Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Enhancement of a Tool for Comprehensive Security Scanning

Fabian Hauck

Course of Study:	Informatics
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Dr. Daniel Fett (yes.com AG), Dipl.-Inf. Guido Schmitz
Commenced:	October 23, 2019
Completed:	May 27, 2020

Abstract

The demand for web applications is rapidly increasing worldwide. Since the world wide web is accessible to everyone with a connection to the internet, web-based systems are especially vulnerable to attacks. This is why cybersecurity is getting increased attention. While it is difficult to defend a system from sophisticated attacks it is rather easy to find and fix insecure system configurations. Since web applications and their infrastructure are rapidly changing, it is hard to manually detect security breaches. Therefore advanced testing software is needed to detect security leaks automatically.

The present work describes several extensions of an automated security scanning tool called *yesses*. The yesses tool was originally designed to scan web servers for basic security properties like open ports, insecure HTTP methods and missing cookie security features. The tool is accessible open-source on GitHub [Fet20b]. Within the scope of this work, the yesses tool was extended by seven modules. Hereby the following three main topics were investigated: Transportation Layer Security (TLS), Domain Name System Security Extensions (DNSSEC) and information leakages. Within the TLS topic, TLS scans of the TLS settings of a server are performed and the differences compared to a Mozilla TLS profile were analyzed. Among other things this gives important insights into possible insecure encryption algorithms. In the scope of DNSSEC, the DNSSEC configuration of a domain name was scanned. Hereby the tool can detect possible misconfigurations, e.g. a missing signature for a DNS resource record. Concerning information leakages, the yesses tool was extended in such a way, that it detects sensitive data exposures which are very useful for potential adversaries. The described extensions do not only make the yesses tool more powerful, they also enable it to detect security leaks that could not have been detected beforehand.

Acknowledgement

First of all I would like to thank Prof. Ralf Küsters for the assignment of a bachelor thesis topic in the field of web security. I have been interested in this topic for a long time. Thanks also to my supervisor Guido Schmitz at the University of Stuttgart for his constructive feedback.

I am especially grateful to my supervisor Daniel Fett at the yes.com AG for always answering my questions and for the interesting discussions. Furthermore I want to thank my family for supporting me during the time of writing this thesis. Last but not least I want to show gratitude to Anke Schult for correcting the language and grammar.

Contents

List of Figures

List of Tables

1 Introduction

Today the internet is more popular than ever before, resulting from the fact that roughly sixty per cent of the world's population is connected with the internet [Cle20] and that each user on average spends more than three hours on the internet every day [Cle19]. As everybody with an internet connection has access to web applications these services are especially vulnerable to a lot of attacks.

The Open Web Application Security Project (OWASP) investigates attack types on web applications every few years. The OWASP Foundation is a non-profit, open community obligated to provide unbiased, practical information about application security. From all attacks found on the web, the OWASP Foundation publishes the ten most frequently found attacks in the OWASP Top 10. These ten attack types are a good starting point to check a web application for vulnerabilities [WWv+17].

While it is certainly difficult to protect web applications from sophisticated attacks, it is rather easy to check the infrastructure for basic security properties and misconfigurations. The significance of security misconfigurations becomes obvious as it is sixth on the OWASP Top 10 list. Since web applications and their infrastructures are rapidly changing, it is hard to manually detect security vulnerabilities. This leads to the necessity for advanced software solutions that can automatically detect security breaches.

This thesis describes several enhancements for an automated security scanning tool called *yesses*. The yesses tool is written in Python and is freely available as open-source software on GitHub. The scanning tool was originally designed to scan web servers for basic security properties like open ports, unsafe HTTP headers as well as insecure HTTP methods. A detailed description of the software can be found in Chapter 2. In the context of this thesis, seven modules with important new functionality were developed for the yesses tool. These modules cover the security-relevant topics: Transportation Layer Security (TLS), Domain Name System Security Extensions (DNSSEC) as well as information leakages. Furthermore, an automated testing environment was added to the yesses tool.

The first topic examined in this thesis is Transportation Layer Security (TLS). TLS is used to secure different protocols on the internet such as HTTP, VPNs, and emails. TLS has a long history with a lot of different versions and encryption algorithms, but not all of them are still secure today. Consequently, it is particularly important to select secure TLS settings for web servers. For this reason, the Mozilla Foundation has published TLS profiles to provide a guideline for state-of-the-art TLS settings by only enabling secure encryption algorithms. Within this topic, the already existing `TLS Settings` module of yesses was extended to do a comprehensive comparison between the TLS settings of a server and a Mozilla TLS profile. This topic is discussed in Chapter 3 as well as Appendix A.

The second topic in this thesis covers the Domain Name System Security Extensions (DNSSEC). The Domain Name System (DNS) is used to map domain names to IP addresses. Since the Domain Name System does not provide data integrity DNSSEC was developed. DNSSEC adds digital signatures and public keys to the Domain Name System to verify the correctness of DNS data like IP addresses, for example. As a result the setup of DNSSEC for a domain name is quite complex. To help finding misconfigurations a newly developed module called `Dnssec` was added to the yesses tool. The `Dnssec` module is a powerful tool to inspect the DNSSEC configuration of a domain name and it warns of misconfigurations like missing public keys or invalid signatures. More about DNSSEC and the `Dnssec` module can be found in Chapter 4 and Appendix B.

The third part of this thesis deals with information leakages in web applications and is described in Chapter 5. Information leakages are application weaknesses that appear whenever a system reveals sensitive data, for example, implementation details or user-specific data. In general, these data are of high potential use for possible adversaries. Consequently it is important for the security of the application and its users to find and fix information leakages. To detect information leakages five new modules were added to the yesses tool.

In addition to the new modules, the yesses tool was also extended by an automated test environment. The test environment uses Docker containers with different web servers which are particularly close to a real server infrastructure. This allows extensive testing of the newly added modules. The test environment is discussed in Chapter 6.

The yesses tool uses several third-party Python packages to scan for security properties. Considering that those packages did not provide all the features needed for the new modules, several code contributions were made to other open-source software. Additionally, multiple severe bugs were found in open-source software while working on this thesis. These bugs were reported via GitHub issues. More information about the contributions to the open-source community can be found in Appendix C.

2 The Yesses Project

In order to guarantee the security of web applications it is crucial to scan them regularly for basic security properties. This is especially important if there are new deployments every day. Existing security compliance testing tools either scan for special vulnerabilities like *Cross-Site Scripting* and *SQL injection* or they need an agent installed on the web server itself. Examples of such scanners are SQL Map [DS20] for SQL injection and Burp Suite [Por20] for XSS vulnerabilities, SQL injections, and other injection attacks. Another example using an agent to do a comprehensive security scan is OpenVAS which performs a scan of several network protocols as well as web applications. It can also be used to check local security properties [Bun].

But so far - without using an agent - there is no security scanner that can scan for security properties on the network level, the TLS configuration, and the web application level. The focus hereby is not on XSS vulnerabilities or SQL injection but more on HTTP header configurations and insecure HTTP methods like TRACE. For these reasons, Daniel Fett developed the yesses tool [Fet20b] for the yes.com AG. This tool is a lightweight scanner for basic security properties that creates alerts if a vulnerability is found. An overview of how yesses works can be seen in Figure 2.1. The yesses tool is available open-source under the GNU Affero General Public License v3.0. This chapter introduces the yesses project and its modules that already existed before this thesis was started.

To use the yesses tool it is necessary to define a baseline for the tool's findings and if the result differs from the baseline an alert is reported. The baseline configuration is done with YAML files in a way that even people who do not know the details of the software can understand which properties the tool is scanning for. In Section 2.2 there is an in-depth explanation of how the configuration works. With this setup, for example, the software can be scheduled as a weekly CRON job.

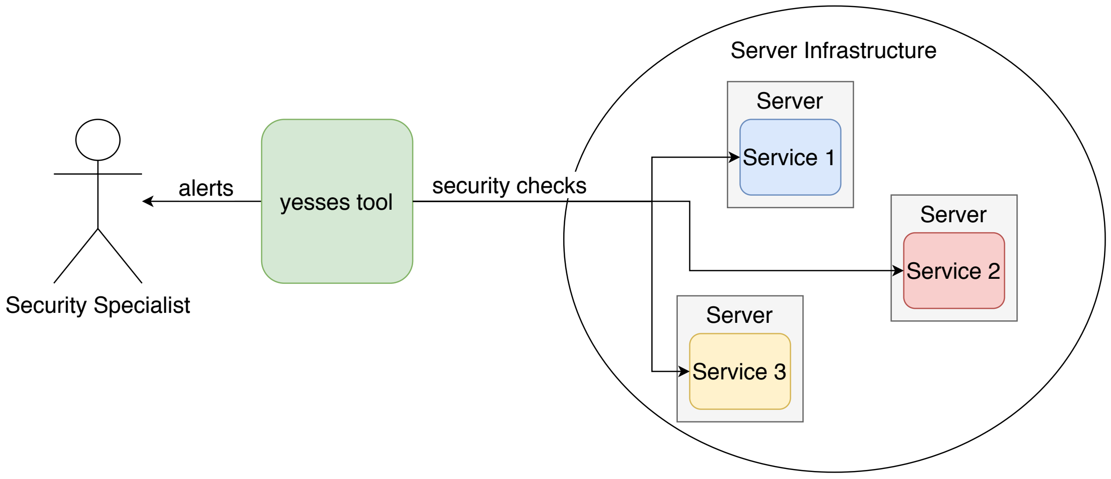


Figure 2.1: Overview of how yesses works.

2.1 Modules

To be easily extensible the software uses modules to scan for security properties. The core of the software only parses the YAML configuration, passes data from one module to another, and schedules the execution of the modules. In the following, there is a description of the modules that were already implemented. Up to this point there are scan and discovery modules. Discovery modules, for example, are checking the server certificate or investigate if there are any web servers on an IP address range. The scan modules, on the other hand, search for open ports or compare the TLS settings of a web server with a Mozilla TLS profile [Fet20b].

2.1.1 Discover TLS Certificates

One of the problems with the public key infrastructure on the web is that it relies on the trustworthiness of the certificate authorities (CA). However, history has shown that CAs do not always issue correct certificates. For example, between 2009 and 2015 Symantec mis-issued 164 certificates for 76 different domains. It was discovered by Google because one of the certificates was issued for 'google.com'. For this reason, Certificate Transparency Logs (CTLs) were introduced. The CTLs are lists that contain domain names and the associated certificates. Everybody can enter new certificates and verify a given certificate. Certificates issued after October 2017 are only accepted by Chrome if they are listed in a CTL [Küs19]. But these logs do not prevent malicious CAs from issuing wrong certificates. Consequently the domain name owner has to query these logs and check whether there are certificates not requested by him for his domain. This is exactly what the `TLS Certificates` module does. It takes domain names as inputs and outputs all associated certificates. In case that there is a new certificate, the yesses tool can be configured to create an alert [Fet20b].

2.1.2 Discover Domains And IPs

This module retrieves IP addresses from a DNS resolver based on given domain names. The domain names can either be specified inside the configuration or they can be passed on from the `TLS Certificate's` module which outputs the domain names listed in the found certificates. The discovered IP addresses can then be compared to a whitelist of IP addresses. Additionally, the module is expanding CNAMEs and is testing the existence of homoglyph domains provided by a list. Homoglyph domains are domains that look similar to the original ones but they are spelled slightly different, for example, *example.com* and *3xample.com*. The module also tries to guess expansions for wildcard domains by testing known subdomain names as wildcards. The idea is that subdomains, for example, used for testing environments could also be used in production. The module's purpose is to detect DNS hijacking and creating an alert if a new IP address or domain name has been found [Fet20b].

2.1.3 Discover Webservers

In an every-changing server configuration, it can always happen that a web server is misconfigured or is running on a wrong IP-Port combination. To detect such problems this module combines a list of IP addresses and ports with a list of domain names. Then it scans every combination to find a running web server. Running web servers will be reported in the output of the module [Fet20b].

2.1.4 Scan Ports

This module uses the *Nmap* [Nmab] tool to scan IP addresses for open ports. Open ports will be reported to the output. This is a useful feature because open ports are always the first thing an adversary would try to exploit [Fet20b].

2.1.5 Scan TLS Settings

TLS is a protocol with a long history and a lot of different versions and ciphers. Not all of them are secure anymore. That is the reason why it is important to use a strong TLS configuration. Mozilla provides three TLS profiles for a different security need and client support. This module takes domain names and the TLS profile name as an input and checks whether the server's TLS configuration matches the Mozilla TLS profile. It should prevent the TLS settings from being misconfigured, which is part of the sixth OWASP Top 10 security risk (Security Misconfiguration). This module already existed but it did only check the ciphers and protocol versions supported by the server. As part of this thesis, a comprehensive check for all properties listed in the Mozilla TLS profiles was implemented into this module. More about the Mozilla TLS profiles can be found in Chapter 3. This module supersedes the `TLS Settings Qualys` module which uses *Qualys SSL Labs* to assess the security of a server's TLS configuration [Fet20b].

2.1.6 Scan Web Security Settings

This module scans the HTTP protocol for insecure configurations. On the one hand, it checks for insecure HTTP methods like TRACE/TRACK and CONNECT. The TRACE/TRACK method can be used to start *Cross Site Tracing* attacks because it echos back what the client sent to the server. Whereas the CONNECT method can be exploited to use the server as a proxy [OWA20d]. On the other hand, the `Web Security Settings` module checks missing redirections to HTTPS and unwanted redirections from HTTPS to HTTP. Furthermore, the `Web Security Settings` module looks at HTTP headers with the result to report an alert if the 'Access-Control-.*' header is present and if headers like 'Strict-Transport-Security' or 'X-Frame-Options' have wrong values or are completely missing. The header and HTTP method check is not limited to the default values because it can be configured through the YAML configuration file. The 'Access-Control-.*' header reports by default an alert because a wrong configuration could allow third party websites with JavaScript to read sensitive information on this website. This would fall into the third OWASP Top 10 vulnerability (Sensitive Data Exposure). Insecure cookies are the last thing this module warns of. Cookies are only considered secure if the 'HttpOnly' attribute is set. Additionally, on HTTPS URIs the name must start with '__Host-' or '__Secure-' and the 'secure' and 'SameSite' attribute must be set [Fet20b]. These properties are a part of the sixth OWASP Top 10 security risks (Security Misconfiguration).

2.2 YAML Configuration

YAML is a data serialization language that provides a human-readable format. Thus, the format is especially suitable to create configuration files for computer programs. Figure 2.2 shows an example of a YAML configuration file for the yesses tool. All yesses configuration files consist of three parts. The first one is called 'data' and contains all necessary information to scan like domain names and DNS resolver IP addresses. In this part it is also possible to define a baseline for the findings.

The second part is called 'run' and specifies the modules and their execution sequence for the scan. Each module is specified by its unique module name and can have several inputs. An input can be an array from the 'data' section. The syntax for this is: use <variable name>. After this the module's outputs that should be used are specified with the keyword 'find'. These data fields are then available for other modules so that it is possible to use them as inputs. Finally, the keyword 'expect' defines what should happen with the module's findings. There are several possibilities. The first possibility is to write 'no Homoglyph-Matches, otherwise alert high', which means that, if there is a homoglyph domain in the array 'Homoglyph-Matches', the tool will create an alert with the level high. The second possibility is to configure an alert, by adding 'no new IPs, otherwise alert medium' which means if there are new items in an array the user should be warned. The tool saves its findings in every run, to compare its results with the previous ones in the following run. The last option is to create an alert if two arrays do not have the same content independent from their order. For example, if the array 'Whitelist-IPs' is defined in the 'data' section the syntax would be 'Whitelist-IPs equals IPs, otherwise alert high'. The severity level of the alert is specified at the end and can either be 'informative', 'medium', 'high' or 'very high'.

The last part of the configuration specifies how the findings should be reported. It is defined with the keyword 'output'. Currently, it is possible to write the findings into an HTML file or send them to a Slack channel. In the following, there is an overview of the most important keywords for the configuration:

1. **data:** Contains all information needed for the scan.
2. **run:** Specifies the modules and their execution order for the scan.
3. **use:** Used to input a somewhere else defined array into the module.
4. **as:** Renames a variable. Useful if there are name conflicts in the global variable namespace.
5. **find:** Defines which findings should be used from the module.
6. **expect:** Creates alerts if the result differs from the baseline.
7. **output:** Specifies how the alerts should be saved. (e.g. HTML)

