to detect them and build exploits. But the vulnerability even will remain, when an updated version of OpenSSL is published, because it cannot be controlled when environments relying on OpenSSL will update their version in use.

5.1.2. OpenCA (2017-11-07)

Test Environment:

- **Test Machine:** OS: Fedora 27 64-bit, CPU: i7 8x2.4 GHz, RAM: 8 GiB, Swap: 8 GiB
- **Configuration of the generation-based model:**
  - Crash threshold: 5
  - Timeout: 10s
  - Fuzzable parts of the ASN.1 TLV triplet: Length, Value
  - Amount of combinatorial length $\times$ value mutations: $\mathbf{10 \times 20}$
  - Random mutations:
    - RandomField: initial tag = 22 (IA5String), mutations = **500**
    - BitString: flip rate = **0.01%**, mutations = **100**
- **Coverage measurement:** Not possible. Errors were raised, when the target was compiled with the flags required for gcov.

Results:

model	generational	mutational
cases	119,212	109,984
duration	24:49:30 h	30:22 min
log size	10.2 GB	0.6 GB
skipped	0	N/A
success	100%	100%
error/fail	0%	0%
coverage	N/A	N/A

Table 5.3.: Results for fuzzing of OpenCA (2017-11-07)

Generation-based model:

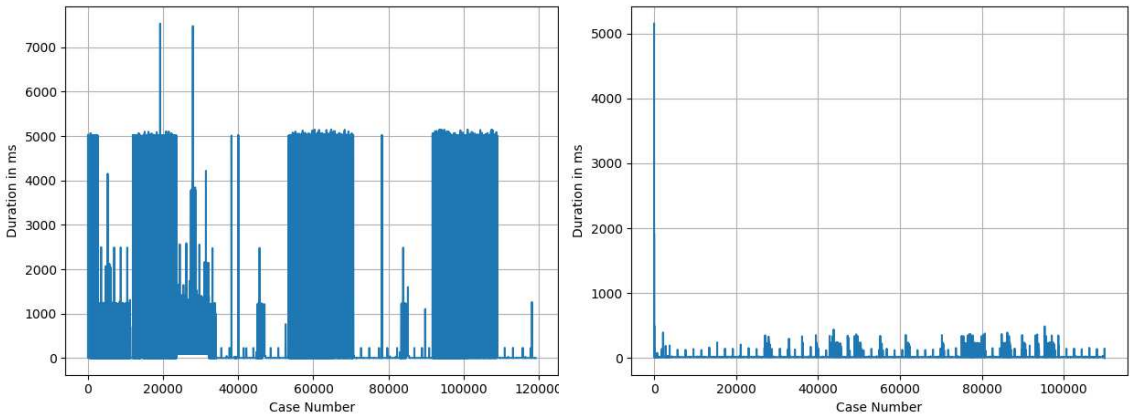
measure	values	min	max	median	average	std	variance	sum
Durations in ms	119211	0	7532	4	750	1755	3080824	89369295
Request-Length in bytes	119212	146	8131302	2292	27737	142880	20414570282	3306525307
Response-Length in bytes	119212	181	1891	181	322	444	197334	38364332

Mutation-based model:

measure	values	min	max	median	average	std	variance	sum
Durations in ms	109983	0	5152	16	17	19	379	1821893
Request-Length in bytes	109984	1062	1062	1062	1062	0	0	116803008
Response-Length in bytes	109984	181	1494	184	200	145	20893	21960410

Figure 5.3.: Statistics for fuzzing of OpenCA (2017-11-07)

Review:
For fuzzing of the OpenCA OCSP Daemon the size of the generation-based test set was reduced compared to the one of OpenSSL, because the random mutations were not able to trigger fuzzing bugs. Due to the same reason also the flip rate for `BitStrings` was set more fine granular.
Fig. 5.3 shows, that no fuzzing bugs were detected with both models. This could be the result of the responder not working correctly, because even valid requests were answered with the error response status `internalError`, if the requested certificate was revoked. This incident was further analyzed, but due to the lack in documentation it is still unclear, if it is a real bug of the responder or just its current configuration is invalid.
The cause of the differing execution times using both models can be seen in fig. 5.4. Again the generation-based model was able to trigger timeouts at the target, but unlike to OpenSSL for OpenCA a server-side timeout can be configured, which is set to 5 seconds by default. Therefore the fuzzer was not able to detect a timeout, because its timeout was set to 10 seconds resulting in the target first terminating the connection by returning an appropriate error response status.



(a) Generation-based model (b) Mutation-based model

Figure 5.4.: Test case durations for OpenCA (2017-11-07)

5.1.3. ESCRYPT

Test Environment:

- **Test machine:** OS: Ubuntu 16.04 64-bit (VirtualBox), CPU: i5 4x3.2 GHz, RAM: 4 GiB, Swap: 2 GiB
- **Configuration of the generation-based model:**
  - Crash threshold: 5
  - Timeout: 10s
  - Fuzzable parts of the ASN.1 TLV triplet: Length, Value
  - Amount of combinatorial length × value mutations: **10** × **20**
  - Random mutations:
    - RandomField: initial tag = 22 (IA5String), mutations = **500**
    - BitString: flip rate = **0.01%**, mutations = **100**
- **Coverage measurement:** JaCoCo, EclEmma for report generation

Results:

model	generational	mutational
cases	119,212	109,984
duration	2:34:30 h	40:36 min
log size	8.1 GB	0.5 GB
skipped	45,525	N/A
success	73,686 (99.15%)	109,983 (100%)
error	1 (0.00%)	0%
fail	634 (0.85%)	0%
covered lines	8.4%	8.6%
coverage(ocsp-common.jar)	40.3%	38.7%
coverage(ocsp-responder.jar)	37.5%	36.4%
coverage(ocsp-responder.jar > ocsp.backend)	46.8%	44.0%
coverage(iaik.jar)	7.9%	8.1%

Table 5.4.: Results for fuzzing of the proprietary commercial OCSP responder

Generation-based model:

measure	values	min	max	median	average	std	variance	sum
Durations in ms	73686	0	35436	8	126	941	885996	9269537
Request-Length in bytes	73686	152	8131300	2882	35888	145069	21044961705	2644450268
Response-Length in bytes	73052	103	174	108	107	3	11	7827833

Mutation-based model:

measure	values	min	max	median	average	std	variance	sum
Durations in ms	109983	0	35692	20	22	108	11695	2435316
Request-Length in bytes	109983	1068	1068	1068	1068	0	0	117461844
Response-Length in bytes	109983	108	108	108	108	0	0	11878164