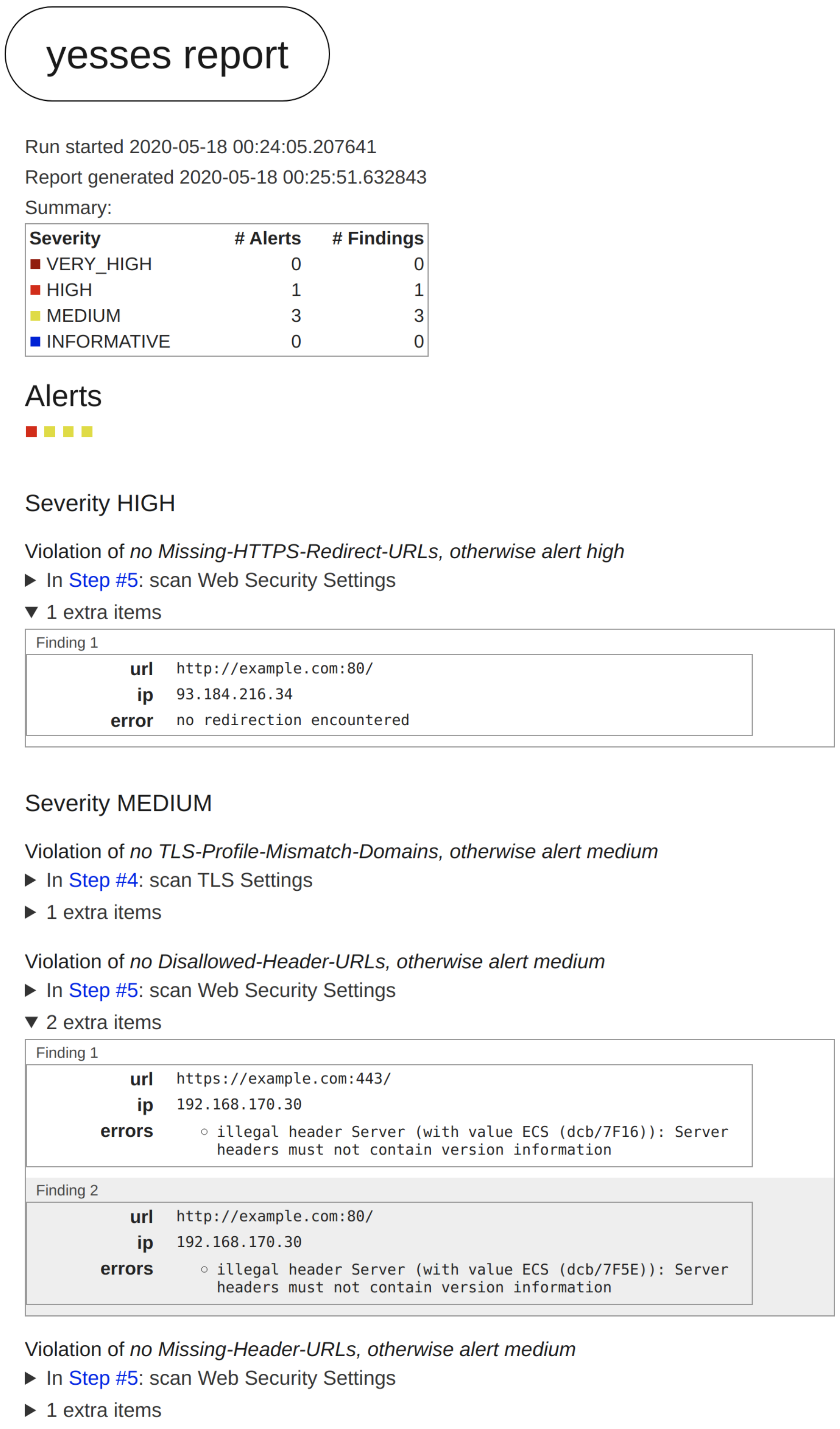
In the following, there is a high-level overview of the configuration file seen in Figure 2.2. This will also show a use-case yesses was developed for. First, the module `Domains and IPs` is used to find IP addresses for the given domain names. After this, the module is run again to find homoglyph domains and create an alert if there are any homoglyph domains. Now the found IP addresses for the domain names can be used to run a port scan with the `Ports` module to find open ports that should be closed or closed ports that should be open. This information is used to find running web servers with the `Webservers` module. The found web servers can be used again as an input for the `TLS Settings` module that matches the web servers' TLS configuration with a Mozilla TLS profile. Finally, the `Web Security Settings` module uses the `Webservers` modules outputs as inputs to check the HTTP protocol for insecurities. At the end of the run, the tool will create an HTML page with all the alerts that were created during the run. A screenshot of this HTML page is shown in Figure 2.3. As can be seen in this example the modules are designed to build upon each other.

```yaml
data:
  Domain-Seeds:
    - domain: example.com
  Homoglyph-Domains:
    - domain: exampl3.com
  DNS-Resolvers:
    - ip: 1.1.1.1
    - ip: 8.8.8.8
    - ip: 9.9.9.9

run:
  - discover Domains and IPs:
      seeds: use Domain-Seeds
      resolvers: use DNS-Resolvers
    find:
      - IPs
      - Domains
      - DNS-Entries
    expect:
      - no new IPs, otherwise alert medium
      - no new Domains, otherwise alert medium
      - no new DNS-Entries, otherwise alert high
  - discover Domains and IPs:
      seeds: use Homoglyph-Domains
      resolvers: use DNS-Resolvers
    find:
      - Domains as Homoglyph-Matches
    expect:
      - no Homoglyph-Matches, otherwise alert high
  - scan Ports:
      protocols:
        - tcp
      ports: 80,443
      ips: use IPs
    find:
      - Host-Ports
      - HTTP-Ports
      - HTTPS-Ports
    expect:
      - no new Host-Ports, otherwise alert very high
  - discover Webservers:
      ips: use HTTP-Ports and HTTPS-Ports
      domains: use Domains
    find:
      - Insecure-Origins
      - Secure-Origins
    expect:
      - no new Insecure-Origins, otherwise alert high
  - scan TLS Settings:
      domains: use Secure-Origins
      tls_profile: intermediate
    find:
      - TLS-Profile-Mismatch-Domains
      - TLS-Validation-Fail-Domains
      - TLS-Vulnerability-Domains
      - TLS-Okay-Domains
      - TLS-Other-Error-Domains
    expect:
      - no TLS-Profile-Mismatch-Domains, otherwise alert medium
      - no TLS-Validation-Fail-Domains, otherwise alert medium
      - no TLS-Vulnerability-Domains, otherwise alert medium
      - no TLS-Other-Error-Domains, otherwise alert medium
  - scan Web Security Settings:
      origins: use Secure-Origins and Insecure-Origins
    find:
      - Missing-HTTPS-Redirect-URLs
      - Redirect-to-non-HTTPS-URLs
      - Disallowed-Header-URLs
      - Missing-Header-URLs
      - Disallowed-Method-URLs
      - Insecure-Cookie-URLs
    expect:
      - no Missing-HTTPS-Redirect-URLs, otherwise alert high
      - no Redirect-to-non-HTTPS-URLs, otherwise alert high
      - no Disallowed-Header-URLs, otherwise alert medium
      - no Missing-Header-URLs, otherwise alert medium
      - no Disallowed-Method-URLs, otherwise alert medium
      - no Insecure-Cookie-URLs, otherwise alert medium

output:
  - Template:
      filename: report-example.html
      template: templates/html/main.j2
```

Figure 2.2: Example YAML configuration file for the yesses tool.



yesses report

Run started 2020-05-18 00:24:05.207641

Report generated 2020-05-18 00:25:51.632843

Summary:

Severity	# Alerts	# Findings
🟥 VERY_HIGH	0	0
🟥 HIGH	1	1
🟨 MEDIUM	3	3
🟦 INFORMATIVE	0	0

Alerts

🟥 🟨 🟨 🟨

Severity HIGH

Violation of *no Missing-HTTPS-Redirect-URLs, otherwise alert high*

▶ In Step #5: scan Web Security Settings

▼ 1 extra items

Finding 1

url	`http://example.com:80/`
ip	`93.184.216.34`
error	`no redirection encountered`

Severity MEDIUM

Violation of *no TLS-Profile-Mismatch-Domains, otherwise alert medium*

▶ In Step #4: scan TLS Settings

▶ 1 extra items

Violation of *no Disallowed-Header-URLs, otherwise alert medium*

▶ In Step #5: scan Web Security Settings

▼ 2 extra items

Finding 1

url	`https://example.com:443/`
ip	`192.168.170.30`
errors	◦ `illegal header Server (with value ECS (dcb/7F16)): Server headers must not contain version information`

Finding 2

url	`http://example.com:80/`
ip	`192.168.170.30`
errors	◦ `illegal header Server (with value ECS (dcb/7F5E)): Server headers must not contain version information`

Violation of *no Missing-Header-URLs, otherwise alert medium*

▶ In Step #5: scan Web Security Settings

▶ 1 extra items

Figure 2.3: Screenshot of the HTML page generated by yesses with the configuration file from Figure 2.2.

3 Transportation Layer Security

With the increasing popularity of the web, communicating with web applications over plain HTTP became insufficient due to the fact that everybody between client and server could read and change the transferred data. In order to guarantee confidentiality and integrity of data sent over insecure networks the Transportation Layer Security (TLS) protocol was developed to establish a secure channel between clients and servers. Besides securing HTTP, the TLS protocol is also used in many other protocols like STARTTLS for emails, SSL-VPNs and WPA-Enterprise for Wifi.

As vulnerabilities were discovered in the TLS protocol over the years, multiple versions had to be developed. At the time of writing the versions SSL 1.0, SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3 existed. The Secure Socket Layer (SSL) protocol is the predecessor of TLS. Nowadays all SSL versions have been deprecated due to severe vulnerabilities. Since not all TLS ciphers provide sufficient protection anymore it is important to have a guideline for secure TLS settings. For this reason, the Mozilla Foundation published a guideline with three different TLS profiles for different security levels and different client support.

This chapter describes the extension of the `TLS Settings` module of the yesses tool. In this work the `TLS Settings` module was extended to do a comprehensive comparison between the TLS settings of a server and a Mozilla TLS profile. Till now the `TLS Settings` module did only check the TLS versions and the supported ciphers, but for a comprehensive comparison it is also necessary to check the key exchange mechanisms and the certificate properties. These properties are exchanged during the TLS handshake.

3.1 TLS Handshake

During the TLS handshake, the TLS properties defined in the Mozilla TLS profiles are exchanged. In this phase the client and server negotiate a symmetric cipher suite to encrypt the payload sent between client and server, a shared premaster secret and other cryptographic parameters.

The premaster secret can be exchanged by using an asymmetric encryption scheme. To do this the client encrypts the premaster secret with the server's public key from the certificate. The alternative is to compute the premaster secret with the Diffie-Hellman key exchange algorithm. The Diffie-Hellman algorithm can either use finite fields or elliptic curves and can have static parameters or ephemeral parameters. The advantage of ephemeral parameters over static parameters is that it provides forward secrecy. Forward secrecy means that even if the private RSA key or Diffie-Hellman parameters are compromised an adversary will not be able to decrypt TLS sessions from the past [Kah16].

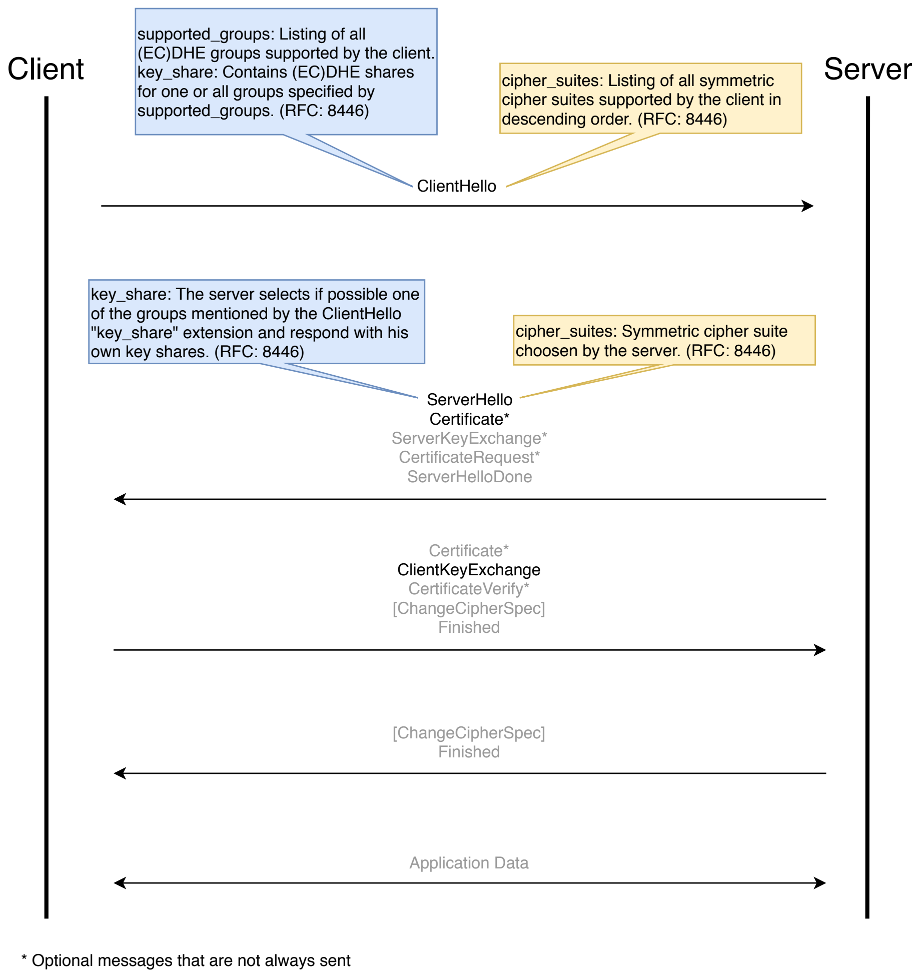

Client
Server
supported_groups: Listing of all (EC)DHE groups supported by the client. key_share: Contains (EC)DHE shares for one or all groups specified by supported_groups. (RFC: 8446)
cipher_suites: Listing of all symmetric cipher suites supported by the client in descending order. (RFC: 8446)
ClientHello
key_share: The server selects if possible one of the groups mentioned by the ClientHello "key_share" extension and respond with his own key shares. (RFC: 8446)
cipher_suites: Symmetric cipher suite choosen by the server. (RFC: 8446)
ServerHello
Certificate*
ServerKeyExchange*
CertificateRequest*
ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished
[ChangeCipherSpec]
Finished
Application Data
* Optional messages that are not always sent

Figure 3.1: Messages exchange during a TLS handshake [RD08].

Figure 3.1 shows the exchanged messages during a TLS handshake. The ClientHello message contains several extensions. The most important ones are the ’supported_groups’ and ’key_share’ extension. The former specifies supported elliptic curves and finite fields for the Diffie-Hellman key exchange; the latter contains (EC)DHE parameters for one or more groups specified by the ’supported_groups’ extension.

Furthermore, the client sends its supported cipher suites in descending order within the ’cipher_suites’ extension. In TLS 1.2 and older the cipher suites define the symmetric encryption scheme and the key exchange mechanism. The symmetric encryption scheme can either be a stream cipher like RC4_128 or CHACHA20_POLY1305 or a block cipher like AES_128_CBC or 3DES_EDE_CBC. CBC, at the end of the cipher string, describes the mode in which the block cipher is used. This can also be the GCM or the CCM mode. Today’s most popular and in TLS 1.3 the only supported symmetric encryption schemes are CHACHA20 and AES. AES can use keys with the size of 128-, 192- and 256-bits but in TLS only 128- and 256-bit keys are allowed. CHACHA20, however, always uses 256-bit keys in TLS. Should future cryptoanalysis reveal vulnerabilities in AES, CHACHA20 provides a secure alternative. With modern hardware support, AES encryption can be done very fast. Nevertheless, CHACHA20 has the advantage that its software-only implementation is about three times as fast as the one of AES making it more suitable for devices without special hardware support [NL15].

To be able to use the symmetric encryption the client and the server have to share a key. This key is calculated from the premaster secret. The problem with the key exchange is that an adversary can still take over the connection by performing a Man-In-The-Middle attack. An adversary with these capabilities could be, for example, a malicious network provider or someone who hacked a router between the client and the server. To prevent this type of attacks, TLS can use a Public Key Infrastructure with x509 v3 certificates to authenticate the server. A certificate confirms that a public key really belongs to the server the client wants to talk to. To confirm the validity of a certificate it must be digitally signed by a trusted Certificate Authority (CA) which checks that the public key belongs to the server before signing the certificate. The advantage of this is that a client does not have to save every certificate from every server it wants to talk to. Instead, it only has to save a few trusted Certificate Authority certificates. If the server sends its certificate during the TLS handshake the client can verify whether it was signed by a trusted CA or not. A successful validation guarantees that the client is talking to the real server and not a Man-In-The-Middle attacker [CSF+08].

If RSA is used for the key exchange the client can simply use the public key from the certificate knowing the key exchange is authenticated. If Diffie-Hellman is used the server signs the parameters to authenticate them and the client can verify the signature with the server's public key. Consequently, with an RSA certificate, the server can do both key exchange mechanisms. The alternative to RSA certificates is DSA or elliptic curve DSA (ECDSA) certificates. ECDSA certificates have the advantage that they use smaller key sizes than RSA or DSA certificates. DSA and ECDSA certificates can only sign messages and therefore have to be used with the Diffie-Hellman key exchange mechanism [Por13].

A complete cipher suite string for TLS 1.2 can, for example, look like this 'TLS_DHE_RSA_WITH-_AES_256_CBC_SHA256'. It means that it uses ephemeral finite field Diffie-Hellman with an RSA certificate to authenticate the key exchange. For the symmetric encryption it uses AES with a key length of 256 bits in CBC mode. The hash function SHA256 at the end is used as a message authentication code (MAC) [RD08]. The Table 3.1 shows a few TLS 1.2 cipher suites.

TLS 1.3 works quite similarly but has a few differences in the key exchange and cipher suite options. Only ephemeral finite field or elliptic curve Diffie-Hellman is allowed for key exchange and DSA certificates were removed. As symmetric ciphers, only AES and CHACHA20 are permitted. The cipher string also looks a bit different because it simply specifies the symmetric cipher suite and the MAC. An example would be 'TLS_AES_256_GCM_SHA384' [Res18].

For the sake of completeness, there is also the possibility of using pre-shared keys for a TLS connection so that the client and server do not have to do public key operations. But this is mostly used in pre-configured closed environments and because of this, it is not relevant for web servers on the internet [ETB+05].

Cipher Suite	**Key Exchange**	**Cipher**	**Mac**
TLS_RSA _WITH_NULL _MD5	Key exchange and server authentication with the public RSA key	No encryption	MD5 hash for message authentication
TLS_RSA _WITH_RC4_128 _SHA	Key exchange and server authentication with the public RSA key	RC4 stream cipher with 128 bit keys	SHA hash for message authentication
TLS_DHE_RSA _WITH_AES_128_CBC _SHA256	Key exchange with ephemeral Diffie-Hellman parameters and server authentication with a RSA certificate	AES block cipher with 128 bit in CBC mode	SHA256 hash for message authentication
TLS_DH_anon _WITH_3DES_EDE_CBC _SHA	Key exchange with static Diffie-Hellman parameters and neither of the parties is authenticated	3DES EDE block cipher in CBC mode	SHA hash for message authentication
TLS_ECDHE_ECDSA _WITH_AES_256_CBC _SHA	Key exchange with ephemeral elliptic curve Diffie-Hellman parameters and server authentication with an ECDSA certificate	AES block cipher with 256 bit keys in CBC mode	SHA hash for message authentication

Table 3.1: Example TLS 1.2 cipher suites [RD08] [BBG+06].

3.2 Mozilla TLS Profiles

As seen in the previous section, TLS supports a lot of different cryptographic algorithms. Some of them are not secure anymore or are not widely adopted. For example, there exists a ciphertext-only attack on RC4 that simply needs many encrypted versions of the target plaintext [ABP+13]. Besides, the cipher suite 3DES and Ed25519 certificates only have limited support from clients today. Additionally, TLS can even be used without encryption if the configuration is wrong.

This presents server maintainers with the challenge of knowing which cipher suites, key exchange mechanisms, and certificates offer good security and are also widely supported by clients. For this reason, the Mozilla Foundation provides three different TLS profiles. They are called 'Modern', 'Intermediate' and 'Old'. These profiles are maintained by the Mozilla Operations Security and Enterprise Information Security teams. However, they have a GitHub repository [Moz20c] for the wiki site so that anyone can suggest changes to the profiles. In the Tables 3.3 and 3.5 a detailed description of the TLS profiles is seen. To simplify the TLS setup for each profile, Mozilla provides an 'SSL Configuration Generator' [Moz20d] that automatically creates configuration files for common web servers [Moz20a].

3.2.1 Modern Profile

The Modern profile has the highest security level by just supporting TLS 1.3, but it is only compatible with up-to-date clients. For example, Android 10 is needed to open a TLS 1.3 connection which has not been supported widely yet.

All cipher suites provide very good security by being forward secret and authenticated. Because of this, the client is allowed to choose the cipher suite according to his support for hardware-accelerated AES encryption. If the client does not have hardware support it can choose the CHACHA20 cipher which has a better software-only performance as mentioned earlier.

This profile only allows ECDSA certificates using the elliptic curve 'prime256v1' or 'secp384r1' [Moz20b]. Mozilla recommends the 'prime256v1' curve because it has better performance due to a smaller key size and the curve 'secp384r1' provides only a negligible security improvement. 'Ed25519' certificates are not allowed in this profile because they are not supported widely enough [Moz20a].

3.2.2 Intermediate Profile

The Intermediate profile is a mix of supporting a wide range of clients as well as providing a good security level. It should be used for most services because it is supported by a vast majority of clients released in the last five years at least. Only very old clients like Windows XP and old OpenSSL versions do not support this setting.

The Intermediate profile permits the TLS protocols 1.2 and 1.3. Moreover, every cipher suite provides forward secrecy and is authenticated. To achieve the forward secrecy only ECDHE and DHE are provided as key exchange mechanisms and none of the cipher suites support the encryption of the premaster secret with the server's public key. Considering that, it is safe to let the client choose the cipher suite according to its hardware acceleration.

ECDSA, as well as RSA certificates, are allowed to be used. However, ECDSA certificates are preferred because of the support of the ECDHE key exchange with Windows 7 using Internet Explorer 11 and IE11 on Windows Server 2008 R2. If ECDSA certificates are not available, administrators can use the cipher suite 'TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA' to support IE11 clients on Windows Server 2008 R2. Mozilla intentionally did not include ciphers like AIRA, Camellia, 3DES, and SEED as they have very little support by clients today [Moz20a].

3.2.3 Old Profile

The Old profile has the weakest security level but it has good support for legacy clients. It should only be used as a last resort and for endpoints that really require it.

This profile supports most ciphers that do not have obvious vulnerabilities. Consequently, forward secrecy is not guaranteed because the cipher suite 'TLS_RSA_WITH_AES_128_GCM-_SHA256' [Moz20b], for example, is allowed which uses the server's public RSA key to encrypt the premaster secret during the TLS handshake. For this reason, the order of the cipher suites in the profile is very important. This implies that the server has to choose the cipher for its connection. The protocol versions TLS 1.0, 1.1, 1.2, and 1.3 are activated and SSL 3 is completely disabled. This means that clients with Windows XP SP2 are no longer supported [Moz20a].

	Modern	**Intermediate**	**Old**
Protocols	TLS 1.3	TLS 1.2, TLS 1.3	TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3
Certificate type	ECDSA (P-256)	ECDSA (P-256) (recommended) or RSA (2048 bits)	RSA (2048 bits)
Certificate signature	ecdsa-with-SHA256, ecdsa-with-SHA384, ecdsa-with-SHA512	sha256WithRSAEncryption, ecdsa-with-SHA256, ecdsa-with-SHA384, ecdsa-with-SHA512	sha256WithRSAEncryption
OCSP stapling	True	True	True
TLS curves	X25519, prime256v1, secp384r1	X25519, prime256v1, secp384r1	X25519, prime256v1, secp384r1
HSTS	max-age=63072000 (two years)	max-age=63072000 (two years)	max-age=63072000 (two years)
DH parameter size	-	2048 (ffdhe2048, RFC 7919)	1024 (generated with openssl dhparam 1024)
Maximum certificate lifespan	90 days	90 days (recommended) to 2 years	90 days (recommended) to 2 years
Cipher preference	client chooses	client chooses	server chooses
Cipher suites TLS 1.3	TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256	TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256	TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256

Table 3.3: Overview of the Mozilla TLS profiles (Part 1) [Moz20a] [Moz20b].

	Modern	**Intermediate**	**Old**
Cipher suites TLS 1.2 and older	-	ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-RSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-RSA-AES256-GCM-SHA384, ECDHE-ECDSA-CHACHA20-POLY1305, ECDHE-RSA-CHACHA20-POLY1305, DHE-RSA-AES128-GCM-SHA256, DHE-RSA-AES256-GCM-SHA384	ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-RSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-RSA-AES256-GCM-SHA384, ECDHE-ECDSA-CHACHA20-POLY1305, ECDHE-RSA-CHACHA20-POLY1305, DHE-RSA-AES128-GCM-SHA256, DHE-RSA-AES256-GCM-SHA384, DHE-RSA-CHACHA20-POLY1305, ECDHE-ECDSA-AES128-SHA256, ECDHE-RSA-AES128-SHA256, ECDHE-ECDSA-AES128-SHA, ECDHE-RSA-AES128-SHA, ECDHE-ECDSA-AES256-SHA384, ECDHE-RSA-AES256-SHA384 ECDHE-ECDSA-AES256-SHA, ECDHE-RSA-AES256-SHA, DHE-RSA-AES128-SHA256, DHE-RSA-AES256-SHA256, AES128-GCM-SHA256, AES256-GCM-SHA384, AES128-SHA256, AES256-SHA256, AES128-SHA, AES256-SHA, DES-CBC3-SHA

Table 3.5: Overview of the Mozilla TLS profiles (Part 2) [Moz20a].

3.3 Attacks on TLS

Due to vulnerabilities in the TLS protocol, several versions were developed as mentioned in the introduction of this chapter. In this section, two selected attacks on the TLS protocol are explained and it is discussed in which Mozilla TLS profile they are possible. These attacks show that the Mozilla TLS profiles provide a good security level because of the wise selection of cipher suites.

3.3.1 Logjam Attack

The Logjam attack is an attack on the Diffie-Hellman key exchange in the TLS protocol. Diffie-Hellman key exchange with ephemeral parameters is the most popular key exchange mechanism today because it provides forward secrecy. Logjam takes advantage of the fact that a lot of servers are using the same group and group generator for the Diffie-Hellman key exchange. To carry out this attack the adversary must be able to eavesdrop on a secure connection.

The Diffie-Hellman key exchange works as follows. The server chooses a group G and a generator g as well as a parameter $b \in \{0, \dots, |G| - 1\}$. It computes g^b and sents G, g and g^b to the client. The client chooses $a \in \{0, \dots, |G| - 1\}$, computes g^a and sents g^a to the server. Now both can compute $g^{a \cdot b}$. The security hereby lies in the fact that solving the discrete logarithm x ($y = g^x \; mod \; p$) with current known algorithms is a hard problem [DH76].

The most efficient algorithm for the discrete logarithm is the number field sieve (NFS) algorithm. This algorithm consists of four phases, but only the last one depends on y and g. The first three phases only need the prime factor p. Consequently it is possible to pre-compute the first three phases that require the most computational effort. This makes sense because many implementations are using one of a few Diffie-Hellman groups. By using the pre-computed values the paper 'Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice' [ABD+15] shows that it is possible to perform an effective Man-In-The-Middle attack on a TLS connection which uses the Diffie-Hellman key exchange with a 512-bit prime.

This attack could easily be adapted to 1024-bit primes. Therefore, an attacker needs a lot more computing power, however, for nation-state attackers the precomputation for a few 1024-bit primes can feasible be committed. As mentioned before many web servers are using the same few groups. The above mentioned paper analyzed that by precomputing ten 1024-bit groups, it is possible to decrypt the traffic of 24% of the Alexa Top Million HTTPS sites [ABD+15].

This attack allows an adversary to decrypt a connection established with a Diffie-Hellman key exchange by a server using the Old Mozilla TLS profile. This attack obviously requires the use of a standardized 1024-bit prime number. For the Intermediate Mozilla TLS profile this attack is infeasible because 2048-bit prime numbers are too big to perform the pre-computation steps. The Modern Profile is also not susceptible to this attack because it only allows the elliptic curve Diffie-Hellman key exchange.

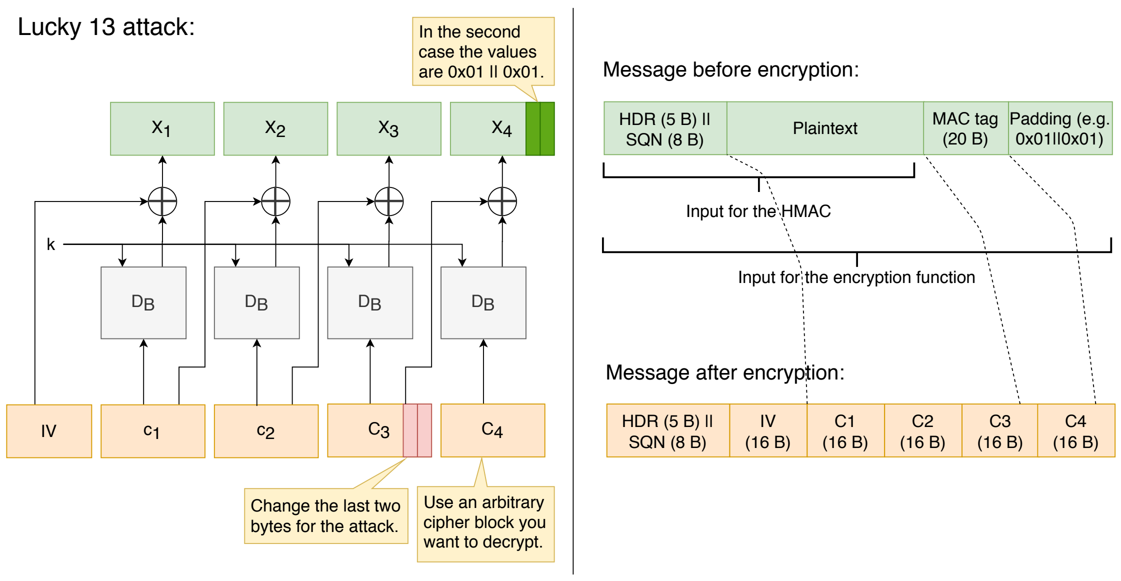

Lucky 13 attack:
In the second case the values are 0x01 II 0x01.
X1
X2
X3
X4
k
DB
IV
c1
c2
C3
C4
Change the last two bytes for the attack.
Use an arbitrary cipher block you want to decrypt.
Message before encryption:
HDR (5 B) II SQN (8 B)
Plaintext
MAC tag (20 B)
Padding (e.g. 0x01II0x01)
Input for the HMAC
Input for the encryption function
Message after encryption:
HDR (5 B) II SQN (8 B)
IV (16 B)
C1 (16 B)
C2 (16 B)
C3 (16 B)
C4 (16 B)

Figure 3.2: Left: CBC mode decryption in the Lucky 13 attack, Right: TLS CBC message [AP13].

3.3.2 Lucky 13 Attack

The Lucky 13 attack is an attack on the CBC mode (Figure 3.2) used with MAC-then-Encrypt in the TLS protocol similar to the Vaudenay's attack [Vau02]. This attack can only be carried out if an adversary is able to eavesdrop on the TLS connection and if he can inject packages into the connection.

The MAC that is used for message authentication is an HMAC that uses the $Merkle\text{-}Damg\mathring{a}rd$ construction. The number of compression function evaluations depends on the message length. For less than 55 bytes it takes 4 compression function evaluations and for 56 to 119 bytes it takes 5 compression function evaluations. This computational difference results in a timing difference that can be used by an adversary.

Before encrypting the ciphertext, it must be padded to be a multiple of the block cipher block length long. The padding consists of bytes that have the value of the padding length, for example, $0x00$, $0x01||0x01$ or $0x02||0x02||0x02$. There always has to be at least one byte of padding.

In order to describe the attack an example with a block cipher of length 16 bytes (AES) and a MAC with a tag of length 20 bytes (HMAC-SHA-1) will be used. As seen in Figure 3.2 the message for the hash function consists of the TLS header with a length of 13 bytes and the plaintext. By using 4 cipher blocks, the ciphertext has a length of $4 \cdot 16\ bytes = 64\ bytes$ without the initialization vector. To get the length of the message that will be used for the MAC function the subtraction of the 20 bytes MAC tag and the addition of the 13 bytes TLS header is needed. Calculating $64\ bytes - 20\ bytes + 13\ bytes$ leaves at most 57 bytes for the hash function. Now the last two bytes of the initialization vector for C_4 are changed and the result is sent to the server for decryption.

In the first case the padding of one byte (0x00) is assumed. That means the message length for the hash function in total is $57\ bytes - 1\ byte = 56\ bytes$. The HMAC function needs 5 compression function evaluations for this message.

In the second case the plaintext was padded with at least two bytes. This means the message for the MAC function is 55 bytes or shorter which implies that the hash function only needs 4 compression cycles.

In the third case the padding is incorrect. The RFCs for TLS 1.1 and TLS 1.2 state that here the plaintext is interpreted without padding, which results in a message length of 57 bytes for the MAC function ($64\ bytes - 20\ bytes + 13\ bytes = 57\ bytes$). This means that the hash function needs 5 compression function evaluations.

Concluding, it is possible to distinguish the second case from the two other cases by measuring the response time. By trying every combination for the last two bytes and measuring the time it takes for decryption it is possible to find a combination for the second case. If the second case is detected, the padding is most likely $0x01||0x01$. With this knowledge the output of the AES block can be recovered by computing the XOR between the found bytes and the plaintext ($0x01||0x01$). As a result the actual plaintext for the last two bytes can be calculated with the output of the AES block and the real initialization vector.

By iterating this attack, similar to the corresponding step in the Vaudenay's attack [Vau02], the whole block can be calculated and by using every block as the last block the complete message can be decrypted [AP13].

This attack even works in TLS 1.2 but not in the Intermediate Mozilla TLS profile despite the fact that it supports TLS 1.2. This is due to the fact that Mozilla only included block cipher suites that use the GCM mode in the Intermediate profile. The same goes for the Modern profile. However, the Old Mozilla TLS profile is vulnerable to this attack because, for example, the cipher suite 'AES128-SHA' is included which uses AES in CBC mode with the HMAC-SHA-1 for message authentication. This is exactly the configuration discussed in the example above.

3.4 TLS Settings Module for yesses

To compare the TLS settings of a server with a Mozilla TLS profile the yesses tool has the `TLS Settings` module. The module does not make the comparison itself. Instead, it uses the *TLS Profiler* Python package created by Daniel Fett. The TLS Profiler is available as open-source software on GitHub under the GNU Affero General Public License v3.0 [Fet20a]. The reason for outsourcing the module was to simplify maintenance and to increase reusability.

The TLS Profiler Python package uses the popular *SSLyze* [Diq19b] Python package to scan the TLS settings of a server. The SSLyze Python package uses the *nassl* [Diq19a] Python package to interact with OpenSSL. OpenSSL is used to establish the actual TLS connection. The TLS properties for each Mozilla TLS profile are read from a JSON file [Moz20b] provided by Mozilla. Consequently, if Mozilla changes the properties of its TLS profiles, the TLS Profiler will just have to download a new JSON file.

So far the TLS Profiler Python package had the following features: find illegal protocol versions, find illegal ciphers, check the certificate's validity and scan the TLS library for known vulnerabilities like Heartbleed. As stated in Table 3.3 and Table 3.5 the Mozilla TLS profiles contain a lot more properties. Until now there was no check for the key exchange mechanism and certificate properties, two essential parts of the TLS protocol.

In order to get the server's certificate the TLS Profiler uses the 'CertificateInfoPlugin' of SSLyze. With the certificate returned by SSLyze, it is possible to check the certificate type as well as the public key's elliptic curve or RSA key length. Additionally, the certificate's signature as well as the certificate's lifespan can also be examined. Not required by the Mozilla TLS profiles but a useful feature is the creation of an alert if the certificate expires soon. The default is to warn 15 days before the certificate expires but this can be altered individually by the user. The 'CertificateInfoPlugin' also returns whether OCSP stapling is supported by the server. OCSP stands for 'online certificate status protocol'. It means that a client can ask the certificate authority (CA) to confirm the validity of a server's certificate. The CA does this by sending a signed message with an expiration date [SMA+13]. To prevent the overhead of asking the CA for this message a server can send the signed OCSP response during the TLS handshake. This is known as OCSP stapling and is currently required by all Mozilla TLS profiles [Pet13].

To scan the properties used during the TLS handshake the TLS Profiler uses the 'OpenSslCipher-SuitesPlugin' of SSLyze. This plugin, however, did not provide all the information needed to draw a comparison with a Mozilla TLS profile. It returned the ciphers and protocol versions supported by the server but it did not return the cipher order and not the elliptic curve nor the finite field parameters used during the Diffie-Hellman key exchange. Fortunately, there were forks of SSLyze that returned the server's cipher suite order and the Diffie-Hellman parameters used for the key exchange. However, the cipher suite order fork had a minor bug and the Diffie-Hellman parameter fork was not compatible with the latest version of the nassl master branch. In the scope of this thesis, both bugs were fixed and a pull request was sent to the maintainer of SSLyze. Detailed information about the pull requests can be found in the Appendix C.2. With these two forks it is possible to check the finite field Diffie-Hellman parameters as well as the cipher suite order for the Old Mozilla TLS profile. With knowledge of the server's cipher suite order the program can check whether the server or the client chooses the cipher suite. In case the client chooses the cipher suite order the server does not have an order.

The last missing property of the Mozilla TLS profiles are the elliptic curves used throughout the key exchange. It is possible to test whether a server supports an elliptic curve by setting only one curve in the 'supported_groups' extension in the 'ClientHello' message. Additionally, a cipher suite that uses elliptic curve Diffie-Hellman for the key exchange must be chosen. If the handshake is successful, the server supports the elliptic curve or else it remains unsupported. Setting values to the 'supported_groups' extension requires direct access to OpenSSL's 'SSL_set1_groups_list' method. In order to make use of this method a Python binding for the 'SSL_set1_groups_list' method was integrated into nassl. The actual test of the elliptic curves was then integrated into the 'OpenSslCipherSuitesPlugin' of SSLyze. These changes were also sent in a pull request to the maintainer of SSLyze and nassl (See Appendix C.2).

Furthermore, all Mozilla TLS profiles require a certain value for the HTTP Strict Transport Security (HSTS) header. HSTS forces a browser to access a site only over HTTPS [HJB12]. A check for this value was not directly requested by the yes.com AG but for the sake of completeness, it was also integrated into the TLS Profiler Python package. The 'HttpHeadersPlugin' of SSLyze is used to query the value of the HSTS header.