Figure 5.5.: Statistics for fuzzing of the proprietary commercial OCSP responder

Crash Analysis:

- **Generation-based model:**

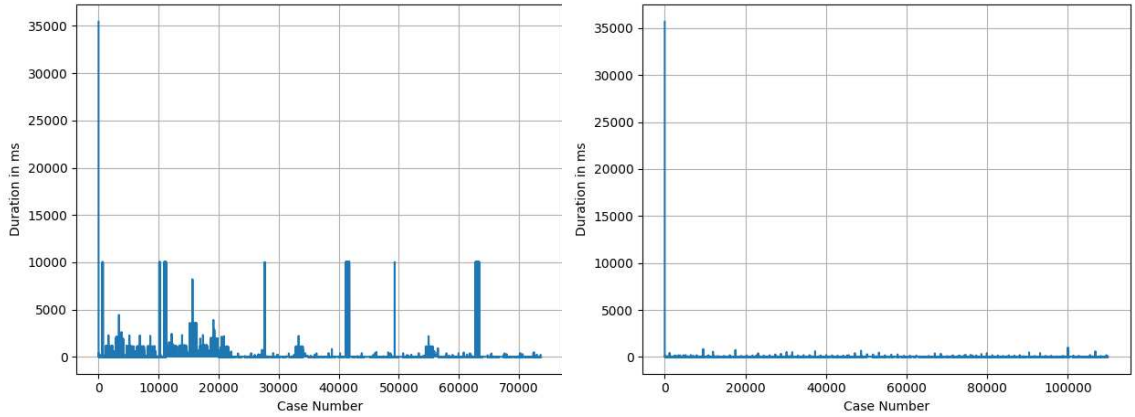
Amount: 499	requestExtension (service-locator)	length, value, combinatorial, optional, custom
Consequence:	OCSPResponderServlet - POST request parsing failed on HTTP level: HTTP_REQ__PATH_LENGTH path has wrong number of components. Min: 3 Max: 4 Found: 2	
Result:	**Timeout**	

Amount: 126	requestorName	length, value, combinatorial, optional, custom
Consequence:	OCSPResponderServlet - POST request parsing failed on HTTP level: HTTP_REQ__PATH_LENGTH path has wrong number of components. Min: 3 Max: 4 Found: 2	
Result:	**Timeout**	

Review:
Building a test environment for fuzzing of ESCRYPT's proprietary commercial OCSP responder turned out to be challenging. As introduced in section 3.1 ESCRYPT's KMS backend consists of multiple different trust zones. Due to the complexity of the backend and the hardware restrictions of the test machine therefore the KMS backend could not be set up completely. The Oracle database of the `Revocation Info DB` had to be replaced by Apache Derby and the dependencies to the `X.509 App`, as well as the `HSM` had to be cut off to be able to remove them. Thereby the `Revocation Info DB` could not be updated with new revocation information and the `OCSP Backend` has to sign its responses using a statically configured keystore. Because the `X.509 App` also instructs the responder about its authorized CAs, just the ASN.1 parsing code of the `OCSP Frontend` can be tested, as result of the responder not being able to successfully verify, if it is authorized to answer certificate status queries for the requested CAs. Therefore the responder declines every request after checking its CA.
After deploying the remaining parts of the OCSP responder in separate docker images, the `KMS Backend` still requires 2 minutes to start and the `OCSP Frontend` 30 seconds. Also the memory utilization of the test machine increases to 90%. Therefore the input creation models of the fuzzer were configured like for OpenCA with the difference, that the fuzzing target is not restarted anymore, when a timeout is detected. Coverage measurement was realized using JaCoCo's Java Agent[3].
Fig. 5.4 shows, that fuzzing bugs could just be detected with the generation-based model resulting in a longer duration of its fuzzing run. The line coverage achieved in total is nearly equal for both models, but in the relevant modules of the OCSP responder the generation-based model achieved an around 2% higher coverage. For the IAIK module, which is responsible for ASN.1 parsing and encoding, both models have a similar coverage with a slightly differing distribution. That the mutation-based model is better suited for fuzzing of this OCSP responder can be seen in fig. 5.5. The response-length did not vary for the mutation-based model, which is an indicator that the single requests were not able to cover different code paths of the target.
During crash analysis of the fuzzing bugs observed for the generation-based model it turned out, that by setting the OCSP response extension `service-locator` as request extension or the optional `requestorName` field timeouts can be triggered. But

[3]JaCoCo - Java Agent, `https://www.eclemma.org/jacoco/trunk/doc/agent.html`, Accessed: 2018-04-21

in comparison to OpenSSL the acquired resources are again successfully released, after the connection was dropped by the REST client used for replaying the single requests. Thereby this is not a security vulnerability, because triggering a DoS is only possible, if all connections of the responder are blocked and kept alive, which is possible for every OCSP responder. The impact of the timeouts on the fuzzing run's duration can be seen in fig. 5.6a. The duration of the first test case is longer for both models, because the `OCSP Frontend` has to be start up at the beginning of the test run.



(a) Generation-based model (b) Mutation-based model

Figure 5.6.: Test case durations for the proprietary commercial OCSP responder

Crashes could not be detected in this audit, but during analysis of the `OCSP Frontend`'s log 16 `java.lang.OutOfMemoryError`s could be detected, which were successfully caught and are a result of the test machine's hardware constraints. In earlier test runs still having a greater memory footprint, memory errors were observed for both the fuzzer and the `OCSP Frontend` requiring them to be restarted, because they were terminated with exit code 1.

5.2. Result

All in all the first audit of OpenSSL proofed, that the fuzzer built in this thesis is able to reveal security vulnerabilities. Thereby assumption 1 investigated in this thesis is correct for this audit, but this cannot be generalized. For the commercial OCSP responder no security vulnerability and for OpenCA not any fuzzing bug could be detected. This could be the consequence of the target not operating correctly, but also a fuzzer cannot test a primitive's whole value range. Therefore audits created by fuzzy testing are no proof, that a tested target has no security vulnerabilities. They just give a better understanding of the target's security level and its robustness to unknown inputs. This is the reason, why the results of the audit created for OpenCA

not revealing any fuzzing bug have to be treated with caution.
Additionally to the confirmation of the investigated assumption, table 5.5 shows, that most of the detected fuzzing bugs could be reproduced, which is an indicator for the fuzzer's effectiveness. This applies especially for the generation-based input creation model, which achieved a higher or at least similar line coverage compared to the mutation-based model for the fuzzed targets. For the generation-based model, the effectiveness could even be increased by disabling the skipping of test cases with the disadvantage of a reduced efficiency due to the additional detection of similar timeouts to the already detected ones. In addition by also replaying the last 100 preceding test cases for every not directly reproducible fuzzing bug the fuzzer's effectiveness could probably be further improved, but this could not be executed within the scope of this thesis.