This detailed information enables the TLS Profiler to draw a comprehensive comparison of the server's TLS settings with a Mozilla TLS profile. In case a TLS property differs from the Mozilla TLS profile specification the TLS Profiler will respond with a detailed error message. With the TLS Profiler, the `TLS Settings` module of yesses simply needs to call the TLS Profiler class for every domain and then write the result into the output array.

To verify the correctness of the TLS Profiler a test environment with four Docker containers using different web servers was built into the TLS Profiler Python package. One for each profile and one that does not match any profile. The certificates for the test containers are always recreated before the test starts to ensure that their lifespan is not expired yet. A similar test setup was made for the yesses tool. More information is given in Chapter 6.

Before the TLS Profiler Python package was developed available software and web services were evaluated that specifically check the TLS configuration of a server and compare them with a Mozilla TLS profile. So far no software existed that can make this comparison. Since the TLS Profiler project provides a Python package that does this task, a web interface for the TLS Profiler was being written. (cf. Appendix A)

4 Domain Name System Security Extensions

Every computer connected to the internet can be reached by a unique IP address. Consequently, the knowledge of the IP address is needed to access a specific computer. Today there are two IP address formats in use. First, there is IPv4 where addresses look like this ’93.184.216.34’ and second there is IPv6 where addresses look like this ’2606:2800:220:1:248:1893:25c8:1946’. It is really hard to remember which IP address belongs to which service, especially if the service uses IPv6.

With the intention of simplifying the access of services on the internet, the Domain Name System (DNS) was developed in the 1980s. It was first standardized in the RFC 1034 [Moc87a] and RFC 1035 [Moc87b] by the Internet Engineering Task Force in 1987. The Domain Name System maps domain names to IP addresses. Thus, it is only necessary to remember the human-readable domain name and not the IP address to access the services. For example, the domain name ’example.com.’ is mapped by the Domain Name System to the address ’93.184.216.34’ from above.

Due to the fact that the Domain Name System does not provide integrity, the Domain Name System Security Extensions (DNSSEC) were developed. Since the DNSSEC setup for a domain name can be quite complex the yesses tool needed a module to check the DNSSEC configuration of domain names. As part of this thesis, the `Dnssec` module was developed for the yesses tool that verifies the DNSSEC chain-of-trust from top to bottom.

4.1 Domain Name System

Each domain name is expressed as a sequence of labels. The labels are separated by dots and the last label is always the null label. For example, the domain name ’www.example.com.’ consists of the labels ’www’, ’example’, ’com’ as well as the null label [Moc87b].

The Domain Name System is organized in a tree structure where every node is called a zone. The child nodes of a zone are called subzones. Figure 4.1 shows a simplified DNS three. Each zone is managed by a set of authoritative nameservers, however, one nameserver can be authoritative for several zones. The subzones of the *root* zone contain the top-level domains like *com*, *de* and *eu*.

To resolve a domain name a recursive DNS resolver asks the *root* zone where to find the name. For ’www.example.com.’ the *root* zone points the resolver to the *com* zone. The *com* zone again knows where to find all domain names that end with ’com’. Consequently, the *com* zone points the resolver to the *example.com* zone. In this zone the resolver finds the answer for ’www.example.com.’.

The answer can be almost any kind of information that can be stored inside a database. One row in the DNS database is called a resource record and contains an owner name (domain name), the class (e.g. internet), the type and the information associated with these properties. The type, for example,

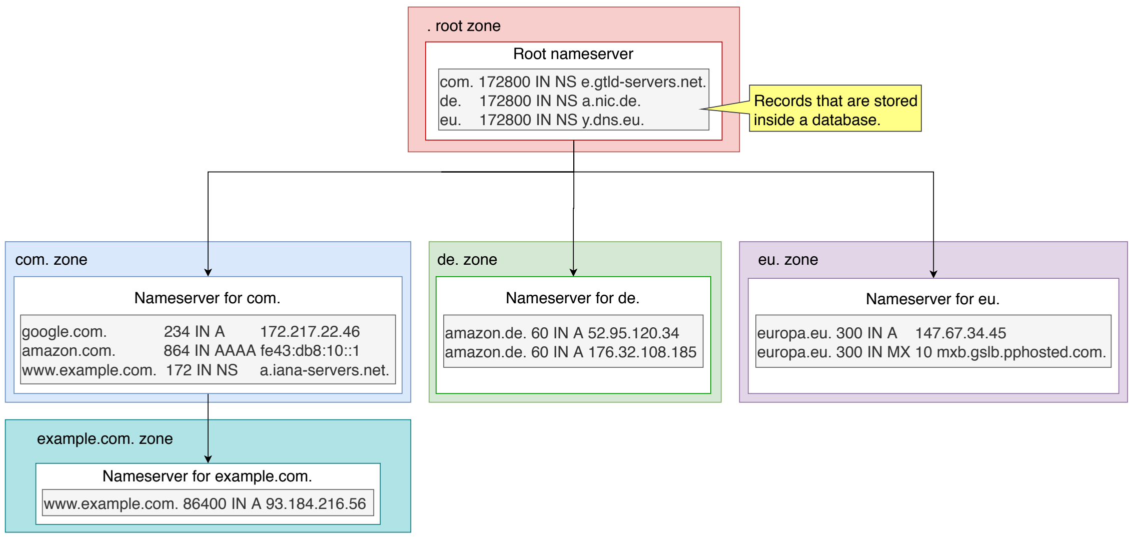

. root zone
Root nameserver
com. 172800 IN NS e.gtld-servers.net.
de. 172800 IN NS a.nic.de.
eu. 172800 IN NS y.dns.eu.
Records that are stored inside a database.
com. zone
Nameserver for com.
google.com. 234 IN A 172.217.22.46
amazon.com. 864 IN AAAA fe43:db8:10::1
www.example.com. 172 IN NS a.iana-servers.net.
de. zone
Nameserver for de.
amazon.de. 60 IN A 52.95.120.34
amazon.de. 60 IN A 176.32.108.185
eu. zone
Nameserver for eu.
europa.eu. 300 IN A 147.67.34.45
europa.eu. 300 IN MX 10 mxb.gslb.pphosted.com.
example.com. zone
Nameserver for example.com.
www.example.com. 86400 IN A 93.184.216.56

Figure 4.1: Simplified DNS tree with the *example.com* zone.

can be IPv4 or IPv6 as well as NS which contains the domain name of another zone's nameserver. Another type is 'Start of Authority' (SOA) which has to be at the beginning of a zone and contains meta-information about the zone [Moc87a].

The Domain Name System was developed with scalability and availability [Moc87a] but not integrity and confidentiality in mind. Thus it is vulnerable to Man-in-the-Middle attacks. An adversary, for example, could change the IP address in the answer to redirect the client to a malicious side controlled by the attacker. Furthermore, some states use the Domain Name System to censor the internet by redirecting clients to a website that says 'This website is blocked!' for certain sites. To prevent tampering with DNS queries the Domain Name System Security Extensions (DNSSEC) were introduced. DNSSEC satisfies two important requirements: First, by only introducing additional record types, it does not break compatibility with legacy resolvers. Second, it is possible to query domain names from zones that do not support DNSSEC. If a zone does not support DNSSEC there is obviously no integrity for queries in that zone [AAL+05a] [AA04].

4.2 Domain Name System Security Extensions

The Domain Name System Security Extensions (DNSSEC) were developed to add integrity to the Domain Name System. It does this by adding additional record types for signatures and public keys. By doing this it does not break the compatibility with devices that do not support DNSSEC because these devices can just ignore the additional records. Also, not every zone has to support DNSSEC. Hence, there has to be proof that records do not exist. Otherwise, an adversary would just remove the signatures from the DNS query to make the resolver believe that the zone does not support DNSSEC [AAL+05b].

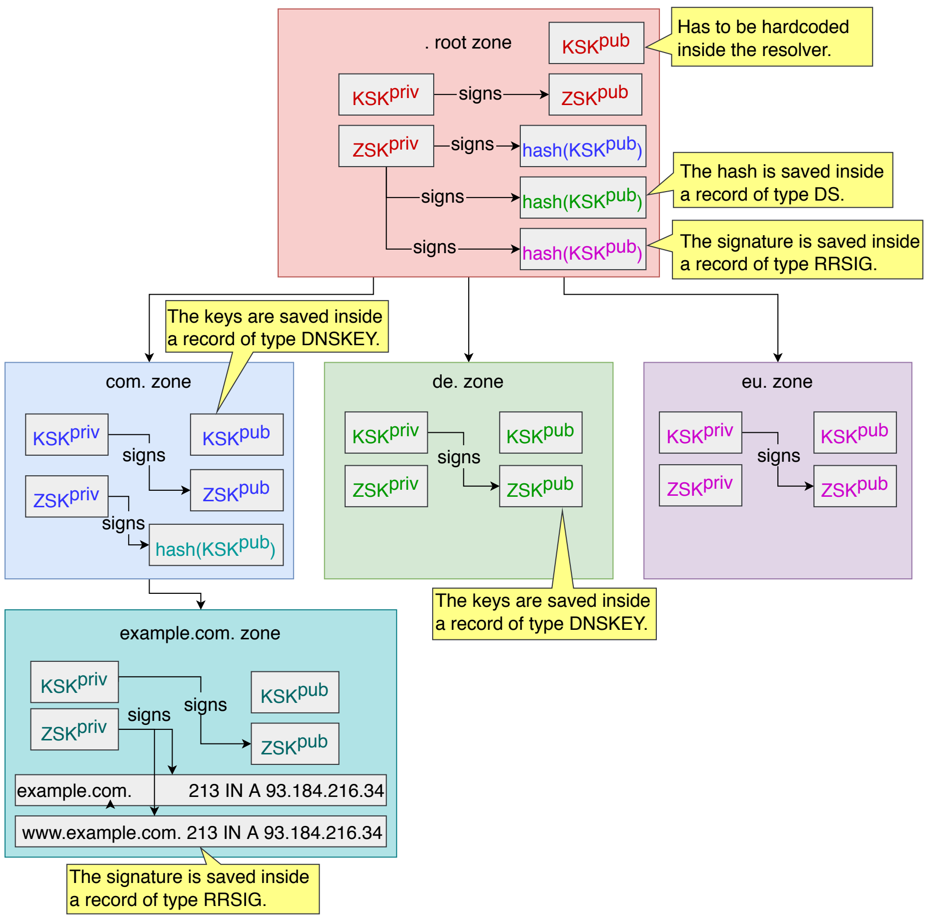

. root zone
KSKpub
Has to be hardcoded inside the resolver.
KSKpriv
signs
ZSKpub
ZSKpriv
signs
hash(KSKpub)
signs
hash(KSKpub)
The hash is saved inside a record of type DS.
signs
hash(KSKpub)
The signature is saved inside a record of type RRSIG.
The keys are saved inside a record of type DNSKEY.
com. zone
KSKpriv
KSKpub
signs
ZSKpriv
ZSKpub
signs
hash(KSKpub)
de. zone
KSKpriv
KSKpub
signs
ZSKpriv
ZSKpub
eu. zone
KSKpriv
KSKpub
signs
ZSKpriv
ZSKpub
The keys are saved inside a record of type DNSKEY.
example.com. zone
KSKpriv
KSKpub
signs
ZSKpriv
signs
ZSKpub
example.com. 213 IN A 93.184.216.34
www.example.com. 213 IN A 93.184.216.34
The signature is saved inside a record of type RRSIG.

Figure 4.2: Simplified DNS tree with DNSSEC support.

To verify a DNS answer DNSSEC uses a Public Key Infrastructure to build a chain-of-trust from the DNS *root* zone to the zone that contains the answer to the query. Each zone has a Key Signing Key Pair (KSK) and a Zone Signing Key Pair (ZSK). The KSK signs the public key of the ZSK and the ZSK signs all the resource records in the zone as well as the hashes of the KSKs from the subzones. Figure 4.2 presents the same DNS tree as Figure 4.1 but with the asymmetric key pairs for each zone. By signing the hash of the subzone's Key Signing Key a validating resolver only needs to know the KSK of the DNS *root* zone. From there the resolver can build the chain-of-trust because every zone signs the trusted KSKs of its subzones [AAL+05b] [KMM12].

The signature for each resource record is stored inside a record of type RRSIG. One RRSIG holds a signature for all resource records that have the same owner name, class and type. Furthermore, it has an expiration date [AAL+05c].

The ZSKs and the KSKs are stored inside resource records of type DNSKEY. A DNSKEY resource record has to have the flag bit 7 set to be a Zone Key. The bit 15 is the Secure Entry Point (SEP) flag. If this flag is set the key is a KSK otherwise it is a ZSK [MSP04]. Another important flag is the revoke bit that is used to prevent the use of a compromised key. The revoke bit is the flag bit 8 and if this flag is set the key has to be self-signed, or else an attacker could intentionally set the revoke bit to break the chain-of-trust [StJ07].

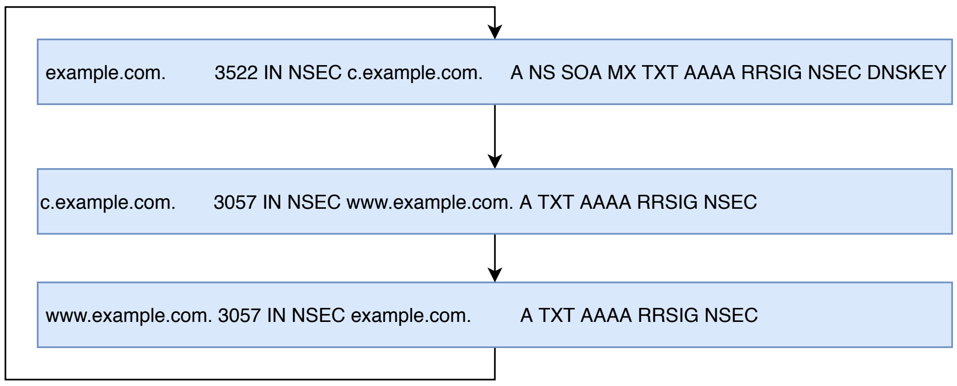

example.com. 3522 IN NSEC c.example.com. A NS SOA MX TXT AAAA RRSIG NSEC DNSKEY
c.example.com. 3057 IN NSEC www.example.com. A TXT AAAA RRSIG NSEC
www.example.com. 3057 IN NSEC example.com. A TXT AAAA RRSIG NSEC

Figure 4.3: NSEC resource records of a small example zone.

Another resource record introduced with DNSSEC is the DS record that holds a subzone's Key Signing Key hash and is signed by the zone's Zone Signing Key. The last two new resource records are the NSEC and NSEC3 resource records that are used to deny the existence of domain names or resource record types [MM14] [AAL+05c].

4.2.1 Authenticated Denial of Existence

Until now the integrity of existing resource records and types with digital signatures have been proven, but the Domain Name System can also respond with the status 'NXDOMAIN', which means that the domain name does not exist or with the status 'NODATA', which means that the requested resource record type does not exist. Since the DNS response is not authenticated the status in the response cannot be trusted by a resolver. In the following, the process of authenticated denial of existence is described as in RFC 7129 [MM14].

A simple solution would be that the nameserver signs the DNS response. But this would go against the 'first sign, then load' principle of DNSSEC. This principle was introduced because nameservers have to process a lot of queries so that it is infeasible to sign requests on the fly. In addition, the nameserver must have access to the private key so that it can create signatures. This, however, is considered to be a security risk. For these reasons, the NSEC and NSEC3 records where introduced.

Authenticated denial of existence with NSEC and NSEC3 resource records work in a similar way but with NSEC3 it is a little bit more complicated. An NSEC record contains the next domain name in the zone in canonical order as well as all record types that are available at the resource record's owner name. Figure 4.3 shows the NSEC records of a small zone that only contains the domain names 'example.com.', 'c.example.com.' and 'www.example.com.'. By specifying the next existing domain name in the zone, it is indirectly specified which domain names do not exist. Every domain name between the owner name and the next existing name in canonical order does not exist.

If domain names in canonical order are compared it works by comparing each label starting with the rightmost label. In case the rightmost labels are the same, it is necessary to continue with the next one and so forth [AAL+05c]. If there is no label left in one of the domain names the name with more labels is sorted after the one with fewer labels. In the example of Figure 4.3 the domain name ’a.example.com.’ would be between ’example.com.’ and ’c.example.com.’. This means that the first NSEC record from Figure 4.3 covers the name ’a.example.com.’. If the NSEC record can be verified with a signature the resolver can conclude that the domain name does not exist.

To show that there are actually no resource records for the domain name ’a.example.com.’ it is also necessary to show that no wildcard expansion is possible. This is why a second NSEC record is needed that covers the wildcard ’*.example.com.’. With one NSEC record covering the domain name and one NSEC record covering the wildcard, it is proven that the status ’NXDOMAIN’ is correct. Further considerations on authenticated denial of existence with NSEC records can be found in Appendix B.

In case the DNS response comes back with the status ’NODATA’ it is necessary to check, if there is an NSEC record where the owner name equals the queried name. If such a record is found the type bitmap lists all types available for the queried name. If the asked type is not listed in the type bitmap it is proven that the queried type does not exist for the queried name. As always the NSEC record must be validated with a signature. With this method it is possible to show that a subzone does not support DNSSEC. Because if there is no DS record for the subzone it will not be possible to build a chain-of-trust.

Due to the nature of NSEC, it is possible to query every domain name in the zone by following the next existing domain names in the NSEC records. It is also possible to detect the zone end because the next domain name of the last NSEC record is the first domain name in the zone. Figure 4.3 shows this principle. Besides, NSEC requires an additional record plus signature for every name in the zone. Especially in big zones, a lot more records cause extra costs which would potentially prevent DNSSEC adoption. For these reasons, the NSEC3 records were developed.

NSEC3 basically works in the same way as NSEC, but every domain name is hashed. Instead of structuring the zone according to the canonical order, the zone is ordered by the hash names. This makes zone walking complicated because an attacker would have to brute-force every single hash. Moreover, a salt and multiple hash function iterations is applied to the domain names to make brute-force attacks more complicated. To reduce the zone size, NSEC3 has an Opt-Out flag. If this flag is set there can be one or more insecure delegations between the owner name and the next domain name of the NSEC3 record. Figure 4.4 shows a zone with NSEC3 records and the Opt-Out flag. Insecure delegation means that these subzones do not support DNSSEC. Thus they need no NSEC record because they are not secured, anyway. In a big zone, especially, with a lot of insecure delegations like the *com* zone, this technique reduces the zone size.

Showing that a record type does not exist works exactly as with NSEC but showing that a domain name does not exist is a bit more complicated with NSEC3. This is caused by losing the depth of the zone by hashing the domain names. To compensate for the loss of information an additional NSEC3 record is necessary.

The first NSEC3 record is needed for the closest encloser. The *closest encloser* is the longest suffix of the queried name that exists in the zone. In the example of Figure 4.4 the closest encloser of ’a.b.example.com.’ would be ’example.com.’. The second NSEC3 record is needed to cover the *next closer name*. The next closer name is the closest encloser with one additional label prefixed to the closest encloser. In the example above, the next closer name would be ’b.example.com.’. Since there is an NSEC3 record that covers the next closer name, the queried name cannot exist, either.

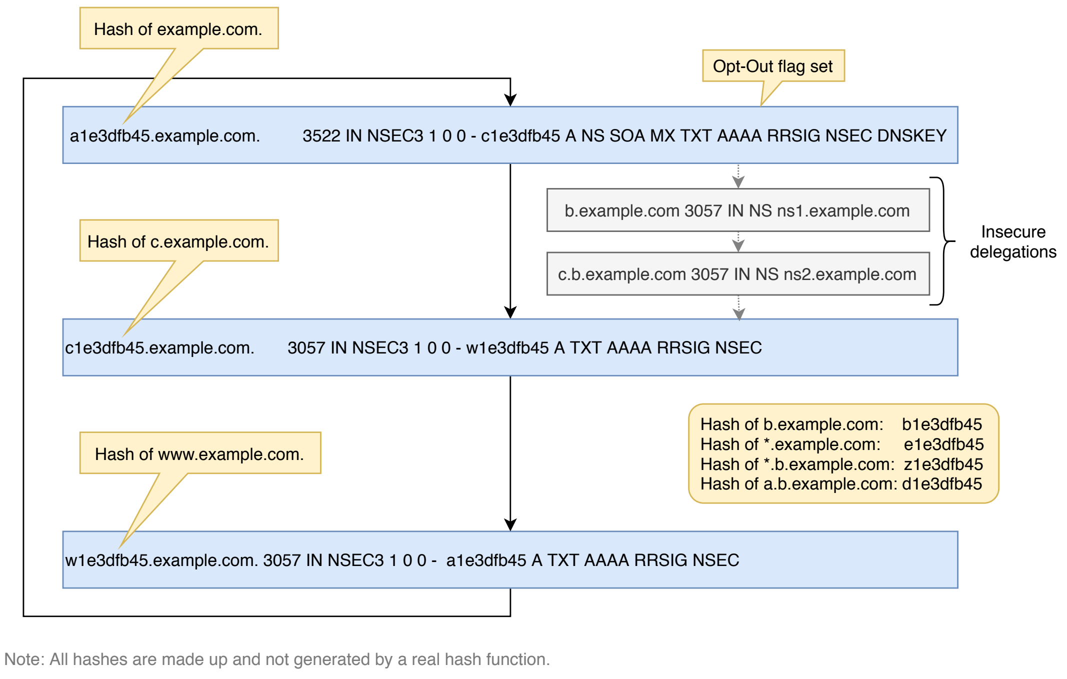

Hash of example.com.
Opt-Out flag set
a1e3dfb45.example.com. 3522 IN NSEC3 1 0 0 - c1e3dfb45 A NS SOA MX TXT AAAA RRSIG NSEC DNSKEY
b.example.com 3057 IN NS ns1.example.com
c.b.example.com 3057 IN NS ns2.example.com
Insecure delegations
Hash of c.example.com.
c1e3dfb45.example.com. 3057 IN NSEC3 1 0 0 - w1e3dfb45 A TXT AAAA RRSIG NSEC
Hash of b.example.com: b1e3dfb45
Hash of *.example.com: e1e3dfb45
Hash of *.b.example.com: z1e3dfb45
Hash of a.b.example.com: d1e3dfb45
Hash of www.example.com.
w1e3dfb45.example.com. 3057 IN NSEC3 1 0 0 - a1e3dfb45 A TXT AAAA RRSIG NSEC
Note: All hashes are made up and not generated by a real hash function.

Figure 4.4: NSEC3 resource records and the Opt-Out flag.

This would already be sufficient to show that a delegation to a subzone does not support DNSSEC, if the Opt-Out flag is set. The reason why there is no proof for wildcards needed is that wildcard records should not own NS records [Lew06].

In order to prove the non-existence of a domain name it is also important to take wildcards into account. That is why a third NSEC3 record is required. The only possible source of synthesis is '*.<closest encloser>' [Lew06]. In the example above, it would be '*.example.com.'. With this it is proven that the domain name does not exist and no wildcard expansion it possible. Like always, every NSEC3 record must be validated with a valid signature.

As mentioned before the extra NSEC3 record for the closest encloser is needed due to losing the depth of the zone. The result of this is that it is impossible to tell where a possible wildcard expansion could be. This is why the closest encloser is needed. It is the only point where a wildcard expansion could take place. But to really know what the closest encloser is, an additional NSEC3 record is needed because of the hashed names.

The following attack uses the zone as seen Figure 4.4 but the wildcard '*.example.com.' exists. Supposing that there is no NSEC3 record for the closest encloser and the query name is again 'a.b.example.com.', the answer to this query should be the expanded wildcard. Without the closest encloser an attacker could just calculate the hash for '*.b.example.com.' and search for an NSEC3 record that covers the hash. The hash of '*.b.example.com.' is 'z1e3dfb45' which is covered by the last NSEC3 record in Figure 4.4. The hash of 'a.b.example.com' is 'd1e3dfb45' which is covered by the second NSEC3 record in Figure 4.4. The adversary just adds these two NSEC3 records to the reply and removes the wildcard expansion from the reply. If the resolver now looks at the answer, it would conclude that the domain name does not exist because there is an NSEC3 record for the queried name and the wildcard. That is why a third NSEC3 record is needed [MM14].

4.3 DNSSEC Module for yesses

The DNSSEC module for yesses has to give detailed information on the DNSSEC verification process. After analyzing existing software no software was found that gives detailed error messages and offers a good integration into a Python-based software. For example, the web services 'dnssec-analyzer.verisignlabs.com' and 'dnsviz.net' deliver detailed information for the DNSSEC validation steps but are hard to integrate into yesses. The already existing Python packages are very poorly documented and do not give detailed information about the validation steps. To satisfy these requirements the `Dnssec` module was developed for the yesses tool in the scope of this thesis. Because of the size of the code base, the source code was outsourced into a separate Python package called *DNSSEC Scanner*. This simplifies the maintenance and increases the reusability. The DNSSEC Scanner can even be used directly from the command-line. The DNSSEC Scanner is available open-source on GitHub [Hau20a] under the BSD 2-Clause "Simplified" License.

The DNSSEC Scanner uses the *dnspython* [Hal18] Python package to query nameservers and to do basic DNSSEC operations. Dnspython cannot validate a DNSSEC chain-of-trust but offers primitive operations for DNSSEC. Although there were a lot of DNSSEC operations there was no function to compute an NSEC3 hash. To improve the dnspython project the NSEC3 hash function for the DNSSEC Scanner was contributed to dnspython. More about the contribution can be found in Appendix C.2.

The DNSSEC Scanner provides yesses with one of the three states 'secure', 'insecure' or 'bogus' for each domain. Secure means it was possible to verify the domain name with a trusted signature. Insecure means it was possible to show that the zone that contains the answer does not support DNSSEC. Bogus means that the DNSSEC validation failed at some point. This can, for example, mean that a signature could not be validated as well as that DNSKEY or DS records are missing. The most important feature of the DNSSEC Scanner for yesses is that it returns detailed messages for the verification process. These messages are returned either as logs, warnings or errors. Logs are all messages that document a successful verification step. A warning would be returned, for example, if one KSK could be validated and there is a DS record for another KSK, however, the hashes do not match. Warnings are basically misconfigurations that do not break the chain-of-trust. Finally, the error messages are problems in the configuration that break the chain-of-trust. In addition, the DNSSEC Scanner returns all resource records it can find. These resource records are sorted into the two categories 'secure' and 'insecure' based on whether a secure signature was found or not.

To simplify the maintenance of the DNSSEC Scanner the architecture is divided into several modules. The main class called `DNSSECScanner` uses the dnspython Python package to query all the records that are needed for the DNSSEC verification process from the nameservers. It starts at the DNS *root* zone and from there recursively asks every zone for an SOA record for the given domain name. If the SOA record was found it would try to find every resource record associated with the queried name by asking the nameserver for the type 'ANY'. In case the 'ANY' typed query does not return any records the class searches for existing records with a list of common record types. Finally, it tries to validate the found resource records. In every zone, the `DNSSECScanner` class first queries the

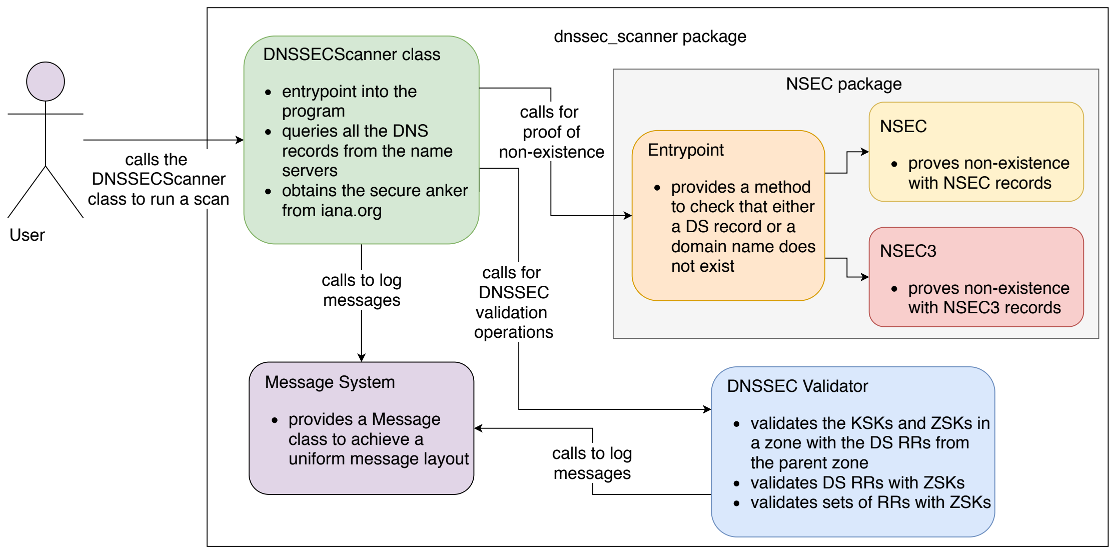

dnssec_scanner package
User
calls the DNSSECScanner class to run a scan
DNSSECScanner class
entrypoint into the program
queries all the DNS records from the name servers
obtains the secure anker from iana.org
calls for proof of non-existence
NSEC package
Entrypoint
provides a method to check that either a DS record or a domain name does not exist
NSEC
proves non-existence with NSEC records
NSEC3
proves non-existence with NSEC3 records
calls to log messages
calls for DNSSEC validation operations
Message System
provides a Message class to achieve a uniform message layout
calls to log messages
DNSSEC Validator
validates the KSKs and ZSKs in a zone with the DS RRs from the parent zone
validates DS RRs with ZSKs
validates sets of RRs with ZSKs

Figure 4.5: Overview of the DNSSEC Scanner architecture.

DNSKEY records and validates them. Then it requests the SOA record and if this cannot be found it will query the IP address of the next name server. The last step is to retrieve the DS records for the next zone from the nameserver and validate them. The DS record for the *root* zone is downloaded from 'data.iana.org'.

The second module of the DNSSEC Scanner validates signatures of resource records, ZSKs and DS records. The module does this by using the `dnssec` module of dnspython. Moreover, it also validates KSKs with DS records from the parent zone. To document the DNSSEC verification process with consistent messages there is a messaging system. It provides a `Message` class that takes all the information and generates a string. The last module is the `NSEC` module which takes NSEC or NSEC3 resource records and tries to prove the absence of a DS record or the non-existence of a domain name. The `NSEC` module is divided into an `NSEC` and an `NSEC3` module. These modules exactly follow the steps explained in Section 4.2.1. An overview of the architecture can also be seen in Figure 4.5.

To verify the correctness of the DNSSEC Scanner there are several Python unit tests with correct configured DNSSEC domains, broken DNSSEC domains, and domains in zones that do not support DNSSEC. Furthermore, there are also tests with domains that do not exist to examine the NSEC and NSEC3 verification process.

5 Information Leakages

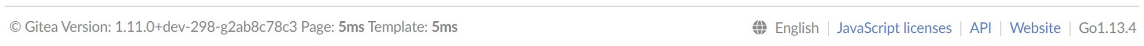
Information leakage is an application weakness that appears whenever a system reveals sensitive data such as implementation details or user-specific data. These data are of high potential use for an adversary because with this knowledge he can find vulnerabilities or specific exploits. Considering that, it is important to find and fix information leakages to improve application security [The10]. In this thesis, five new modules were developed for the yesses tool to automatically find information leakages in web applications.

5.1 Types of Information Leakages

Websites regularly display error messages if something went wrong or if users gave an incorrect input. These error messages may reveal too much information about software versions or the internal workflow of the web application. An attacker can use this information to either retrieve private data from the website or even perform a specific attack. An example of how private data could be exposed is a login form that displays different error messages if the email address is not registered or the password is wrong. With this information an attacker can get email addresses that are registered on the web service by trying random email addresses [The10].

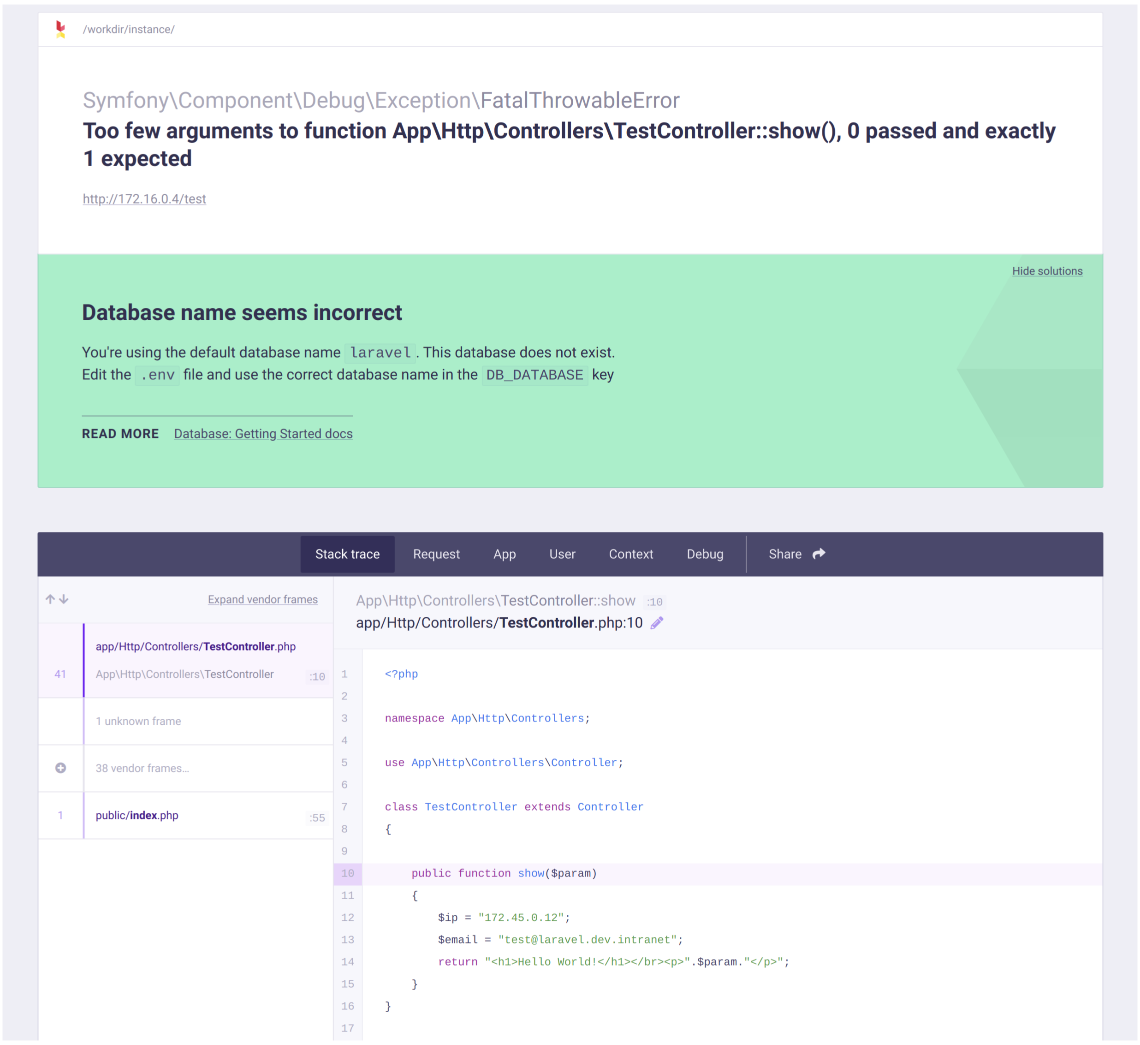
Moreover, a software version string displayed somewhere on the web page could be used for a specific attack. Figure 5.1 shows the footer of a *Gitea* [Git19b] installation. It clearly shows the version of the installation. This can be used to find specific exploits of the software [The10]. As seen in Figure 5.1 the application displays the time it needed to process the request. Timing differences are another example of information leakages. They can be used to retrace how an application works [OWA20c].

Another source for information leakages are detailed error messages like stack traces or failed SQL statements. Figure 5.2 gives an example stack trace from the Laravel framework in debugging mode. Revealing a lot of information about internal paths and PHP code is useful throughout development but it is also useful for potential attackers. Consequently, in production, these messages should be removed from the application. During deployment, HTML and Script comments containing sensitive information should also be deleted. This sensitive information can be internal IP addresses, email addresses, passwords or paths [The10].



© Gitea Version: 1.11.0+dev-298-g2ab8c78c3 Page: **5ms** Template: **5ms**   English | JavaScript licenses | API | Website | Go1.13.4

Figure 5.1: Example of information leakages on a website. It shows the footer of a Gitea installation [Git19a]. Here the exact version and duration of generating the page are seen.



/workdir/instance/

Symfony\Component\Debug\Exception\FatalThrowableError

Too few arguments to function App\Http\Controllers\TestController::show(), 0 passed and exactly 1 expected

http://172.16.0.4/test

Hide solutions

Database name seems incorrect

You're using the default database name `laravel`. This database does not exist.
Edit the `.env` file and use the correct database name in the `DB_DATABASE` key

READ MORE Database: Getting Started docs

41 app/Http/Controllers/**TestController**.php
App\Http\Controllers\TestController :10

1 unknown frame

38 vendor frames...