	OpenSSL		**OpenCA**		**ESCRYPT**	
	generational	mutational	gen.	mut.	gen.	mut.
detected bugs	39	124	0	0	635	0
reproducible bugs	13 33%	124 100%	-	-	634 99.8%	-
code coverage	8.1%	6.4%	-	-	8.4%	8.6%
total duration	30h	6h	25h	0.5h	2.5h	41min
average duration	532ms	192ms	750ms	17ms	126ms	22ms
duration/bug	46min	10min	-	-	25min	-

Table 5.5.: Fuzzing campaign results

In terms of efficiency the mutation-based input creation method produced better results. The total duration of its fuzzing run took only a fraction of the one of the generation-based model with the disadvantage of detecting not any or only less interesting fuzzing bugs like timeouts. The efficiency of the generation-based model is lower, because it is better applicable for fuzzing of a complex protocol like OCSP. Especially for the audit of OpenCA this had an impact, because more timeouts were caused resulting in a 44 times longer average duration per test case for the generation-based model.
Finally, the single audits confirmed, that besides OpenCA all targets provide a good reliability towards valid input and also most of them are robust to invalid input by reliably parsing it without directly causing a DoS [44]. Just one type of request could be found triggering a segmentation fault for OpenSSL and thereby directly terminating the target and causing a DoS.

6. Current State of the Art

This chapter introduces the related works, which were previously available in the domain of this thesis. Afterwards a brief outline is given with ideas how this work could be continued.

6.1. Related Works

Hagelkruys [45] already picked up the idea of taking an OCSP responder as fuzzing target in 2014. He wrote simple code snippets in Python generating binary data to be send as OCSP request to an OCSP responder using HTTP-POST. For generation-based fuzzing he generated random binary data of varying length and for mutation-based fuzzing he replaced a randomly selected data block of an existing OCSP request with random data of varying length. But this approach was not extended with any monitoring capabilities to be able to execute full fuzzy testing.

One step further in fuzzing went Marquis et al. [46]. They performed syntax-based black-box fuzzing of an ASN.1 based network protocol by defining a Structure and Context-Sensitive language (SCL). With the grammar of this language they could formally describe a protocol's syntax and semantic constraints used to decode the protocol's binary packets into textual form. Thereby they were able to built a model to automate the generation-based test case creation. Mutation-based input did they generate by mutating valid data, captured previously from the network.

Another work to be mentioned is the one of Rawat et al. [47]. They underlined the high potential of fuzzing and set new standards with the publication of VUzzer at the beginning of 2017. VUzzer is a mutation-based, application-aware and evolutionary fuzzer not requiring prior knowledge of the fuzzed target or its input format to be able to maximize its code coverage and efficiently explore its interesting execution paths. This is achieved through static and dynamic analysis of the target application to prioritize more interesting deep code paths not resulting in error-handling code. Therefore they added a feedback loop, which evaluates the fuzzed input based on its ability to discover interesting new code paths. It also enables the selection of the best performing inputs as parents for the generation of new inputs by crossover.
So far tracing paths by symbolic and concolic execution was challenging for large targets, because of their limited scalability due to state explosion. This constraint did they solve by applying Dynamic Taint Analysis (DTA) on the target's binary

code to determine the memory locations and registers affected by the fuzzed input during runtime. Thereby they can monitor the target, which means VUzzer has to be categorized as gray-box fuzzer, because of its partial code access. The lightweight data-flow features gained from DTA differentiate VUzzer from other already existing fuzzers, because they enable an effective input selection based on heuristics making heavyweight program analysis techniques obsolete.

From another point of view OCSP was regarded by Ivanov [2]. He searched for flaws in the protocol's design[1] instead of in its implementation and identified several weaknesses.
One weakness was the predictability of responses. Because the defined extensions can be controlled in the request and if the status of the requested certificate is unchanged, only the responses' time values differ during multiple requests. Their accuracy is not exactly specified and in practice all responders assessed by Ivanov just provided an accuracy of one second, which is also specified for OCSP's lightweight profile[2]. Thereby the value of the response's `tbsResponseData` can be predicted dependent on the point in time, when the request was sent and its hash can be calculated in advance. By that the responder can be brought to sign any hash value, if a hash collision was found and the request was submitted at the correct point in time. The detection of this attack is nearly impossible due to the minimal interaction with the victim.
To scale this attack the `nonce` extension containing a random salt can be used to increase the number of possible differing `tbsResponseData` values at a specific point in time. Further expansion of this attack is possible, if the CA's certificate is used to sign OCSP responses. Then even a valid signature for a X.509 certificate could be created and if its serial number is shared with a legitimate one it is impossible for the OCSP responder to distinguish them. Therefore random serial numbers should be assigned to the certificates issued by a CA to prevent their prediction.
That this kind of attack is possible the responses must be computed in real-time, which was observed at 75% of the 70 OCSP responders assessed by Invanov. Additionally the used hash function must be susceptible for collision attacks or it has to be brute-forced. SHA-1 is used by 57% of his assessed responders. Among those, 41% also support response scaling using the `nonce` extension, which enables low-cost collision attacks with a cloud resource budget of less than $120K [48]. Thereby this attack gets a serious incident, if the CA's certificate is used to sign the OCSP responses in combination with response scaling. This setting enables the generation of fake certificates and was chosen by 1 of the 70 responders assessed by Ivanov.