1 public/**index**.php :55

App\Http\Controllers\TestController::show :10
app/Http/Controllers/**TestController**.php:10

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class TestController extends Controller
{

    public function show($param)
    {
        $ip = "172.45.0.12";
        $email = "test@laravel.dev.intranet";
        return "<h1>Hello World!</h1></br><p>".$param."</p>";
    }
}
```

Figure 5.2: Example stack trace from the Laravel framework [Lar19] which leaks a lot of information if not turned off in production.

Furthermore, unlinked pages or directories can also reveal information or knowledge on how the application works. Hidden pages can be there because they were used during development or because the link was deleted but the site itself was not. This does not mean that the pages cannot be found. An attacker, for example, could search with a list of common directories, file names or with a brute-force attack for unlinked pages and directories [OWA20a]. He could also look at the robots.txt file that should prevent web spiders, robots and crawlers from scraping certain parts of the application. Directories listed in this file can also disclose hidden pages or directories that are useful to start an attack on the website [OWA20b].

Not only can the website itself reveal sensitive information but also the properties of the HTTP protocol can leak. For example, there could be private data like usernames or access tokens as GET parameters in the URI. If the connection is not encrypted with HTTPS anybody between the client and the server can read this information. Even if the data is transmitted encrypted the GET parameters are still saved in web logs, browser cache, browser history and are sent to other websites via the 'Referer Header' [Gil20].

The 'Referer Header' is not the only HTTP Header that can leak too much data. The 'Server' header reveals the web server type, which is not necessarily a problem because this can also be found out by the behavior of the server. In the standard configuration most web servers also display the exact software version. On a Raspberry Pi, for example, the Apache web server with the standard configuration responds together with the following header: 'Server: Apache/2.4.38 (Raspbian)'. This does not only disclose the version number but also the operating system which the server is running on. Further headers which should not be sent by the web server are 'X-Powered-By' and 'X-AspNet-Version'. The first one gives information about the server's scripting language like 'X-Powered-By: PHP/7.1.22' and the second simply gives the version number of the ASP.NET framework if it is used by this server [Sin18].

In addition to the web server, services on any port can also disclose useful hints for potential adversaries. With the Nmap tool [Nmab], an attacker can find these services and sometimes also the version of these services. This information can be used to exploit known vulnerabilities [Nmaa]. The occurrence of information leaks is not uncommon, which shows the rank of 'Sensitive Data Exposure' on the 3rd place of the OWASP top 10.

5.2 Information Leakage Modules for yesses

In the context of this thesis, five new modules were developed for the yesses tool to automatically find information leakages. Since the occurrence of information leakages can be very different it is difficult to detect them automatically. That is why the focus was on the header properties and searching web pages with regular expressions for sensitive information. The former is realized by the `Header Leakage` module whereas the latter is implemented in the `Information Leakage` module.

These modules do not communicate with the actual web server. Instead, they get their data from the `Linked Paths`, `Hidden Paths` and `Error Paths` modules. Figure 5.3 describes the interaction between these modules and Figure 5.4 gives an example configuration of how to use these modules. Users have the option to use the 'origins' as inputs that are found by the `Webservers` module.

5.2.1 Information Leakage Module

This module searches for strings that reveal too much information in HTML, JavaScript and CSS comments as well as in the visible text of the web pages. The module does this by parsing the web pages with the Python package *BeautifulSoup* [Ric20] and searching in the parsed text with regular expressions. Currently, the following regular expressions are implemented:

1. **E-Mail addresses:** Searches for strings with a @ sign and a domain afterward.
2. **IP addresses:** Searches for four numbers separated by dots. If such a string is found a function will check whether every number is smaller than 256.

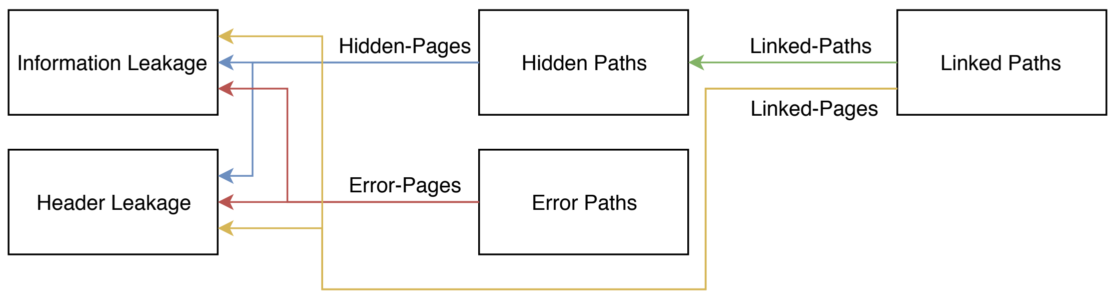


Figure 5.3: Interaction of all modules to detect information leakages.

```yaml
data:
  Origins:
    - url: http://example.com/
      ip: 93.184.216.34
      domain: example.com

run:
  - discover Linked Paths:
      origins: use Origins
    find:
      - Linked-Paths
      - Linked-Pages
  - discover Hidden Paths:
      origins: use Origins
      linked_paths: use Linked-Paths
    find:
      - Hidden-Paths
      - Hidden-Pages
  - discover Error Paths:
      origins: use Origins
    find:
      - Error-Pages
  - scan Information Leakage:
      pages: use Linked-Pages and Hidden-Pages and Error-Pages
    find:
      - Leakages
  - scan Header Leakage:
      pages: use Linked-Pages and Hidden-Pages and Error-Pages
    find:
      - Leakages as Header-Leakages
```

Figure 5.4: Example YAML configuration of the `Information Leakage` modules.

3. **Paths:** Searches for strings that are separated by slashes. If such a string is found a function will check whether it ends with a known file ending, whether the path has a depth greater than three or if at least half of the strings of the path are in a list of common directory names.
4. **Files:** Searches for a string that can start with a slash and has a dot in it. If such a string is found a function will check whether the file ending is in a list of known file endings.
5. **Server-Info:** Searches for a server information string. Such a string starts with letters followed by a slash with a version number. At the end there can be the OS name. An example would be 'Apache/2.4.38 (Raspbian)'.
6. **Version-Info:** Searches for a string that can start with a name then 'version: ' and after this, there is a version number. It can find web application version numbers like the one from Figure 5.1.

All these strings may begin at the first character of the search string or with a whitespace or an opening bracket. At the end there is either the last character of the string or a whitespace or a closing bracket. Only the IP address regular expression does not have these constraints at the beginning and at the end. Because of the numbers an IP address can be spotted in a string. If wanting to search for custom regular expressions it is possible to specify them in the YAML configuration of the module.

5.2.2 Header Leakages Module

In the information leakage introduction, it was mentioned that certain header attributes reveal sensitive information. A leakage alert will be created, if the 'Server' header attribute contains a version number or if the 'X-Powered-By' or the 'X-Aspnet-Version' header attribute exists. This module checks the headers of all the pages because each page can have a different header.

5.2.3 Linked Paths Module

As described above the `Header Leakage` module and the `Information Leakage` module do not communicate with the server. This module is responsible for scraping recursively all linked pages including JavaScript and CSS files. It only follows links where the top- and the second-level domain name is the same. Besides, there is a maximum recursion depth so that the runtime can be limited. It scrapes the links in parallel with multiple threads. The module takes one or more origins as inputs.

5.2.4 Hidden Paths Module

This module takes the Linked-Paths from the `Linked Paths` module and with its help it tries to find possible folders. Then it searches these folders with a common directory and file name list for hidden files and directories. The module takes multiple origins as inputs, but first of all it checks whether the web server responds to a random path that should not exist with a 200 status. If this is the case the origin has to be skipped because it is not possible to distinguish existing paths from not existing ones. To guarantee its termination it also has a maximum recursion depth, and in order to speed up the search it runs in multiple threads.

5.2.5 Error Paths Module

The purpose of this module is to find paths that respond with an error. Currently, it only requests a none existing path and returns the web server's error page. In future works, this module could be extended to provoke other error messages as well.

6 Test Environment

Ensuring the correctness of software is one of the most important or even the most important quality attribute of an IT system. The best method to archive software correctness is to do comprehensive testing. There are two different types of testing: manual ones done by a developer and automated unit tests. Automated testing is especially useful to prevent regression errors. A regression error is an error that appears in an already tested code caused by enhancing the software. It was required to set up an automated unit test environment for the yesses tool in order to test the modules developed in this thesis. In the following, there is a detailed description of this environment.

So far yesses has not had a testing environment but to guarantee the correctness of the modules it was important to develop a test environment with *Docker* containers. Docker containers are used because they allow the setup of an environment close to real server infrastructure. At the moment there are two Docker containers running different web servers. One runs an Nginx web server with several linked and unlinked pages and files. The other runs an Apache web server with the Laravel Framework installed. For comprehensive testing two different frameworks and web servers were used which made two Docker containers necessary. Altogether, the test environment contains three Docker containers as shown in Figure 6.1. The third runs the test suite. Managing those containers separately would be unnecessarily complex. That is why *Docker Compose* is used to start and stop the containers as well as define the network topology between the three Docker containers. The test cases themselves, are defined in YAML files that have the same syntax as the yesses configuration files. This makes it possible to easily write test cases and understand what is tested in each test case.

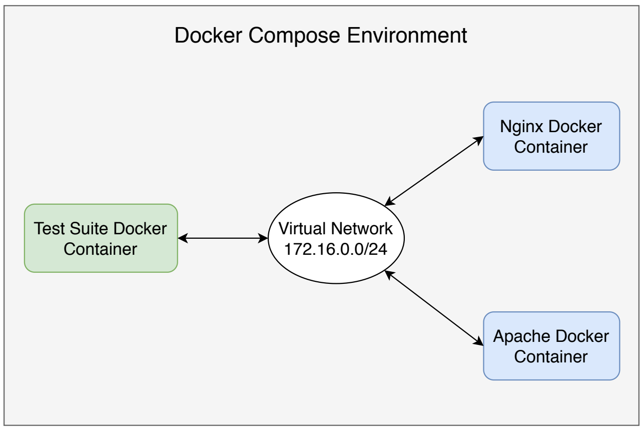

Docker Compose Environment
Nginx Docker Container
Test Suite Docker Container
Virtual Network 172.16.0.0/24
Apache Docker Container

Figure 6.1: Test environment structure.

The first test cases developed in this thesis were for the `Information Leakage` modules. For each module, there is one test case to test the module against a Docker container. Additionally, for each of the two Docker containers there is a test case that validates the workflow with all the modules together as shown in Section 5.2. As a result, the Apache container with Laravel is used to test the detection of information leakages on a web site with a real framework, whereas the Nginx container is used to test the website enumeration and finding of hidden files.

To test the `TLS Settings` module that uses the TLS Profiler, certificates were added and HTTPS was installed for both web servers. The Apache container is configured with the 'Modern' Mozilla TLS profile were as the Nginx uses the 'Intermediate' Mozilla TLS profile. In contrast to the Apache's certificate the Nginx's certificate is created with a short lifespan of only 12 days. This is used to test the 'Certificate Expiration Warning' feature. For both Docker containers there is one test case defined as a YAML file.

For the `Dnssec` module, there are no tests integrated into the yesses test environment because the DNSSEC Scanner itself has a comprehensive test environment. Moreover, it would be complicated to define tests with static results as done in the yesses test environment. The reason for this is that the DNSSEC tests examine real domain names and the DNSSEC infrastructure is rapidly changing. This would cause the test to fail every few days.

This test environment setup is especially suited to be enhanced in the future because it is easy to define new test cases in YAML files as well as to add new Docker containers which are to simulate various network services. This simplifies writing new test cases for upcoming modules.

7 Summary and Outlook

Within the scope of this thesis, an automated security scanning tool named yesses was extended with the aim of covering several new security topics. The yesses tool which is written in Python was originally designed to scan web servers for basic security properties e.g. open ports, insecure HTTP methods and missing cookie security features. The tool and its newly developed features are available open-source on Github. In this thesis, seven modules were added to the yesses tool or extended covering three major security topics, including Transportation Layer Security (TLS), Domain Name System Security Extensions (DNSSEC) and information leakages. Also, an automated test environment was added to the yesses tool to check on the correct implementation of the newly developed modules.

The first topic investigates the TLS protocol which is built upon the already existing `TLS Settings` module as part of the yesses tool. The `TLS Settings` module compares the TLS settings of a web server with the Mozilla TLS profiles. These Mozilla TLS profiles were chosen for comparison since their security algorithms provide state-of-the-art security levels. However, the original `TLS Settings` module was limited to check only a few TLS properties such as TLS versions and supported ciphers. Therefore, within the scope of this work, the module was extended to now allowing it to compare all the TLS properties of a web server based on the Mozilla TLS profiles. As a result, it states possible differences within the TLS settings, therefore, allowing it to show compliance with a Mozilla TLS profile.

The second topic covers DNSSEC and is integrated into the yesses tool as a newly developed module called `Dnssec`. Since the Domain Name System (DNS) does not provide integrity, DNSSEC was developed. With the use of DNSSEC, it is now possible to guarantee the correctness of DNS data. The task of the `Dnssec` module is to check the correctness of the DNSSEC configuration of domain names. In detail, it checks the DNSSEC chain-of-trust from the DNS root zone to the subzone containing the domain name. Considering that DNSSEC adds additional complexity to DNS this module provides extensive error messages for incorrect DNSSEC configurations. Hereby it outperforms existing DNSSEC scanners already available as open-source Python packages.

The third topic deals with information leakages of web applications in general. In this context, information leakage is understood as the exposure of sensitive data which is of high potential use for possible adversaries. Within the scope of this work five new modules dealing with information leakages were added to the yesses tool, three of these modules covering the task of either find linked web pages, hidden web pages or web pages throwing error messages. The other two modules further analyze the found pages. In detail, they investigate the pages for server software versions, internal IP addresses as well as paths to files on the web server. The result of the five newly developed modules is the output of possible information leaks. The hereby found leaked information is extremely valuable when it comes to finding and fixing the underlying security bugs within the computer system.

Additionally, in the context of this thesis, an automated test environment for the yesses tool was developed. The test environment uses Docker containers with different web servers which are especially close to real server infrastructures. This allows comprehensive testing of the newly added modules, thus it increases their software quality significantly.

The examination of the topics TLS, DNSSEC, and information leakages in this work does not only extend the functionality of the yesses tool, but also makes it more powerful in general. Furthermore, all newly implemented modules were examined within an extensive automated test environment which was also developed in this thesis.

Outlook

Despite the fact that yesses is already a powerful tool, there are further enhancements possible. For example, it would be interesting to integrate a scanner for *SQL injections* as well as *Cross-Site Scripting* attacks. A suitable scanner for SQL injections would be sqlmap [DS20] because it is also written in Python.

Furthermore, in the context of this thesis, the `Dnssec` module of yesses was developed to support most of the standardized DNSSEC mechanisms. The ones, especially, used in practice are implemented and well tested. However, the proof of the correctness of wildcard expansions has not yet been implemented. Wildcard expansions will only be allowed, if the domain name does not exist. This case must be verified with a suitable NSEC record.

Moreover, the `Information Leakage` modules were designed to work with distinct HTML pages as it was requested by the yes.com AG. A further enhancement would be a support of single-page applications. Therefore, a real browser is needed to execute the JavaScript code on the website. The Selenium [Mut18] Python package would make this possible.

Bibliography

[AA04] D. Atkins, R. Austein. *Threat Analysis of the Domain Name System (DNS)*. 2004. URL: https://tools.ietf.org/html/rfc3833 (cit. on p. 36).

[AAL+05a] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose. *DNS Security Introduction and Requirements*. 2005. URL: https://tools.ietf.org/html/rfc4033 (cit. on p. 36).

[AAL+05b] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose. *Protocol Modifications for the DNS Security Extensions*. 2005. URL: https://tools.ietf.org/html/rfc4035 (cit. on pp. 36, 37).

[AAL+05c] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose. *Resource Records for the DNS Security Extensions*. 2005. URL: https://tools.ietf.org/html/rfc4034 (cit. on pp. 37, 38, 59).

[ABD+15] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, P. Zimmermann. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *22nd ACM Conference on Computer and Communications Security*. Oct. 2015 (cit. on p. 29).

[ABP+13] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, J. C. N. Schuldt. "On the Security of RC4 in TLS". In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 305–320. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan (cit. on p. 24).

[AP13] N. J. Al Fardan, K. G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 526–540. DOI: 10.1109/SP.2013.42 (cit. on pp. 30, 31).

[BBG+06] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, B. Moeller. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. 2006. URL: https://tools.ietf.org/html/rfc4492 (cit. on p. 24).

[Boo20] Bootstrap team. *Bootstrap*. 2020. URL: https://getbootstrap.com/ (cit. on p. 58).

[Bun] Bundesamt für Sicherheit in der Informationstechnik. *Open Vulnerability Assessment System (OpenVAS)*. URL: https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/Tools/OpenVAS/OpenVAS.html (cit. on p. 13).

[Cel20] Celery. *Celery*. 2020. URL: https://github.com/celery/celery (cit. on p. 58).

[Cle19] J. Clement. *Daily time spent with the internet per capita worldwide from 2011 to 2021, by device*. 2019. URL: https://www.statista.com/statistics/319732/daily-time-spent-online-device/ (cit. on p. 11).

[Cle20] J. Clement. *Global digital population as of January 2020*. 2020. URL: https://www.statista.com/statistics/617136/digital-population-worldwide/ (cit. on p. 11).

[CSF+08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, T. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. 2008. URL: https://tools.ietf.org/html/rfc5280 (cit. on p. 23).

[DH76] W. Diffie, M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 1557-9654. DOI: 10.1109/TIT.1976.1055638 (cit. on p. 29).

[Diq19a] A. Diquet. *nassl*. 2019. URL: https://github.com/nabla-c0d3/nassl (cit. on pp. 31, 63).

[Diq19b] A. Diquet. *SSLyze*. 2019. URL: https://github.com/nabla-c0d3/sslyze (cit. on p. 31).

[Dja20] Django Software Foundation. *django*. 2020. URL: https://www.djangoproject.com/ (cit. on p. 57).

[DS20] B. Damele Assumpcao Guimaraes, M. Stampar. *sqlmap*. 2020. URL: http://sqlmap.org/ (cit. on pp. 13, 52).

[ETB+05] P. Eronen, H. Tschofenig, M. Badra, O. Cherkaoui, I. Hajjeh, A. Serhrouchni. *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*. 2005. URL: https://tools.ietf.org/html/rfc4279 (cit. on p. 23).

[Fet20a] D. Fett. *TLS Profiler*. 2020. URL: https://github.com/danielfett/tlsprofiler (cit. on p. 31).

[Fet20b] D. Fett. *yesses*. 2020. URL: https://github.com/danielfett/yesses (cit. on pp. 3, 13–16, 67).

[Gil20] R. Gilbert. *Information exposure through query strings in url*. 2020. URL: https://owasp.org/www-community/vulnerabilities/Information_exposure_through_query_strings_in_url (cit. on p. 45).

[Git19a] Gitea. *Gitea test instance*. 2019. URL: https://try.gitea.io/ (cit. on p. 43).

[Git19b] Gitea. *Gitea website*. 2019. URL: https://gitea.io/en-us/ (cit. on p. 43).

[Goo20] Google. *Material Design*. 2020. URL: https://material.io/develop/web/ (cit. on p. 58).

[Hal18] B. Halley. *dnspython Python package*. 2018. URL: http://www.dnspython.org/ (cit. on p. 41).

[Hau20a] F. Hauck. *DNSSEC Scanner*. 2020. URL: https://github.com/fabian-hk/dnssec_scanner (cit. on p. 41).

[Hau20b] F. Hauck. *Web TLS Profiler*. 2020. URL: https://github.com/fabian-hk/WebTLSProfiler (cit. on p. 57).

[HJB12] J. Hodges, C. Jackson, A. Barth. *HTTP Strict Transport Security (HSTS)*. 2012. URL: https://tools.ietf.org/html/rfc6797 (cit. on p. 33).

[Kah16] D. Kahn Gillmor. *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*. 2016. URL: https://tools.ietf.org/html/rfc7919 (cit. on p. 21).

[KMM12] O. M. Kolkman, W. (Matthijs) Mekking, R. (Miek) Gieben. *DNSSEC Operational Practices, Version 2*. 2012. URL: https://tools.ietf.org/html/rfc6781 (cit. on p. 37).

[Küs19] R. Küsters. *Lecture Grundlagen der Informationssicherheit*. 2018/2019 (cit. on p. 14).

[Lar19] Laravel. *Laravel Framework*. 2019. URL: https://laravel.com/ (cit. on p. 44).

[Lew06] E. Lewis. *The Role of Wildcards in the Domain Name System*. 2006. URL: https://tools.ietf.org/html/rfc4592 (cit. on pp. 40, 60, 61).

[MM14] R. (Miek) Gieben, W. (Matthijs) Mekking. *Authenticated Denial of Existence in the DNS*. 2014. URL: https://tools.ietf.org/html/rfc7129 (cit. on pp. 38, 41, 59–61).

[Moc87a] P. Mockapetris. *DOMAIN NAMES - CONCEPTS AND FACILITIES*. 1987. URL: https://tools.ietf.org/html/rfc1034 (cit. on pp. 35, 36).

[Moc87b] P. Mockapetris. *DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION*. 1987. URL: https://tools.ietf.org/html/rfc1035 (cit. on p. 35).

[Moz20a] Mozilla Operations Security team, Mozilla Enterprise Information Security team. *Security/Server Side TLS*. 2020. URL: https://wiki.mozilla.org/Security/Server_Side_TLS (cit. on pp. 24–28).

[Moz20b] Mozilla Operations Security team, Mozilla Enterprise Information Security team. *Security/Server Side TLS JSON*. 2020. URL: https://ssl-config.mozilla.org/guidelines/5.4.json (cit. on pp. 25, 27, 31, 63).

[Moz20c] Mozilla Operations Security team, Mozilla Enterprise Information Security team. *Server Side TLS*. 2020. URL: https://github.com/mozilla/server-side-tls (cit. on p. 24).

[Moz20d] Mozilla Operations Security team, Mozilla Enterprise Information Security team. *SSL Configuration Generator*. 2020. URL: https://ssl-config.mozilla.org/ (cit. on p. 24).

[MSP04] O. M. Kolkman, J. Schlyter, E. P. Lewis. *Domain Name System KEY (DNSKEY) Resource Record (RR) Secure Entry Point (SEP) Flag*. 2004. URL: https://tools.ietf.org/html/rfc3757 (cit. on p. 37).

[Mut18] B. Muthukadan. *Selenium with Python*. 2018. URL: https://selenium-python.readthedocs.io/ (cit. on p. 52).

[NL15] Y. Nir, A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. 2015. URL: https://tools.ietf.org/html/rfc7539 (cit. on p. 23).

[Nmaa] Nmap. *Chapter 15. Nmap Reference Guide*. URL: https://nmap.org/book/man.html (cit. on p. 45).

[Nmab] Nmap. *Nmap website*. URL: https://nmap.org/ (cit. on pp. 15, 45).

[OWA20a] OWASP. *Review Old Backup and Unreferenced Files for Sensitive Information*. 2020. URL: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/02-Configuration_and_Deployment_Management_Testing/04-Review_Old_Backup_and_Unreferenced_Files_for_Sensitive_Information.html (cit. on p. 44).

[OWA20b] OWASP. *Review Webserver Metafiles for Information Leakage*. 2020. URL: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/01-Information_Gathering/03-Review_Webserver_Metafiles_for_Information_Leakage.html (cit. on p. 44).

[OWA20c] OWASP. *Test for Process Timing*. 2020. URL: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/10-Business_Logic_Testing/04-Test_for_Process_Timing.html (cit. on p. 43).

[OWA20d] OWASP. *Test HTTP Methods*. 2020. URL: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/02-Configuration_and_Deployment_Management_Testing/06-Test_HTTP_Methods.html (cit. on p. 15).

[Pet13] Y. N. Pettersen. *The Transport Layer Security (TLS) Multiple Certificate Status Request Extension*. 2013. URL: https://tools.ietf.org/html/rfc6961 (cit. on p. 32).

[Por13] T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. 2013. URL: https://tools.ietf.org/html/rfc6979 (cit. on p. 23).

[Por20] PortSwigger Ltd. *Burp Suite*. 2020. URL: https://portswigger.net/burp (cit. on p. 13).

[RD08] E. Rescorla, T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. 2008. URL: https://tools.ietf.org/html/rfc5246 (cit. on pp. 22–24).

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. 2018. URL: https://tools.ietf.org/html/rfc8446 (cit. on p. 23).

[Ric20] L. Richardson. *Beautiful Soup*. 2020. URL: https://www.crummy.com/software/BeautifulSoup/ (cit. on p. 45).

[Sin18] S. Singh. *Essential HTTP Headers for Securing Your Web Server*. 2018. URL: https://pentest-tools.com/blog/essential-http-security-headers/ (cit. on p. 45).

[SMA+13] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. 2013. URL: https://tools.ietf.org/html/rfc6960 (cit. on p. 32).

[StJ07] M. StJohns. *Automated Updates of DNS Security (DNSSEC) Trust Anchors*. 2007. URL: https://tools.ietf.org/html/rfc5011 (cit. on p. 37).

[The10] The Web Application Security Consortium. *Information Leakage*. 2010. URL: http://projects.webappsec.org/w/page/13246936/Information%20Leakage (cit. on p. 43).

[Vau02] S. Vaudenay. "Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS..." In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by L. R. Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 534–545. ISBN: 978-3-540-46035-0 (cit. on pp. 30, 31).

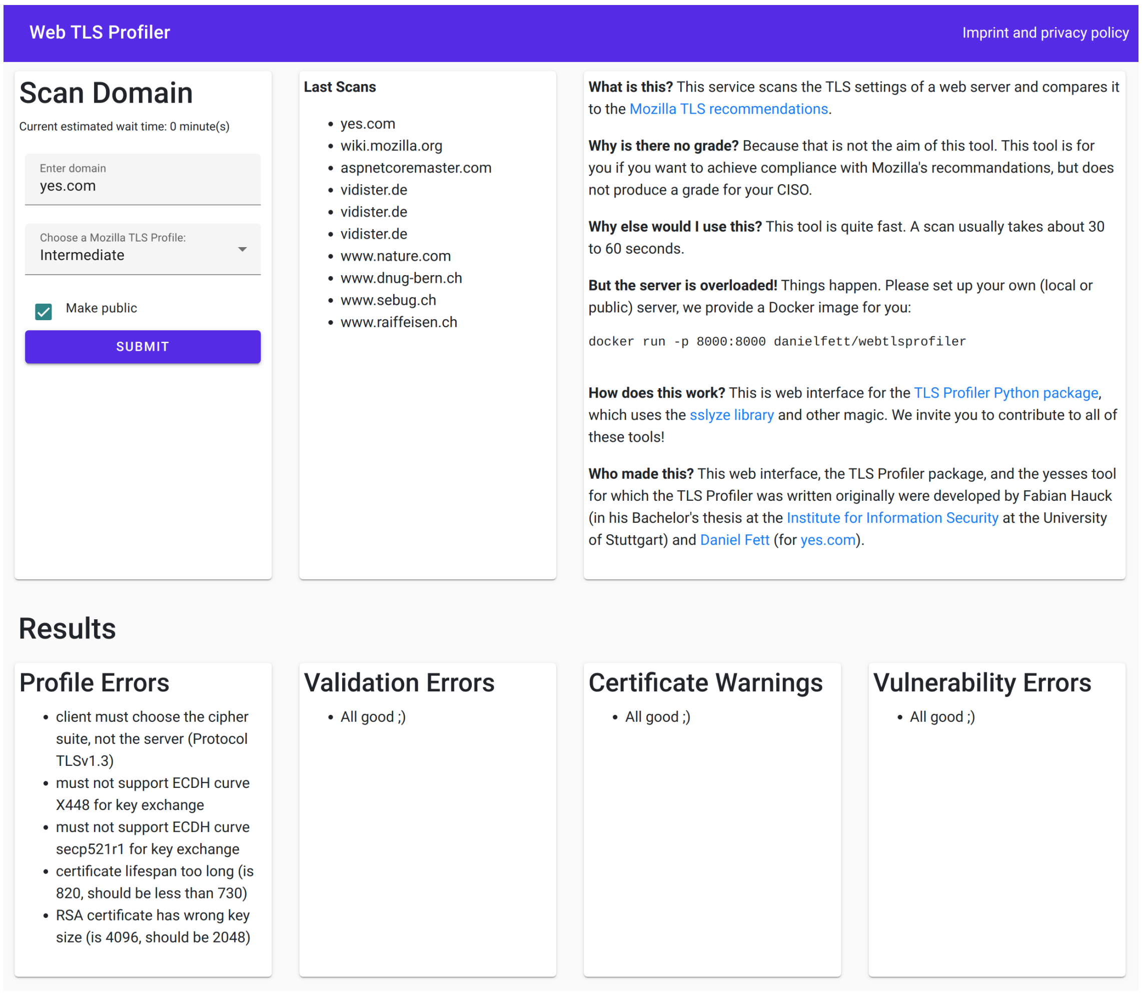
[WWv+17] D. Wichers, J. Williams, A. van der Stock, B. Glas, N. Smithline, T. Gigler. *OWASP Top 10 -2017*. 2017. URL: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf (cit. on p. 11).

All links were last followed on May 24, 2020.

A Web TLS Profiler

Web services analyzing the TLS configuration of a server are a quick and easy way to improve the security of web applications. There are already some TLS scanner web services that produce a grade for the server's TLS settings like *Qualys SSL Labs*. But so far there has been no solution to compare the TLS settings of a server with a Mozilla TLS profile. This is the reason why the *Web TLS Profiler* was developed.

The Web TLS Profiler is a web interface for the TLS Profiler Python package that was developed in the context of this thesis. The source code is available open-source on GitHub under the GNU Affero General Public License v3.0 [Hau20b]. It is also written in Python and uses the *django* [Dja20]



Current estimated wait time: 0 minute(s)

Enter domain: yes.com

Choose a Mozilla TLS Profile: Intermediate

[✓] Make public

Last Scans

- yes.com
- wiki.mozilla.org
- aspnetcoremaster.com
- vidister.de
- vidister.de
- vidister.de
- www.nature.com
- www.dnug-bern.ch
- www.sebug.ch
- www.raiffeisen.ch

What is this? This service scans the TLS settings of a web server and compares it to the Mozilla TLS recommendations.

Why is there no grade? Because that is not the aim of this tool. This tool is for you if you want to achieve compliance with Mozilla's recommandations, but does not produce a grade for your CISO.

Why else would I use this? This tool is quite fast. A scan usually takes about 30 to 60 seconds.

But the server is overloaded! Things happen. Please set up your own (local or public) server, we provide a Docker image for you:

```
docker run -p 8000:8000 danielfett/webtlsprofiler
```

How does this work? This is web interface for the TLS Profiler Python package, which uses the sslyze library and other magic. We invite you to contribute to all of these tools!

Who made this? This web interface, the TLS Profiler package, and the yesses tool for which the TLS Profiler was written originally were developed by Fabian Hauck (in his Bachelor's thesis at the Institute for Information Security at the University of Stuttgart) and Daniel Fett (for yes.com).

Profile Errors

- client must choose the cipher suite, not the server (Protocol TLSv1.3)
- must not support ECDH curve X448 for key exchange
- must not support ECDH curve secp521r1 for key exchange
- certificate lifespan too long (is 820, should be less than 730)
- RSA certificate has wrong key size (is 4096, should be 2048)

Validation Errors

- All good ;)

Certificate Warnings

- All good ;)

Vulnerability Errors

- All good ;)

Figure A.1: Screenshot of the Web TLS Profiler's desktop site.

framework to serve the website. One of the main features of the Web TLS Profiler is the scheduling of the scan tasks. This prevents the run of too many scans in parallel which would use all the server's resources. To schedule the tasks the *Celery* [Cel20] Python package is used. Celery is an asynchronous task queue that uses workers to process the tasks. This makes it possible to restrict the number of parallel scans by specifying the number of workers Celery should use. While a scan is running the frontend queries the backend with Ajax requests every two seconds whether the scan has completed. If the scan has finished the page reloads and the results are shown.

In the frontend, the Web TLS Profiler uses *Bootstrap* [Boo20] for a responsive user interface and *Material Design* [Goo20] for a modern look. Thanks to Bootstrap, the user interface also scales to mobile devices. In Figure A.1 the desktop version of the Web TLS Profiler can be seen. This web service is available under the domain name 'tlsprofiler.danielfett.de' but a Docker container is also freely available on Docker Hub.

B Reflection on NSEC Authenticated Denial of Existence

The RFCs 7129 [MM14] and 4035 [AAL+05c] describe the authenticated denial of existence with NSEC as well as NSEC3 records. While the description for NSEC3 is very detailed, the information on NSEC is a bit vague. This chapter is an in-depth reflection on the NSEC authenticated denial of existence process because it is not directly defined in the RFCs mentioned above. This will be illustrated by multiple examples. The first three examples explain what the DNSSEC Scanner does and the last example makes clear why it is important to verify the NSEC records correctly.

As mentioned before the verification process for NSEC3 is fully described in the standards. Firstly, the resolver has to find the closest encloser. Secondly, it has to verify that the next closer name does not exist. Finally, it has to prove that no wildcard expansion is possible. The verification process is clear for NSEC3, whereas it is not clearly described for NSEC. The standard only mentions that one NSEC record has to verify that the queried name does not exist and one NSEC has to verify that no wildcard expansion is possible. In order to show that no wildcard expansion is possible the resolver has to know what the closest encloser is. With NSEC3 it is easy to find the closest encloser because there is an additional NSEC3 record for it. Since there is no extra NSEC record for the closest encloser, the question of how to determine the closest encloser comes up. The closest encloser must be one of the NSECs owner names or it must be a suffix of the owner names. In any case, the queried name must be a subdomain of the closest encloser and the closest encloser must exist in the domain name space tree. An example of such a tree can be seen in Figure B.1. Every

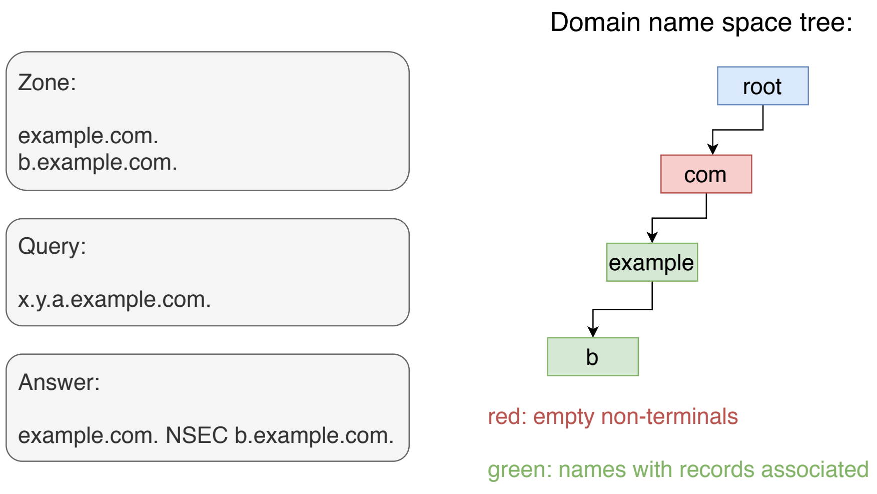
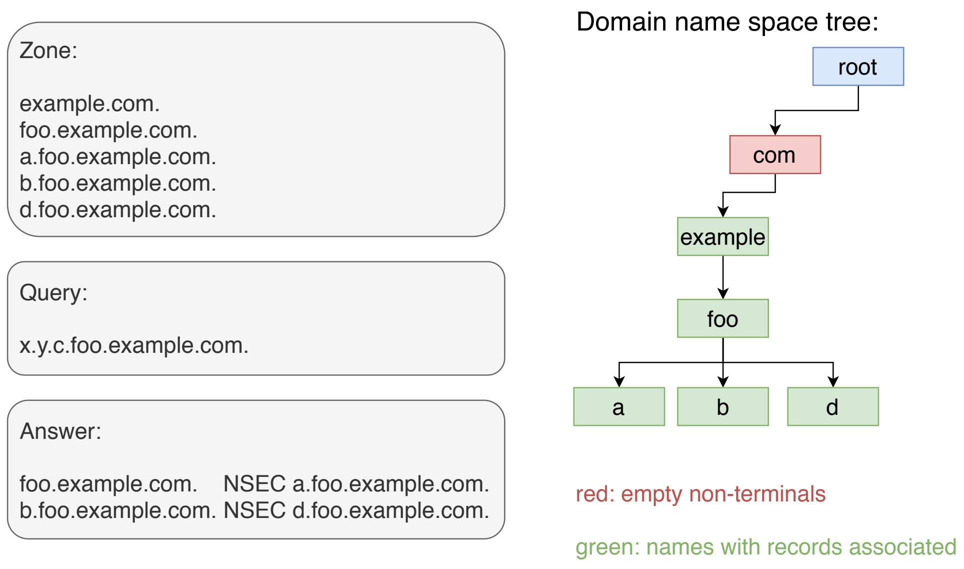


Figure B.1: Simple NSEC example.

leaf node has to have a resource record associated with the name, whereas the inner nodes do not have to have this. These empty inner nodes are called empty non-terminals but they can also be the closest encloser [Lew06]. It is important to know that empty non-terminals get an NSEC3 record but no NSEC record [MM14]. The resolver can only be sure that the determined closest encloser is correct if it can prove that every name that is closer to the queried name does not exist in the zone. This is exactly what the DNSSEC Scanner does, which was developed in the scope of this thesis. The following examples show how authenticated denial with NSEC records should work. This is not directly described in the RFC standards. The examples omit the resource record types to simplify the graphics.

The first example can be seen in Figure B.1. This is the simplest example since it requires only one NSEC record. The queried name is a subdomain of the NSEC's owner name. This means that it could be the closest encloser. The resolver has to verify that the following names that are closer to the queried name do not exist: 'a.example.com.', 'y.a.example.com' and 'x.y.a.example.com.'. Considering that the NSEC record proves that all these records including the queried name do not exist, the resolver can be sure that 'example.com.' is the closest encloser and 'x.y.a.example.com.' does definitely not exist. With the knowledge of the closest encloser, the resolver can verify that there is no wildcard '*.example.com.'. This can also be shown with the returned NSEC record. With all this information the resolver is ensured that there are no records for 'x.y.a.example.com.'.


Zone:

example.com.
foo.example.com.
a.foo.example.com.
b.foo.example.com.
d.foo.example.com.

Query:

x.y.c.foo.example.com.

Answer:

foo.example.com. NSEC a.foo.example.com.
b.foo.example.com. NSEC d.foo.example.com.

Figure B.2: NSEC example where two NSEC records are needed.

The second example in Figure B.2 needs two NSEC records to verify the non-existence of the queried domain name 'x.y.c.foo.example.com.'. In this case, the resolver has to check both NSECs owner names if they could be the closest encloser. Since the queried name is only a subdomain of the first NSEC record, the owner name of the second NSEC record cannot be the closest encloser. With 'foo.example.com.' as the closest encloser the resolver has to verify that 'c.foo.example.com.', 'y.c.foo.example.com.' and 'x.y.c.foo.example.com.' do not exist. All these domain names are covered by the second NSEC record. This means that 'foo.example.com.' is the correct closest encloser and that the domain name 'x.y.c.foo.example.com.' is unavailable. With the closest encloser, the resolver can verify with the first NSEC record that there is no wildcard '*.foo.example.com.'.

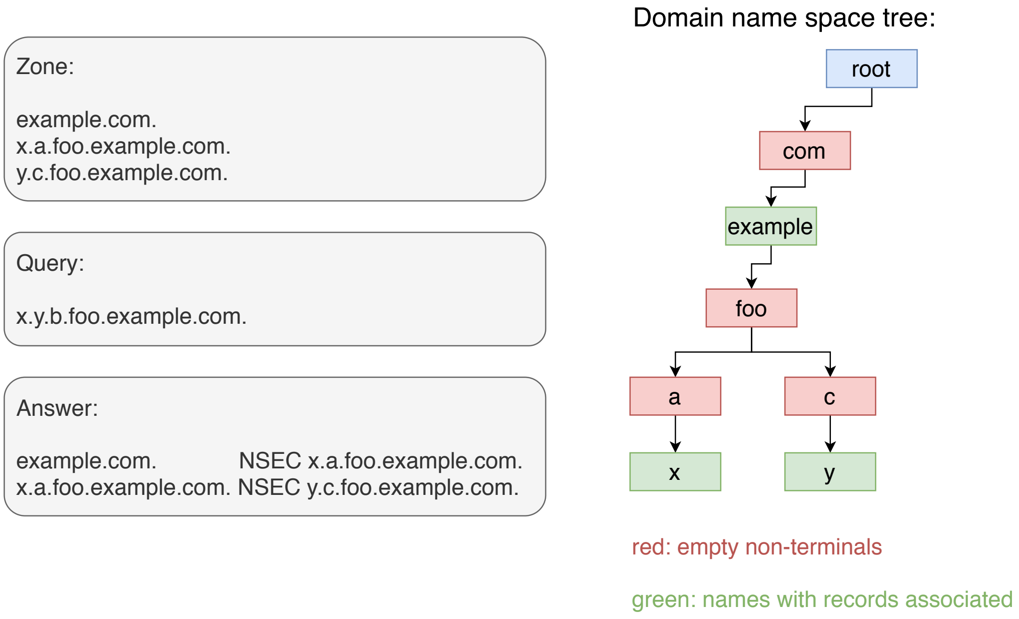
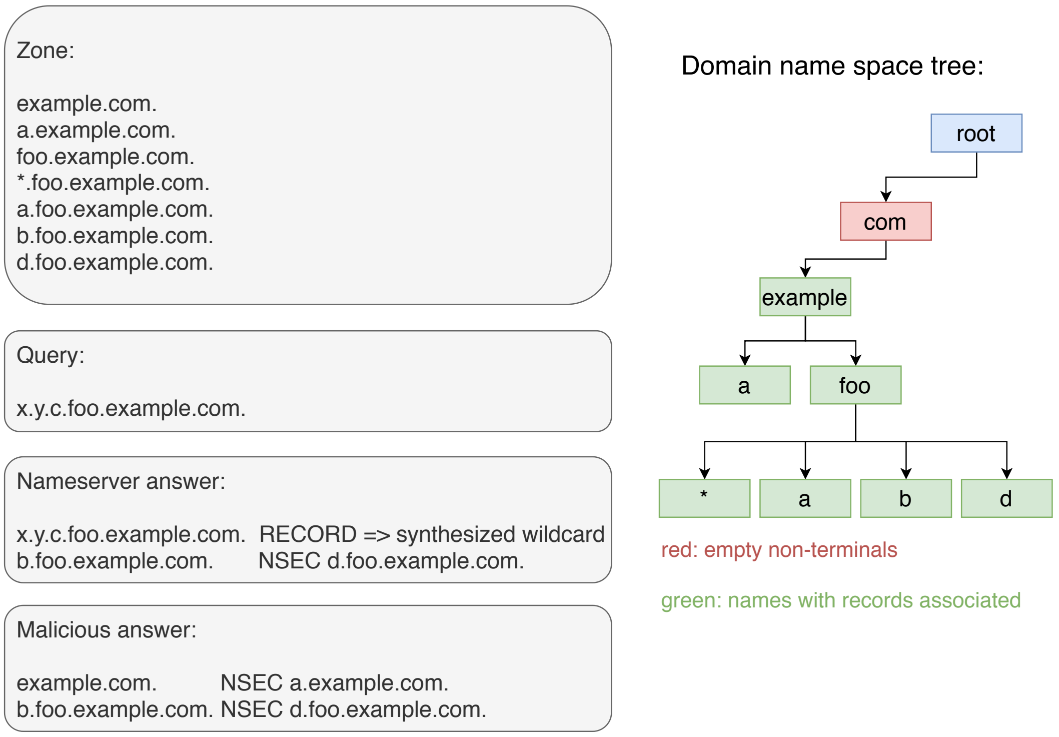


Figure B.3: NSEC example where the closest encloser is an empty non-terminal.

The third example in Figure B.3 is the most complicated one since none of the NSECs owner names is the closest encloser of the queried name. The problem in this example is that the closest encloser is an empty non-terminal and therefore it does not have an NSEC record. Consequently, the resolver has to compare the labels of the queried name with the NSECs owner names starting with the rightmost label. The longest suffix of labels that the queried name shares with one of the NSEC records owner names is the closest encloser. In this example, the closest encloser is the empty non-terminal 'foo.example.com.'. To prove the determined closest encloser the resolver has to show that 'b.foo.example.com.', 'y.b.foo.example.com.' and 'x.y.b.foo.example.com.' do not exist. All these domain names are covered by the second NSEC record. As there may be wildcards at empty non-terminals, the wildcard '*.foo.example.com.' must also be denied [Lew06]. The resolver can do this with the first NSEC record.

The last example in Figure B.4 shows what can happen if the resolver does not verify the closest encloser correctly. The example zone can synthesize the name 'x.y.c.foo.example.com.' from the wildcard '*.foo.example.com.' but the attacker replaces the answer with the NSEC record that proves that '*.example.com.' does not exist. This attack is similar to the one described in RFC 7129 [MM14] Section 5.6 for NSEC3. Assuming the resolver would not correctly verify the closest encloser, it would just verify with the second NSEC record that the queried name 'x.y.c.foo.example.com.' does not exist. After this, the resolver determines 'example.com.' as the closest encloser and verifies with the first NSEC record that the wildcard '*.example.com.' is not available. Consequently, the resolver would conclude that there are no records for the queried name. Since this conclusion is wrong, it shows why it is important to correctly verify the closest encloser. Because if the resolver checks that there is no record for 'foo.example.com.', c.foo.example.com.',