[1] OCSP's ASN.1 specification: Appendix A
[2] Lightweight OCSP Profile: Section 2.2.3

6.2. Future Works

That the custom black-box fuzzer for protocol-based fuzzing of OCSP responders can be used to reveal security vulnerabilities in OCSP responders, was successfully demonstrated by the results of the fuzzing campaign in chapter 5. Based on these results different approaches exist, which can be followed to investigate, if they are able to improve the already achieved results.
First, the few remaining optional parts of the OCSP request, which were not modeled could be added to the generation-based model like the client certificate[3] used to sign the request. They were excluded, because I assume that due to their rare use many OCSP responders not implement them and thereby the fuzzer cannot hit additional code paths with them.
Additionally, the mutation-based model, which was not the focus of this thesis, can be improved by not just relying on the single byte mutations of boofuzz. More fine granular mutations could be implemented removing single bytes or replacing them by an arbitrary number of interesting other bytes, as well as additional mutation templates could be added to achieve a varying request size and improve the coverage of the target.
But more interesting would it be to investigate the effect of smart fuzzing and also verify the OCSP responses. The advantages of an application-aware evolutionary fuzzing strategy could be investigated by adding a feedback loop to the fuzzer and performing static and dynamic code analysis. Thereby the hit code paths could be tracked and interesting ones prioritized to maximize the target's coverage like it was performed by VUzzer [47], which is introduced in the previous section.
Furthermore, to be able to detect more and also less severe memory access violations an address sanitizer could be compiled into the target to be able to detect buffer overflows not directly causing a crash of the fuzzed target like it was proposed by Böck[4].
Also to maximize the performance of parsing code fuzzing for single targets the approach of Acri [49] could be investigated, who performed in-memory fuzzing by modifying the target and isolating its parsing subroutine to be able to insert an infinite mutation loop. Thereby the parsing code can be tested in the fastest way by feeding it with a large number of fuzzed inputs. Disadvantage of this approach is its complexity, because the target has to be reverse engineered to be able to insert the mutation loop or breakpoints to restore the target's initial state for every test case. This is very expensive, because the approach has to be individually implemented for every target.

[3] see `certs` in appendix A.1
[4] The Fuzzing Project - Find more Bugs with Address Sanitizer, `https://fuzzing-project.org/tutorial2.html`, Accessed: 2017-04-22

Another way to continue this work not directly related to the fuzzer would be to further investigate the the detected fuzzing bugs. The not directly reproducible ones could be replayed with their preceding test cases to check their influence on the results, as well as the security vulnerabilities identified for OpenSSL could be further investigated. Especially the detected segmentation fault could be analyzed in detail to investigate, if an exploit can be built for it. But since the developers of OpenSSL already were informed about the identified vulnerabilities its impact will low.

7. Conclusion

In this thesis a penetration testing framework for fuzzy testing of OCSP responders was developed to be able to investigate and improve the reliability and fail safety of OCSP responders by identifying their security vulnerabilities. Therefore first OCSP was analyzed in detail including its encoding rules (ASN.1 DER) and design related vulnerabilities, before different fuzzing approaches and open-source frameworks could be considered
After the selection of generation-based fuzzing as the preferred approach for the creation of the fuzzed inputs, boofuzz was chosen as base to build the custom protocol-based black-box network fuzzer. Its generation-based model directly was evaluated and compared with a second mutation-based model within a fuzzing campaign rolled out on two open-source and a proprietary commercial target. Thereby the fuzzer's effectiveness could be proofed and that it is able to reveal security vulnerabilities. Additionally, the better applicability of a generation-based input creation model for a complex protocol like OCSP could be successfully confirmed.
Thereby the applicability of the fuzzing framework developed in this thesis on arbitrary OCSP responders to investigate them for fuzzing bugs and reveal security vulnerabilities was proofed resulting in the following contributions and limitations.

7.1. Contributions

The main contributions of this thesis, which previously were not available are:

- **Detailed and compact explanation of ASN.1's Distinguished Encoding Rules (DER):** Their understanding is a prerequisite to be able to encode OCSP. Previously no such document was available. Therefore its content had to be time-consumingly collected out of several partially contradictory sources, which is the reason why chapter 2.1 explains them in such detail.

- **Custom OCSP fuzzer:** The fuzzer build in this thesis can be used directly to test arbitrary OCSP responders for security vulnerabilities as proofed in the fuzzing campaign described in chapter 5.

- Detection of implementation related security vulnerabilities in OCSP responders and communication of these vulnerabilities to the responsible persons.

7.2. Limitations

The chosen approach based on fuzzy testing also has limitations. A fuzzer is just able to efficiently test special interesting parts of a primitive's value range, which likely trigger code defect patterns. Thereby it cannot be the only tool for a full penetration test, whose quality also relies on the results produced by other tools and the creativity, as well as the experience of the executing penetration tester, who selects the tools in use and interprets their results. Therefore during building of a generation-based fuzzer the main attention should always lie on the single primitives' fuzz libraries, that they cover the value ranges most likely triggering code defect patterns.
Furthermore, it must be clear that the results of the single audits cannot be generalized and transferred to other targets. Therefore every single software target has to be assessed separately.

Anti-Fuzzing

Additionally, a fuzzed target can manipulate the results of fuzzy testing by anti-fuzzing techniques masking vulnerabilities. Thereby the return of investment and the performance of fuzzy testing is decreased, that other more promising targets are preferred by penetration testers for more extensive fuzzing runs. This slows down fuzzy testing with the goal to decrease the number of explored code paths and to hide security vulnerabilities until they were fixed in an improved version of the target following the current one.
The target realizes, that it is currently fuzzed by observing the execution environment, which is often virtualized for testing purposes, or the data source, whose submission rate during fuzzing normally deviates of the expected one. Additionally, the detection of fuzzing is possible by the increasing error rate in data processing or by the traversal of unusual code path.
Then temporarily an exception handler can be registered catching every unhandled exception or raising fake crashes. Thereby the target looks not exploitable or analysis time of the penetration tester is wasted for debugging of uninteresting fake crashes. Another way of fooling the fuzzer after its detection is the replacement of its inputs with random safe ones.
Disadvantages of anti-fuzzing are, that it just slows down attacks by hiding the target's vulnerabilities and not fixes them. Also anti-fuzzing is only possible for closed-source targets, because otherwise its detection would be trivial. Furthermore, anti-fuzzing also complicates crash debugging and in case of misbehavior it also negatively affects legitimate data sources. Therefore it has to be carefully implemented, which binds development and test resources which could otherwise be used to fix bugs causing security vulnerabilities [50].

A. ASN.1 Specification

This appendix contains the ASN.1 specification of the data exchanged by OCSP, which was specified by Santesson et al. [19].

A.1. OCSP Request

```
OCSPRequest     ::=     SEQUENCE {
    tbsRequest                  TBSRequest,
    optionalSignature   [0]     EXPLICIT Signature OPTIONAL }

TBSRequest      ::=     SEQUENCE {
    version             [0]     EXPLICIT Version DEFAULT v1,
    requestorName       [1]     EXPLICIT GeneralName OPTIONAL,
    requestList                 SEQUENCE OF Request,
    requestExtensions   [2]     EXPLICIT Extensions OPTIONAL }

Signature       ::=     SEQUENCE {
    signatureAlgorithm      AlgorithmIdentifier,
    signature               BIT STRING,
    certs               [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL}

Version         ::=             INTEGER  {  v1(0) }

Request         ::=     SEQUENCE {
    reqCert                     CertID,
    singleRequestExtensions     [0] EXPLICIT Extensions OPTIONAL }

CertID          ::=     SEQUENCE {
    hashAlgorithm       AlgorithmIdentifier,
    issuerNameHash      OCTET STRING, -- Hash of issuer's DN
    issuerKeyHash       OCTET STRING, -- Hash of issuer's public key
    serialNumber        CertificateSerialNumber }
```

Listing A.1: ASN.1 specification of the OCSP request

A.1.1. Additional constructed types of the OCSP request

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY DEFINED BY algorithm OPTIONAL }

-- Source: http://luca.ntop.org/Teaching/Appunti/asn1.html
```

Listing A.2: Additional constructed types of the OCSP request (1/4)

```
GeneralName ::= CHOICE {
   otherName                [0] OtherName,
   rfc822Name               [1] IA5String,
   dNSName                  [2] IA5String,
   x400Address              [3] ORAddress,
   directoryName            [4] Name,
   ediPartyName             [5] EDIPartyName,
   uniformResourceLocator   [6] IA5String,
   iPAddress                [7] OCTET STRING,
   registeredID             [8] OBJECT IDENTIFIER }

OtherName ::= SEQUENCE {
        type-id    OBJECT IDENTIFIER,
        value      [0] EXPLICIT ANY DEFINED BY type-id }

EDIPartyName ::= SEQUENCE {
    nameAssigner            [0]     DirectoryString OPTIONAL,
    partyName               [1]     DirectoryString }

DirectoryString ::= CHOICE {
      teletexString       TeletexString   (SIZE (1..MAX)),
      printableString     PrintableString (SIZE (1..MAX)),
      universalString     UniversalString (SIZE (1..MAX)),
      utf8String          UTF8String      (SIZE (1..MAX)),
      bmpString           BMPString       (SIZE (1..MAX)) }

-- Source: https://tools.ietf.org/html/rfc5280
```

Listing A.3: Additional constructed types of the OCSP request (2/4)

```
Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE {
        extnID      OBJECT IDENTIFIER,
        critical    BOOLEAN DEFAULT FALSE,
        extnValue   OCTET STRING }

-- Source: https://tools.ietf.org/html/rfc2459
```

Listing A.4: Additional constructed types of the OCSP request (3/4)

```
CertificateSerialNumber  ::=  INTEGER

Certificate  ::=  SEQUENCE {
        tbsCertificate       TBSCertificate,
        signatureAlgorithm   AlgorithmIdentifier,
        signatureValue       BIT STRING }

TBSCertificate  ::=  SEQUENCE {
        version         [0]  EXPLICIT Version DEFAULT v1,
        serialNumber         CertificateSerialNumber,
        signature            AlgorithmIdentifier,
        issuer               Name,
        validity             Validity,
        subject              Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                             -- If present, version MUST be v3
        }

```

```
Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

Validity ::= SEQUENCE {
        notBefore      Time,
        notAfter       Time }

Time ::= CHOICE {
    utcTime        UTCTime,
    generalTime    GeneralizedTime }

UniqueIdentifier  ::=  BIT STRING

SubjectPublicKeyInfo  ::=  SEQUENCE  {
        algorithm            AlgorithmIdentifier,
        subjectPublicKey     BIT STRING  }

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
        extnID      OBJECT IDENTIFIER,
        critical    BOOLEAN DEFAULT FALSE,
        extnValue   OCTET STRING
                    -- contains the DER encoding of an ASN.1 value
                    -- corresponding to the extension type identified
                    -- by extnID
        }

Name ::= CHOICE { -- only one possibility for now --
     rdnSequence  RDNSequence }

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET SIZE (1..MAX) OF AttributeTypeAndValue

AttributeTypeAndValue ::= SEQUENCE {
     type     AttributeType,
     value    AttributeValue }

AttributeType ::= OBJECT IDENTIFIER

AttributeValue ::= ANY -- DEFINED BY AttributeType

-- Source: https://tools.ietf.org/html/rfc5280
```

Listing A.5: Additional constructed types of the OCSP request (4/4)

A.2. OCSP Response

```
OCSPResponse ::= SEQUENCE {
    responseStatus         OCSPResponseStatus,
    responseBytes          [0] EXPLICIT ResponseBytes OPTIONAL }

OCSPResponseStatus ::= ENUMERATED {
    successful            (0),  -- Response has valid confirmations
    malformedRequest      (1),  -- Illegal confirmation request (not OCSP syntax
        conform)
    internalError         (2),  -- Service exists but temporarily unable to respond
    tryLater              (3),  -- Try again later
    -- (4) is not used
    sigRequired           (5),  -- Client must sign the request
    unauthorized          (6)   -- Request unauthorized -> client not authorized to
         query server or server cannot respond authoritatively }
```

```
13
14 ResponseBytes ::=       SEQUENCE {
15     responseType   OBJECT IDENTIFIER,  -- OID encoded as OCTET STRING
16     response       OCTET STRING }
17
18 BasicOCSPResponse       ::= SEQUENCE {
19     tbsResponseData      ResponseData,
20     signatureAlgorithm   AlgorithmIdentifier,
21     signature            BIT STRING,
22     certs            [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL }
23
24 ResponseData ::= SEQUENCE {
25     version              [0] EXPLICIT Version DEFAULT v1,
26     responderID              ResponderID,
27     producedAt               GeneralizedTime,
28     responses                SEQUENCE OF SingleResponse,
29     responseExtensions   [1] EXPLICIT Extensions OPTIONAL }
30
31 ResponderID ::= CHOICE {
32     byName               [1] Name,
33     byKey                [2] KeyHash }
34
35 KeyHash ::= OCTET STRING -- SHA-1 hash of responder's public key (excluding the tag
       and length fields)
36
37 SingleResponse ::= SEQUENCE {
38     certID                       CertID,
39     certStatus                   CertStatus,
40     thisUpdate                   GeneralizedTime,
41     nextUpdate         [0]       EXPLICIT GeneralizedTime OPTIONAL,
42     singleExtensions   [1]       EXPLICIT Extensions OPTIONAL }
43
44 CertStatus ::= CHOICE {
45     good        [0]     IMPLICIT NULL,
46     revoked     [1]     IMPLICIT RevokedInfo,
47     unknown     [2]     IMPLICIT UnknownInfo }
48
49 RevokedInfo ::= SEQUENCE {
50     revocationTime              GeneralizedTime,
51     revocationReason    [0]     EXPLICIT CRLReason OPTIONAL }
52
53 UnknownInfo ::= NULL
```

Listing A.6: ASN.1 specification of the OCSP response

A.3. OIDs

```
1  -- IETF RFC 6960
2  id-kp-OCSPSigning            OBJECT IDENTIFIER ::= { id-kp 9 }
3  id-pkix-ocsp                 OBJECT IDENTIFIER ::= { id-ad-ocsp }
4  id-pkix-ocsp-basic           OBJECT IDENTIFIER ::= { id-pkix-ocsp 1 }
5  id-pkix-ocsp-nonce           OBJECT IDENTIFIER ::= { id-pkix-ocsp 2 }
6  id-pkix-ocsp-crl             OBJECT IDENTIFIER ::= { id-pkix-ocsp 3 }
7  id-pkix-ocsp-response        OBJECT IDENTIFIER ::= { id-pkix-ocsp 4 }
8  id-pkix-ocsp-nocheck         OBJECT IDENTIFIER ::= { id-pkix-ocsp 5 }
9  id-pkix-ocsp-archive-cutoff  OBJECT IDENTIFIER ::= { id-pkix-ocsp 6 }
10 id-pkix-ocsp-service-locator OBJECT IDENTIFIER ::= { id-pkix-ocsp 7 }
11 id-pkix-ocsp-pref-sig-algs   OBJECT IDENTIFIER ::= { id-pkix-ocsp 8 }
12 id-pkix-ocsp-extended-revoke OBJECT IDENTIFIER ::= { id-pkix-ocsp 9 }
13
```

```
14 -- German Common PKI Profile
15 id-commonpki-at-certHash      OBJECT IDENTIFIER ::= {id-commonpki-at 13}
```

Listing A.7: ASN.1 specification of the OIDs used by OCSP extensions

B. Dummy OCSP Responders

B.1. Runtime Error

```
# src: https://pymotw.com/2/socket/tcp.html
import socket, sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Bind the socket to the port
server_address = ('localhost', 8088)
print >> sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)
sock.listen(1)  # Listen for incoming connections

def infinite_recursion(depth=0):
    print depth
    infinite_recursion(depth=depth + 1)

while True:  # Wait for a connection
    print >> sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()

    try:
        print >> sys.stderr, 'connection from', client_address
        infinite_recursion()
    finally:  # Clean up the connection
        connection.close()
```

Listing B.1: Dummy OCSP responder with an infinite recursive loop

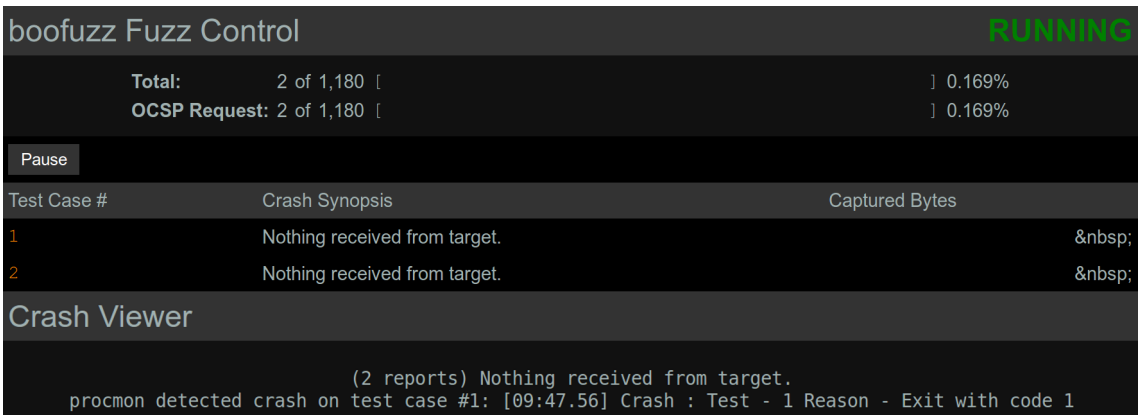


Figure B.1.: Web Monitor Interface

```
[11:24.49] Process Monitor PED-RPC server initialized:
[11:24.49] Listening on 0.0.0.0:26002
[11:24.49] awaiting requests...
[11:24.58] updating start commands to: ['python2.7 dummy_ocsp_responder.py']
[11:24.58] updating stop commands to: ['kill -9 $(fuser 8088/tcp 2>/dev/null)']
[11:24.58] starting target process
['python2.7', 'dummy_ocsp_responder.py']
[11:24.58] done. target up and running, giving it 5 seconds to settle in.
starting up on localhost port 8088
waiting for a connection
connection from ('127.0.0.1', 44354)
0...998
Traceback (most recent call last):
  File "/home/stefan/master-thesis/0_snippets/python/dummy_ocsp_responder.py", line
      28, in <module>
    infinite_recursion()
  File "/home/stefan/master-thesis/0_snippets/python/dummy_ocsp_responder.py", line
      18, in infinite_recursion
    infinite_recursion(depth=depth + 1)
  ...
RuntimeError: maximum recursion depth exceeded
[11:25.04] stopping target process
[11:25.05] starting target process
['python2.7', 'dummy_ocsp_responder.py']
[11:25.05] done. target up and running, giving it 5 seconds to settle in.
starting up on localhost port 8088
waiting for a connection
[11:25.13] updating start commands to: ['python2.7 dummy_ocsp_responder.py']
[11:25.13] updating stop commands to: ['kill -9 $(fuser 8088/tcp 2>/dev/null)']
connection from ('127.0.0.1', 44368)
0...998
Traceback (most recent call last):
  File "/home/stefan/master-thesis/0_snippets/python/dummy_ocsp_responder.py", line
      28, in <module>
    infinite_recursion()
  File "/home/stefan/master-thesis/0_snippets/python/dummy_ocsp_responder.py", line
      18, in infinite_recursion
    infinite_recursion(depth=depth + 1)
  ...
RuntimeError: maximum recursion depth exceeded
[11:25.14] stopping target process
[11:25.14] starting target process
...
```

Listing B.2: Procmon Log

B.2. Segmentation Fault

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include<string.h>

int main() { // compile: gcc -g -o server server.c
    printf("Starting segfault server");
    char str[100];
    int listen_fd, comm_fd;
    struct sockaddr_in servaddr;

    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
```

```
14      bzero( &servaddr, sizeof(servaddr));
15      servaddr.sin_family = AF_INET;
16      servaddr.sin_addr.s_addr = htons(INADDR_ANY);
17      servaddr.sin_port = htons(8088);
18
19      bind(listen_fd, (struct sockaddr *) &servaddr, sizeof(servaddr));
20      listen(listen_fd, 10);
21      comm_fd = accept(listen_fd, (struct sockaddr*) NULL, NULL);
22
23      while(1) {
24          bzero( str, 100);
25          read(comm_fd,str,100);
26          printf("Echoing back - %s",str);
27
28          // raise segmentation fault
29          char *s="hello world";
30          *s='H';
31
32          write(comm_fd, str, strlen(str)+1);
33      }
34  }
```

Listing B.3: Dummy OCSP responder with a segmentation fault

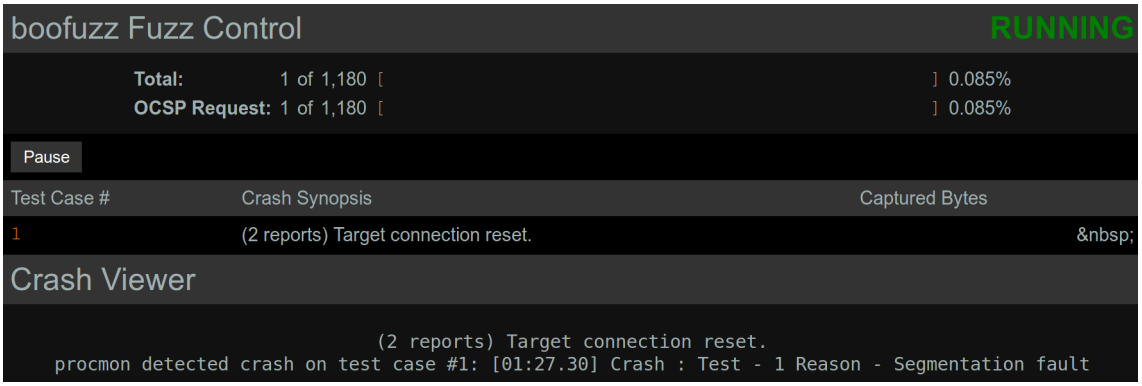


Figure B.2.: Web Monitor Interface

```
1   [01:26.57] Process Monitor PED-RPC server initialized:
2   [01:26.57] Listening on 0.0.0.0:26002
3   [01:26.57] awaiting requests...
4   [01:27.25] updating start commands to: ['server']
5   [01:27.25] updating stop commands to: ['kill -9 $(fuser 8088/tcp 2>/dev/null)']
6   [01:27.25] starting target process
7   ['server']
8   [01:27.25] done. target up and running, giving it 5 seconds to settle in.
9   [01:27.30] moving core dump ./core -> audits/coredumps/1
10  [01:27.31] stopping target process
11  ...
```

Listing B.4: Procmon Log

List of Figures

List of Tables

Acronyms

ASN.1 Abstract Syntax Notation One.

BER Basic Encoding Rules.

BSI Federal Office for Information Security.

CA Certification Authority.

CNSS Committee on national security systems.

CRL Certificate Revocation List.

CT Certificate Transparency.

DER Distinguished Encoding Rules.

DoS Denial of Service.

DTA Dynamic Taint Analysis.

GUI Graphical User Interface.

HSM Hardware Security Module.

IA5 International Alphabet number 5.

IETF Internet Engineering Task Force.

IoT Internet of Things.

KMS Key Management Service.

MMD Maximum Merge Delay.

netmon network monitor agent.

NIST National Institute of Standards and Technology.

OCSP Online Certificate Status Protocol.

OID Object Identifier.

OS operating system.

Peach CE Peach Community Edition.

PKI Public Key Infrastructure.

procmon process monitor agent.

RTT Round-Trip Time.

SCL Structure and Context-Sensitive language.

SCT Signed Certificate Timestamp.

SCVP Server-Based Certificate Validation Protocol.

SDL Secure Development Lifecycle.

SID sub identifier.

TLS Transport Layer Security.

TLV triplet tag-length-value triplet.

UCS Universal Character Set.

UI user interface.

URI Uniform Resource Identifier.

UTC Universal Time Coordinate.

VM Virtual Machine.

vmcontrol VMware control agent.

Bibliography

[1] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Retrieved from `https://tools.ietf.org/pdf/rfc5280.pdf`, 2008. Accessed: 2017-09-22.

[2] Ken Ivanov. Autonomous collision attack on OCSP services. *CoRR*, abs/1609.03047, 2016.

[3] BSI TR03125 - Beweiswerterhaltung kryptographisch signierter Dokumente. Retrieved from `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03125/BSI_TR_03125-V1_2.pdf?__blob=publicationFile&v=1`, 2014. Accessed: 2018-02-23.

[4] Manuel Koschuch and Ronald Wagner. *Trust Revoked — Practical Evaluation of OCSP- and CRL-Checking Implementations*, pages 26–33. Springer International Publishing, Cham, 2015.

[5] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 291–304, New York, NY, USA, 2013. ACM.

[6] R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280 - Certificate and CRL Profile. Retrieved from `https://buildbot.tools.ietf.org/pdf/rfc3280.pdf`, 2002. Accessed: 2017-09-23.

[7] Christopher Domas. Breaking the x86 ISA. Retrieved from `https://github.com/xoreaxeaxeax/sandsifter/raw/master/references/domas_breaking_the_x86_isa_wp.pdf`, 2017. Accessed: 2017-11-21.

[8] Olivier Dubuisson and Philippe Fouquart. ASN. 1: communication between heterogeneous systems. 2001.

[9] Burton S Kaliski Jr. A Layman's Guide to a Subset of ASN. 1, BER, and DER. Retrieved from `http://luca.ntop.org/Teaching/Appunti/asn1.html`, 1993. Accessed: 2018-01-08.

[10] Encoded Length and Value Bytes . Retrieved from `https://msdn.microsoft.com/de-de/library/windows/desktop/bb648641(v=vs.85).aspx`. Accessed: 2018-01-19.

[11] ITU-T X.690 - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). Retrieved from `http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf`, 2002. Accessed: 2018-01-29.

[12] INTEGER. Retrieved from `https://msdn.microsoft.com/de-de/library/windows/desktop/bb540806(v=vs.85).aspx`. Accessed: 2018-01-19.

[13] BIT STRING. Retrieved from `https://msdn.microsoft.com/en-us/library/windows/desktop/bb540792(v=vs.85).aspx`. Accessed: 2018-02-06.

[14] OCTET STRING. Retrieved from `https://msdn.microsoft.com/de-de/library/windows/desktop/bb648644(v=vs.85).aspx`. Accessed: 2018-01-19.

[15] NULL. Retrieved from `https://msdn.microsoft.com/en-us/library/windows/desktop/bb540808(v=vs.85).aspx`. Accessed: 2018-02-02.

[16] Yury Strozhevsky. ASN.1 by simple words. Retrieved from `https://www.strozhevsky.com/free_docs/asn1_by_simple_words.pdf`, 2012. Accessed: 2018-01-24.

[17] OBJECT IDENTIFIER. Retrieved from `https://msdn.microsoft.com/de-de/library/windows/desktop/bb540809(v=vs.85).aspx`. Accessed: 2018-01-25.

[18] String Types. Retrieved from `https://msdn.microsoft.com/de-de/library/windows/desktop/bb540814(v=vs.85).aspx`. Accessed: 2018-02-07.

[19] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. RFC 6960 - X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. Retrieved from `https://buildbot.tools.ietf.org/pdf/rfc6960.pdf`, 2013. Accessed: 2017-09-22.

[20] CertHash - IAIK. Retrieved from `http://javadoc.iaik.tugraz.at/iaik_jce/current/iaik/x509/ocsp/extensions/commonpki/CertHash.html`. Accessed: 2018-04-20.

[21] CertHash - Bouncy Castle. Retrieved from `https://people.eecs.berkeley.edu/~jonah/bc/org/bouncycastle/asn1/isismtt/ocsp/CertHash.html`. Accessed: 2018-04-20.

[22] A. Deacon and R. Hurst. RFC 5019 - The Lightweight Online Certificate Status Protocol (OCSP) Profile for High-Volume Environments. Retrieved from `https://tools.ietf.org/pdf/rfc5019.pdf`, 2007. Accessed: 2017-11-08.

[23] D. Eastlake. RFC 6066 - Transport Layer Security (TLS) Extensions Definitions. Retrieved from `https://tools.ietf.org/pdf/rfc6066.pdf`, 2011. Accessed: 2017-12-11.

[24] P. Hallam-Baker. RFC 7633 - X.509v3 Transport Layer Security (TLS) Feature Extension. Retrieved from `https://tools.ietf.org/pdf/rfc7633.pdf`, 2015. Accessed: 2017-12-08.

[25] P. Black and R. Layton. Be Careful Who You Trust: Issues with the Public Key Infrastructure. In *2014 Fifth Cybercrime and Trustworthy Computing Conference*, pages 12–21, Nov 2014.

[26] Certificate Transparency. Retrieved from `https://www.certificate-transparency.org/`. Accessed: 2018-02-23.

[27] Emily Stark. Is Certificate Transparency usable (RWC 2018). Retrieved from `https://www.youtube.com/watch?v=e_rwG7MA5VU`, January 2018. Accessed: 2018-02-27.

[28] How Certificate Transparency Works - Certificate Transparency. Retrieved from `https://www.certificate-transparency.org/how-ct-works`. Accessed: 2018-02-28.

[29] B. Laurie, A. Langley, and E. Kasper. RFC 6962 - Certificate Transparency. Retrieved from `https://buildbot.tools.ietf.org/pdf/rfc6962.pdf`, 2013. Accessed: 2018-02-28.

[30] T. Rontti, A. M. Juuso, and A. Takanen. Preventing DoS attacks in NGN networks with proactive specification-based fuzzing. *IEEE Communications Magazine*, 50(9):164–170, September 2012.

[31] Charlie Miller and Zachary NJ Peterson. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 2007.

[32] Michael Eddington. Demystifying Fuzzers. *Black Hat Europe*, 2009.

[33] John Neystadt. Automated penetration testing with white-box fuzzing. *MSDN Library*, 2008.

[34] Scapy. Retrieved from `http://www.secdev.org/projects/scapy/`. Accessed: 2017-12-12.

[35] Joshua Pereyda. Finding a Fuzzer: Peach Fuzzer vs. Sulley. Retrieved from `https://medium.com/@jtpereyda/finding-a-fuzzer-peach-fuzzer-vs-sulley-1fcd6baebfd4`, 2016. Accessed: 2017-12-05.

[36] Pedram Amini and Aaron Portnoy. Fuzzing Sucks - Introducing Sulley Fuzzing Framework. *Black Hat US*, 2007. Accessed: 2017-12-13.

[37] Pedram Amini and Aaron Portnoy. Sulley: Fuzzing Framework. Accessed: 2017-12-05.

[38] CW Dukes. Committee on national security systems (CNSS) glossary. Technical report, Technical report CNSSI, 2015.

[39] Joshua Pereyda. boofuzz Documentation. Retrieved from https://media.readthedocs.org/pdf/boofuzz/latest/boofuzz.pdf, 2017. Accessed: 2018-01-09.

[40] Kristen K Greene, John Kelsey, and Joshua M Franklin. NISTIR 8040 - Measuring the Usability and Security of Permuted Passwords on Mobile Platforms. Retrieved from https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8040.pdf, 2016. Accessed: 2018-02-23.

[41] Svetlana Z Lowry. NISTIR 7741 - NIST Guide to the Processes Approach for Improving the Usability of Electronic Health Records. Retrieved from https://www.nist.gov/sites/default/files/documents/itl/hit/Guide_Final_Publication_Version.pdf, 2010. Accessed: 2018-02-23.

[42] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[43] Wenliang Du. *Computer Security: A Hands-on Approach*. CreateSpace Independent Publishing Platform, 2017.

[44] Wayne Jansen. NISTIR 7564 - Directions in Security Metrics Research. Retrieved from https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7564.pdf, 2009. Accessed: 2018-02-23.

[45] Patrick Hagelkruys. Analyse von CA-Software-Systemen mit anschließender Untersuchung ausgewählter Probleme beim Design einer CA-Software. 2015.

[46] Sylvain Marquis, Thomas R. Dean, and Scott Knight. SCL: A Language for Security Testing of Network Applications. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 155–164. IBM Press, 2005.

[47] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[48] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision on full SHA-1. *IACR Cryptology ePrint Archive*, 2015:967, 2015.

[49] Emanuele Acri. In-memory-fuzzing in Linux (with GDB and Python). Retrieved from https://crossbowerbt.github.io/in_memory_fuzzing.html, 2012. Accessed: 2018-02-21.

[50] Ollie Whitehouse. Introduction to Anti-Fuzzing: A Defence in Depth Aid. Retrieved from https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/

`introduction-to-anti-fuzzing-a-defence-in-depth-aid/`, 2014. Accessed: 2017-12-12.