```
Zone:

example.com.
a.example.com.
foo.example.com.
*.foo.example.com.
a.foo.example.com.
b.foo.example.com.
d.foo.example.com.
```

```
Query:

x.y.c.foo.example.com.
```

```
Nameserver answer:

x.y.c.foo.example.com.  RECORD => synthesized wildcard
b.foo.example.com.      NSEC d.foo.example.com.
```

```
Malicious answer:

example.com.          NSEC a.example.com.
b.foo.example.com. NSEC d.foo.example.com.
```

Figure B.4: Example of an attack on the NSEC verification process.

'y.c.foo.example.com.' and 'x.y.c.foo.example.com.' it would detect that 'foo.example.com.' might exist. The reason for this is that 'foo.example.com.' is not covered by one of the two NSEC records. At this point, the resolver has detected that an adversary has manipulated the DNS answer.

C Contributions to Open-Source Projects

This chapter gives an overview of all the contributions that were made to the open-source community in the context of this thesis. These contributions contain GitHub issues and pull requests. In the following, there is a list with a description of each contribution.

C.1 Issues

1. nassl #56: While working on the `TLS Settings` module, a function was developed to find out which elliptic curves were supported by a server. In the process, an enum in nassl [Diq19a] was found that had a wrong string in it. It was fixed by the maintainer shortly after the issue was opened.
2. Mozilla TLS profiles #265: The JSON description of the Mozilla TLS profiles was inconsistent about naming elliptic curves for the Diffie-Hellman key exchange. The problem was that the curve 'prime256v1' was named 'prime256v1' in the 'Modern' profile and on the website but in the 'Intermediate' and 'Old' profile it was defined as 'secp256r1'. The curves are the same but the different naming made it difficult to write a clean implementation for the `TLS Settings` module. An issue was opened and the curve name got changed to'prime256v1'.
3. Mozilla TLS profiles #267: This issue stated that in the 'Intermediate' Mozilla TLS profile ECDSA certificates are allowed but the variable 'certificate_curves' was 'null'. As a consequence, an error will be raised by the `TLS Settings` module, if an ECDSA certificate is used and the server gets scanned with the 'Intermediate' profile. A maintainer of the Mozilla TLS profiler corrected this issue in the JSON file [Moz20b].
4. Mozilla TLS profiles generator #76: As mentioned in the TLS chapter Mozilla provides a configurator to generate web server configuration files that match a specific Mozilla TLS profile. While setting up the test environment for the TLS Profiler Python package it was detected that the configuration files of the Mozilla configurator do not specify the elliptic curves for the key exchange that are defined on the website. This implies that more curves are accepted by the server than specified in the Mozilla TLS profiles. Unfortunately, the maintainer did not want to include those curves because of inconsistent support from the web servers.
5. Mozilla TLS profiles #269: On the Mozilla TLS profiles website, there are recommended values for the certificate lifespan and the certificate type. To inform the user about these recommended values the `TLS Settings` module would need those values in the Mozilla TLS profiles JSON file [Moz20b]. Unfortunately, the maintainer has not include those values yet.

6. Nextcloud documentation #2038: The Nextcloud documentation recommends using a Mozilla TLS profile to harden the TLS settings of a web server. In this issue, it was proposed to link the Web TLS Profiler in the Nextcloud documentation to simplify the debugging of a server's TLS configuration. The Nextcloud documentation pull request '#2060' adds the link to the documentation page.

C.2 Pull Requests

1. SSLyze #409: This pull request was to restore the compatibility of the SSLyze 'dh_info' branch by the GitHub user FWinterborn with the nassl master branch. The 'dh_info' branch adds DH and ECDH key exchange parameters to the 'OpenSslCipherSuitesPlugin'. This changes needed the changes from the '#57' pull request for nassl. Later the maintainer of SSLyze did a big refactoring of the SSLyze code and did not need the changes from this pull request anymore.
2. SSLyze #410: So far SSLyze has no plugin to scan the elliptic curves which are supported for key exchange. As part of the work on the `TLS Settings` module, a scan for elliptic curves was integrated into the 'OpenSslCipherSuitesPlugin' of SSLyze. This pull request needs the changes from the '#58' pull request for nassl to function. At the time of writing the author of SSLyze did not review this pull request.
3. nassl #57: This pull request is only a minor change to nassl to make the pull request '#409' for SSLyze work. As stated before, after a big refactor of SSLyze the maintainer did not need those changes anymore.
4. nassl #58: In order to manually set the elliptic curves during a key exchange in OpenSSL, it is necessary to have direct access to the OpenSSL's 'SSL_set1_groups' method. This function is used by SSLyze to scan the elliptic curves which are supported by a server. To make this work the C binding for the 'SSL_set1_groups' method was added to nassl.
5. TLS Profiler #1: This pull request contains all scans and checks to compare the TLS settings of a server with a Mozilla TLS profile. Furthermore, it contains a full testing environment with Docker containers and certificates.
6. SSLyze #413: To test whether the client or the server chooses the cipher suite, SSLyze establishes two handshakes with two cipher lists, where the first cipher is different. If the first ciphers from the lists are chosen for both connections, the client will be allowed choose the cipher. While testing the TLS Profiler, it was observed that, if the cipher suite 'CHACHA20' is the first cipher in one of the lists, SSLyze will return a wrong answer. By looking further into the issue, it was noticed that SSLyze has to use the modern OpenSSL library to make a correct comparison, if the cipher suite 'CHACHA20' is involved. In this pull request, the issue was fixed and the pull request was merged by the maintainer.
7. TLS Profiler #3: To complete the TLS Profiler Python package the README file was enhanced and a command-line interface was added. Additionally, the TLS Profiler was improved by giving a certificate warning if the certificate expires soon.

8. TLS Profiler #4: After submitting an issue for the Mozilla TLS profile JSON file the file was updated. This pull request makes the TLS Profiler Python package compatible with the new JSON file. Furthermore, the test environment was improved by automatically creating the certificates on the host before running the tests.

9. dnspython #433: The DNSSEC Scanner Python package uses the dnspython Python package to query DNS records and do basic DNSSEC operations. The dnspython Python package has most of the basic DNSSEC operations like resource record validation and DS resource record creation. But so far there has been no method to calculate an NSEC3 hash. For the DNSSEC Scanner Python package an NSEC3 hash function was already implemented in this thesis. That is why the maintainers of dnspython were asked in this issue, if they were interested in integrating it into their `dnssec` module. This pull request integrates the NSEC3 hash function into dnspython. Additionally, tests for the hash function were also added to dnspython. The pull request was merged shortly after it was submitted.

10. TLS Profiler #5: This pull request contains two minor changes to the TLS Profiler Python package. First, it changes the `TLSProfilerResult` class to a Python `dataclass`. This was necessary for the *Web TLS Profiler* to easily convert the TLS Profiler results into a dictionary and then send them from the worker back to the web application. Second, a command-line interface was added to call the TLS Profiler directly from a terminal. Finally, the string representation of the `TLSProfilerResult` class was improved by printing the findings in tables.

11. Nextcloud documentation #2060: This pull request adds a link to the Web TLS Profiler to the Nextcloud documentation.

D German Abstract

Die Nachfrage nach Webanwendungen ist in den letzten Jahren weltweit stark gestiegen. Da für den Zugriff auf das World Wide Web lediglich ein Internetzugang erforderlich ist, sind Webanwendungen durch eine Vielzahl von Angriffen besonders gefährdet. Während der Schutz von Computersystemen vor ausgeklügelten Angriffen sehr schwierig ist, stellt das Finden und Beheben falscher Konfigurationen ein vergleichsweise einfaches Unterfangen dar. Da sich Webanwendungen jedoch ständig verändern, ist es sehr zeitaufwändig, die gesamte Infrastruktur regelmäßig manuell auf Sicherheitslücken zu überprüfen. Daher werden fortschrittliche Softwarelösungen zum automatischen Finden von Sicherheitslücken benötigt.

Diese Arbeit beschreibt die Erweiterung der Sicherheits-Scanning Software *yesses*. Die yesses Software wurde dafür entwickelt, eine Server Infrastruktur auf grundlegende Sicherheitseigenschaften zu überprüfen, wie zum Beispiel auf offene Ports, unsichere HTTP-Methoden und fehlende Cookie-Sicherheitsfunktionen. Die Software ist open-source verfügbar und kann von GitHub [Fet20b] heruntergeladen werden. Im Rahmen dieser Arbeit wurden sieben Module für yesses neu entwickelt oder erweitert. Dabei wurden die sicherheitsrelevanten Themen Transportation Layer Security (TLS), Domain Name System Security Extensions (DNSSEC) und Information-Leakages untersucht. Bei dem TLS-Thema ging es darum, die TLS-Einstellungen eines Servers zu ermitteln und mit einem vorgegebenen Mozilla-TLS-Profil zu vergleichen. Dadurch lassen sich unsichere Verschlüsselungsalgorithmen finden. Im Rahmen von DNSSEC wurden die DNSSEC-Konfigurationen von Domain-Namen gescannt und mögliche Fehler in diesen gemeldet. Die Software kann unter anderem fehlerhafte oder fehlende Signaturen erkennen. Für das Information-Leakage Thema wurde yesses so erweitert, dass es in einer Webanwendung die Offenlegung von sensiblen und für Angreifer sehr nützlichen Daten erkennen kann. Die hier beschriebenen Erweiterungen machen yesses nicht nur leistungsfähiger, sondern ermöglichen auch das Auffinden von Sicherheitslücken, die vorher nicht erkannt werden konnten.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature