FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Security assessment of web application penetration testing tool
Student:	Bc. Tomáš Stefan
Supervisor:	RNDr. Daniel Joščák, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Information Security
Validity:	Until the end of summer semester 2021/22

Instructions

1) Study the current state of the Burp Suite penetration testing tool or Owasp ZAP penetration testing tool (at least one of them).
2) Describe its functionality.
3) Manually examine security aspects of the application, look for weak spots. Focus on new functionalities of the applications (e.g. WebSockets).
4) Write a fuzzer application that will automatically generate input data and monitor the tested application for unexpected behaviour.
5) Discuss and analyze the results, with a focus on their security aspects.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague September 21, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Security assessment of web application penetration testing tool

Bc. Tomáš Stefan

Department of Information Security
Supervisor: RNDr. Daniel Joščák, Ph.D.

January 4, 2021

Acknowledgements

I would like to express my gratitude to my supervisor RNDr. Daniel Joščák, Ph.D., for his constructive suggestions in making of this thesis. I would also like to thank my family for their support throughout my study.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 4, 2021 …………………

Citation of this thesis

Stefan, Tomáš. *Security assessment of web application penetration testing tool*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨`https://gitlab.stdin.cz/ts/dp-security-assessment`⟩.

Abstract

The subject of the presented thesis is a security evaluation of a penetration testing tool – Burp Suite. A theoretical part first describes the application, its features, and ordinary usage. Later, we explain how a WebSocket protocol works. The practical part consists of a manual evaluation of specific components of this application, running automated scans, developing a fuzzer application to make an in-depth analysis of the WebSocket implementation, and examining network traffic, which Burp generates in the background. We identified several minor flaws such as webserver violating the HTTP standard, or an undocumented REST API call. Moreover, we managed to decipher most of Burp's network traffic and verify that it does not contain sensitive or suspicious data.

Keywords security assessment, penetration testing, Burp Suite, web, proxy

Abstrakt

Tématem předkládané práce je bezpečnostní analýza nástroje pro provádění penetračních testů – Burp Suite. V teoretické části práce je nejprve popsána samotná aplikace, její možnosti a základy běžného používání. Následuje vysvětlení fungování protokolu WebSockets. Praktická část se skládá z manuálního testování vybraných částí aplikace, automatizovaného skenování, vytvoření aplikace k provedení podrobné analýzy implementace WebSocket protokolu pomocí fuzzingu a nakonec prozkoumání síťového provozu, který Burp generuje na pozadí. Podařilo se nám najít několik drobných chyb, jako například webserver, který porušuje HTTP standard nebo nezdokumentované REST API volání. Navíc se povedlo rozklíčovat většinu síťového provozu, který Burp generuje a ověřit, že tento neobsahoval citlivá nebo podezřelá data.

Klíčová slova bezpečnostní analýza, penetrační testování, Burp Suite, web, proxy

Contents

List of Figures

Introduction

Web applications and services are probably the most popular software products nowadays, known even to the least technical person. Many people use them throughout the whole day. Search engines, video streaming services, social networks, e-shops, online banking – people can probably name at least one popular website for each of the categories mentioned. The importance of these products and the amount of money involved fortunately leads companies to realization. It is not just about the *look and feel*, but the security aspects are essential as well.

Burp Suite, a web application security testing framework, has become the de-facto standard tool for discovering such products' vulnerabilities. It combines manual testing approach with automatic scanning, making it quick in discovering the potentially vulnerable places but leaving enough space for the creativity of a security specialist.

This thesis aims to choose specific components of the penetration testing framework itself and examine the security properties, whether there might be any vulnerability present. Also, examine all suspicious parts discovered during this process. The past years have shown us that no big application is immune to security incidents. For example, the recent indicent with SolarWinds Orion should be a huge warning not to take security of any product lightly. No matter how widely used an application is, it is never harmful to be extremely cautious.

Burp Suite is a proxy between a user and a server. That gives it access to all the data transferred on the channel, including login credentials and sensitive content of the websites accessed. When conducting penetration testing inside companies, it might gain access to private networks inside. Furthermore, people using Burp are mostly security professionals expected to have valuable information on their computers. Compromise of this application would have a significant impact.

In chapter 1, we describe the Burp Suite penetration testing framework and some of the features it provides. In the following chapter 2, we take a look at WebSockets as the chosen component of Burp to be tested in this thesis. And in the last chapter 3, we proceed to the practical part – finding web endpoints, scanning, developing a WebSockets fuzzer, and analysing Burp's network communication.

Chapter 1

Web application penetration testing tools

Many tools are allowing to study the traffic going between a client and a web server. However, only a small fraction of them are useful for security researchers performing dynamic[1] penetration tests of web applications and services. [3]

In the core, such a tool is nothing else than a proxy server [4] sitting between a web browser (or other local client) and an outside world – typically Internet. However, it usually has some extended capabilities – logging of the requests and responses, intercepting the traffic, modifying it before sending further, finding vulnerable patterns, crawling locations on the server, and much more. [5, 6]

These are the two most notable products matching this description:

Burp Suite is a mature product with quite a comprehensive feature set. It is being developed by the PortSwigger Ltd. company and comes in three different editions. There is a free *Community* edition, which comes with limited features (especially in the automated scanning part, but also lacks some advanced manual tools). Then there is a *Professional* edition, which unlocks the automated scanning, Intruder, and other features. And finally, there is an *Enterprise* edition, which has completely different use case (scheduled and repeated vulnerability scans across the whole organization) and does not fit into the scope of this thesis. [7]

[1] DAST (Dynamic Application Security Testing) is performed on running application in an environment similar to the production. It simulates a real attack by hackers. On the contrary, SAST (Static Application Security Testing) is taking advantage of the knowledge of the codebase, documentation, and other sources, but does not include testing a live instance of the application. IAST (Interactive Application Security Testing) is a combination of the two approaches. [1, 2]

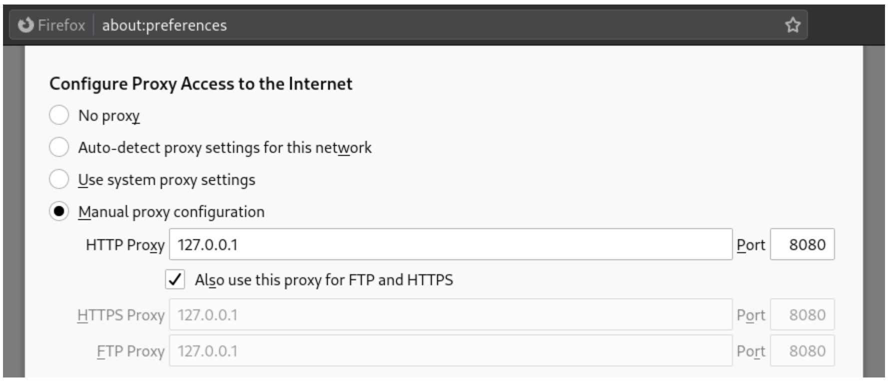

Figure 1.1: Configure proxy in Firefox's settings

Owasp ZAP is a free and open-source (Apache License 2.0) competitor to Burp suite, a solution developed by volunteers around the world. [8]

1.1 Burp Suite

The software is written in Java programming language [9], making it easily portable to various platforms, but being heavier on system resources. [10] The official installer provides versions for Windows, Linux, Mac OS, and also a plain JAR file. [11]

When we launch the application, it starts right away a proxy server, listening for new connections on the user configured address and port (by default the address is *localhost* and port *8080*). [12]

1.1.1 Usage

To see any data in the application, we need to redirect some traffic into the proxy. It can be achieved in many different ways, but let us focus on one with a great use case – web browsing.

1.1.1.1 Connecting a web browser

The first step to connect a web browser through Burp is to go to the browser's settings and configure the proxy details similarly as shown in Figure 1.1.

Any request from the web client goes through Burp now. Nevertheless, when we try to connect for example to `https://fit.cvut.cz`, we get a warning "Software is Preventing Firefox From Safely Connecting to This Site" (or similar, depending on the specific web browser). This cryptic message says, that the other side of the TLS encrypted connection is no longer the original server. Instead, it is the Burp, which uses a different certificate, not trusted by our browser. Connecting to another website on plain HTTP, e.g. `http://httpforever.com`[2], works without errors. But using the `http://` prefix for all sites is not a solution and many times even no longer possible because of the following reasons:

HSTS HTTP Strict Transport Security is a security mechanism allowing servers to declare themselves as accessible only via a secure connection (HTTPS). Most commonly it is implemented as an HTTP response header `Strict-Transport-Security`, which uses a *trust-on-first-use* [13] model. However, domains can also be specified in the *HSTS preload* [14] of web browsers, mitigating this drawback and leaving no space to an attacker, even on the first connection attempt. [15]

Redirection The server may respond to the request of unsecured version with HTTP status code 301 – Moved Permanently (or other 3xx), followed by the location of the secure version of the site. [16]

Ideally, redirection to the correct location should be handled by the web server itself. [17, 18] However, in some cases, it is implemented in the code running in the browser (e.g. JavaScript) loaded from the insecure version of the site. [19]

No confidentiality and integrity Even if we could use the plain HTTP, there is still a good argumentation against it, always using the encrypted TLS tunnel. The most important reasons are keeping a confidentiality and data integrity of the communication – no third party on the way can see, nor modify the content of the communication. [20] Otherwise, it would be possible to read the login credentials, bank account information, place an advertisement, virus, or cryptocurrency mining script into the content of any site, and more malicious actions, only limited by the creativity of an attacker.

The preferred solution should be to import the certification authority (Burp's self-signed certificate) into a browser's CA database and mark it as trusted to identify websites. The certificate can be obtained either from the GUI in the tab *Proxy – Options – Import/export CA certificate* or by visiting a special URL `http://burpsuite`, which is served from the local Burp's instance.

1.1.1.2 Tips for initial configuration

The tool is fully prepared for usage with the proxy configured and the Burp's CA imported according to the previous section. But before we dive into the features, there are several tricks to make life easier when using Burp extensively.

[2] Site intended to always run on plain HTTP, owned by a security researcher Scott Helme.

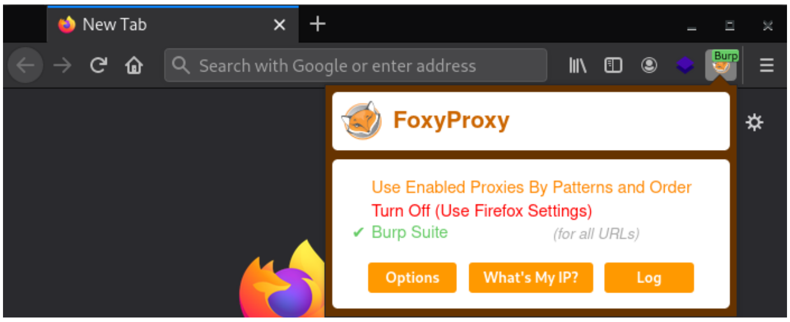

Figure 1.2: Browser extension allowing to quickly switch between different proxy configurations.

Sometimes, it is needed to let only specific requests into the Burp and other not. That leads to manually switching the proxy on and off quite often. The solution is to either find some pattern and configure automatic rules, or install a browser extension. One notable is FoxyProxy, which allows to configure several proxy entries and switch between them on one click from the corresponding menu in the top bar, as shown in Figure 1.2.

Another possibility is to use several browser profiles. This method has a great benefit, that each profile is completely separate, including history, settings, extensions, and other parts of the user profile. This way, we can, for example, create a *penetration testing* profile, which uses the proxy as discussed earlier, cleans the whole history and cookies when closed, and is visually different, to easily recognize on first sight which profile is in use. With profiles, we can have several opened windows, each with a different configuration. In Firefox, this can be achieved from a special URL `about:profiles` as shown in Figure 1.3.

Yet another possibility was brought by a quite recent update. It is now possible to use an integrated browser based on Chromium. The traffic goes right away to the proxy without configuring anything. The only disadvantage is that we have less control over the browser itself, which might not be ideal for every use case. This can be achieved with the steps shown in Figure 1.4.

1.1.1.3 Intercept

The intercept feature allows us to see and make decisions about all the requests before they are sent to the server. We can either make some modifications or forward the request further without any change. In case we do not like the request, we can drop it. Example is given in Figure 1.5. [21]

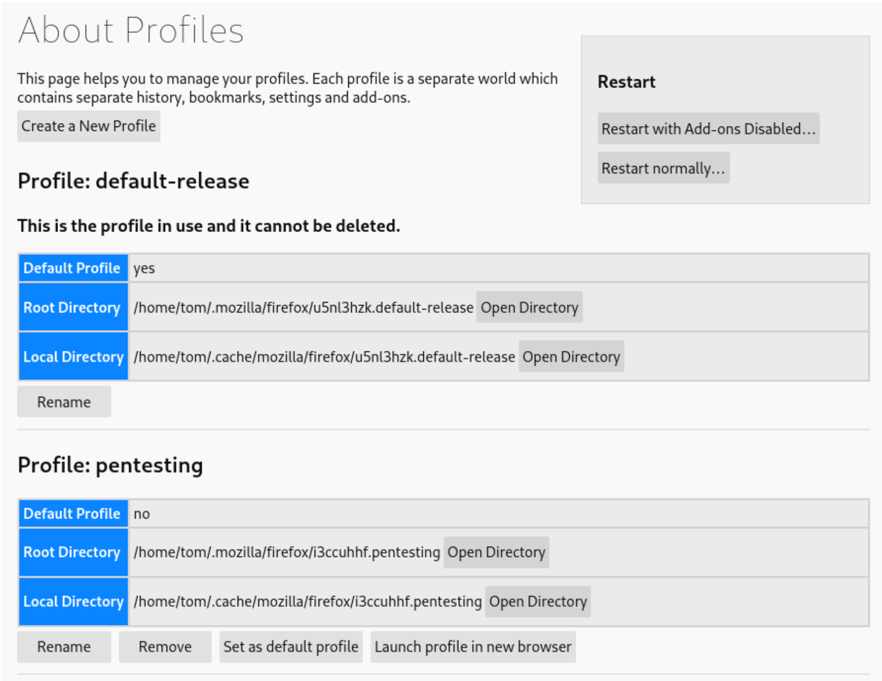

Figure 1.3: Special site (`about:profiles`) built into Firefox for managing multiple profiles.

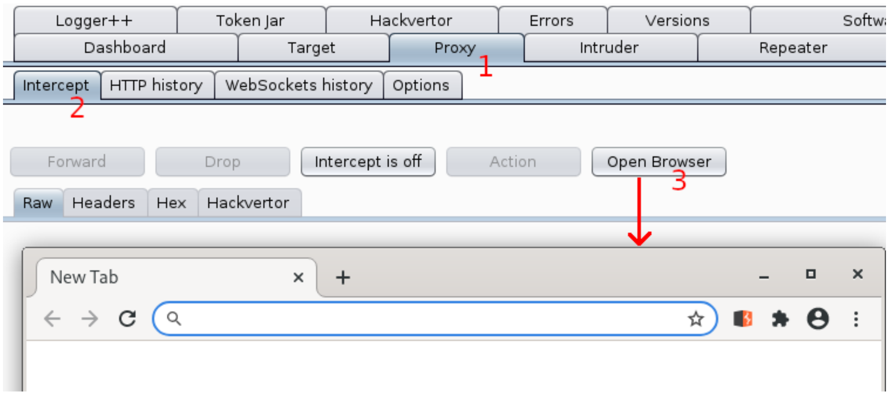

Figure 1.4: Using Burp's integrated web browser based on Chromium.

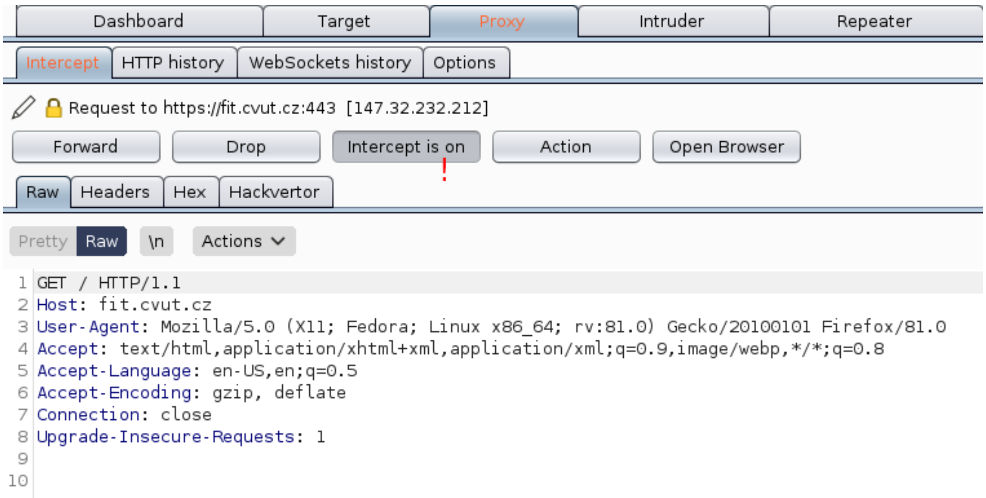

Figure 1.5: Intercepted GET request to `https://fit.cvut.cz/` is waiting for user's decision.

This feature can be handy in situations, where JavaScript validation prevents sending invalid data; however, the validation takes place only on the client's side. The server processes the data without any further checks. A similar case is character encoding before sending the request from the client. Insufficient user input validation, is an ordinary programmer's offence, which might introduce severe vulnerabilities, such as SQL Injection and Cross-site scripting (XSS). [22]

1.1.1.4 HTTP history

All the traffic going through Burp is also recorded, both requests and responses. It is possible to reach this information from the *Proxy – HTTP history* tab, as shown in Figure 1.6. Additionally, this section provides advanced filtering capability, allowing to specify various keys such as MIME type, response status code, specific string (including regular expressions), and much more. [23]

1.1.1.5 Automatic issue detection

Burp also tries to recognize some vulnerable patterns and reports specific findings with a short description and recommendation on how to mitigate them.

Often, this might be quite useful to quickly recognize the problematic places in the application, where to spend the most time, or just not overlook some severe issue. Nevertheless, it does not mean that all the reported vulner-

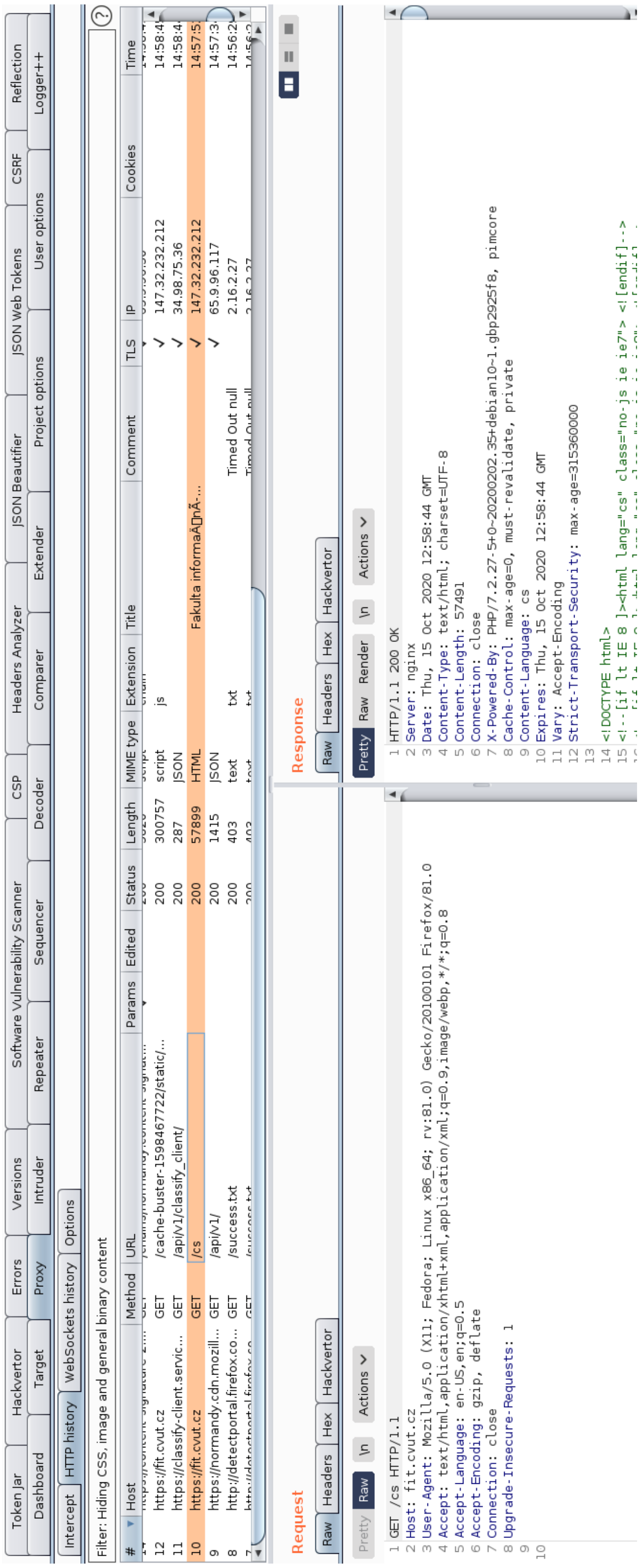

Figure 1.6: History of all the requests and corresponding server responses.

abilities are valid. The scanning has false positives as well as false negatives. It should be taken only as another input for a skilled security researcher who should have a final word. [24]

The detected issues are accessible from the *Target* tab. There is a sitemap in a tree format of the detected site content so far on the left side of this window. The right side displays related issues to the currently selected subtree. An example is shown in Figure 1.7. In this case, Burp was scanning a locally running instance of DVWA[3].

1.1.1.6 Other tools and extensions

There are many more useful features built-it (e.g. Intruder, Sequencer, Collaborator client). However, this section's goal was not to summarise the official documentation, but rather give some insight into what the tool is capable of and point out how big application it really is.

One more thing needs to be mentioned in the context of Burp. It is not just about the tools provided in the application, but there is also a great support for extensions, which play a significant role in the ecosystem. [26]

[3]Damn Vulnerable Web Application is a web application with many common vulnerabilities included on purpose. The goals are to help security professionals test their skills, improve the tooling, and educate others. [25]

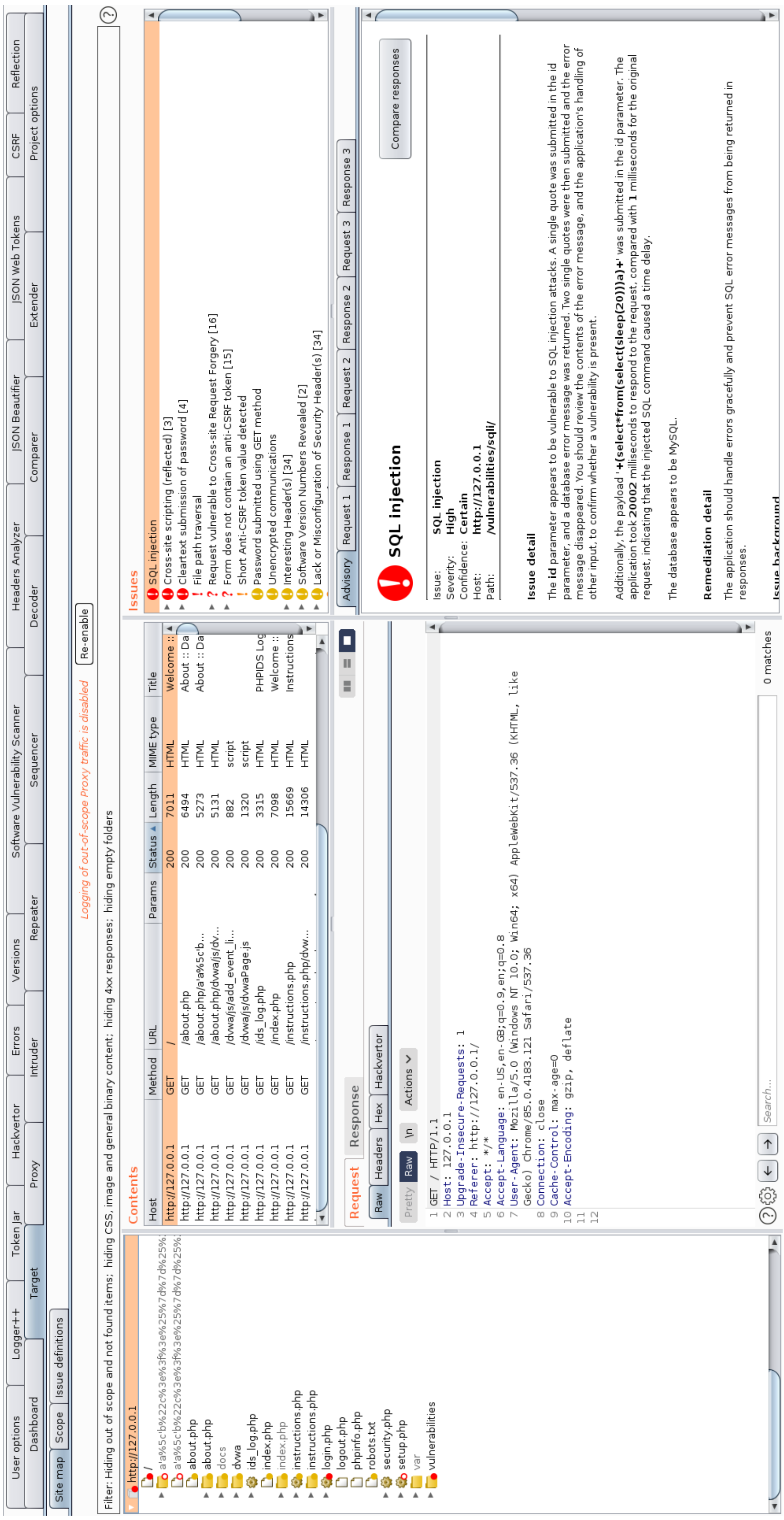

Figure 1.7: *Target* tab containing automatically detected issues.

Chapter 2

WebSockets

2.1 Overview

This chapter will explain the inner working of the WebSocket protocol. We will build upon this knowledge later in section 3.4, where we will develop a WebSocket fuzzer.

Many modern web applications require full-duplex[4] communication between a client and a server. Before WebSockets, the possible solution was to implement some polling mechanism, because the HTTP protocol uses the request-response model (the client sends an HTTP request, and the server responds with an appropriate response). [27]

The polling is a bypass from the request-response model, simulating a bidirectional communication, which the authors of the HTTP protocol did not incorporate into the specification. The trick is in sending additional requests from a client to the server. There are two types of polling, depending on the chosen strategy. *Short polling* strategy periodically sends a request to the server, expecting an immediate answer – either some data or empty message if the server has nothing to say. If the tolerated communication latency is low, this will generate an excessive load on the server and network. The other strategy is *long polling*, where the server responds to a request only when there is a message to be sent or timeout has occurred. Long polling allows lower latency and decreases the use of network and server resources. [28]

The WebSocket protocol, defined in RFC 6455, comes as a solution to the problem. It is a full-duplex communication protocol, using only a single TCP connection as an underlying layer. The commonly used ports for WebSockets are the same as for HTTP(S) communication – TCP ports 443 and 80 for secure communication using TLS layer, and insecure communication, respectively. Usage of the same ports and other similarities to the HTTP protocol (especially in the handshake part) are on purpose. It allows easy integration

[4]Communication in both directions.

with existing HTTP proxies, other intermediaries[5], and prevents some firewall issues. [29]

The registered URI[6] scheme is:

- `ws` for an insecure *WebSocket* protocol
- `wss` for *WebSocket Secure* (which uses TLS)

Example of the whole URI:

`wss://server.example.com/chat`

Except for a different scheme, the other distinction from HTTP URIs is that the WebSocket ones must not include fragment identifiers (starting with `#` character). [30]

2.2 Protocol details

The protocol is divided into two parts – a *handshake* for establishing the connection, and if that succeeds, a *data transfer* can follow. [31]

The WebSocket connection can be in one of the following states:

- `CONNECTING` – initial state; after the client sends a handshake
- `OPEN` – after successfully establishing the connection
- `CLOSING` – after either sending or receiving a closing handshake
- `CLOSED` – when the closing handshake is finished

2.2.1 Opening handshake

2.2.1.1 Client to server

The first part of the handshake, sent from the client to the server, is a GET request with an upgrade offer, which has to conform to the HTTP protocol. [31]

An example of such a request:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

[5] The server may be using a load balancer or a reverse proxy.

[6] Uniform Resource Identifier (URI) = scheme:[//authority]path[?query][#fragment]

Line 1 must be a Request-Line[7], as defined in RFC 2616 (Hypertext Transfer Protocol) with HTTP-Version at least 1.1[8]. The remaining lines are headers. Their ordering can be arbitrary; however, the following ones must be present in a valid handshake:

- `Host`: the value specifies the host and optionally a port number (if not the default value); obtained from URI of the origin server
- `Upgrade`: the value must include the "websocket" keyword
- `Connection`: the value must include the "Upgrade" token
- `Sec-WebSocket-Key`: a 16-byte random value; base64 encoded
- `Origin`: (required only for requests from a browser client); the value is the address from where the client started the handshake attempt
- `Sec-WebSocket-Version`: the value must be 13

Other headers, including `Sec-WebSocket-Protocol`, are optional. After this request is sent, the connection moves to the `CONNECTING` state. [33]

2.2.1.2 Server to client

The second part of the handshake, from the server to the client, is similar to the following example:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Line 1 is a Status-Line[9], which must contain the HTTP status code 101 for establishing the connection successfully. If any other value is present, the request must be processed as a standard HTTP request. For example, it can be a platform authentication request with a status code 401. Or redirection to another location, with status code 3xx. [33, 34]

Similarly to the handshake's first part, the leading line is followed by an unordered set of headers. The following headers are required for establishing the connection:

- `Upgrade`: the value must include the "websocket" keyword
- `Connection`: the value must include the "Upgrade" token

[7] Request-Line = Method `SP` Request-URI `SP` HTTP-Version `CRLF`
[8] For HTTP/2, there is RFC 8441 – Bootstrapping WebSockets with HTTP/2. [32]
[9] Status-Line = HTTP-Version `SP` Status-Code `SP` Reason-Phrase `CRLF`

- `Sec-WebSocket-Accept`: the value must be a base64-encoded[10] SHA-1[11] hash value of the concatenation of the string value received in the `Sec-WebSocket-Key` header with the string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"[12]

Other headers, including `Sec-WebSocket-Protocol`, are optional. After this response is sent, and if both parts of the handshake were successful, the connection state becomes `OPEN`. [33]

2.2.2 Data transfer

Unlike the handshake part, the data transfer is no longer similar to the HTTP protocol. WebSockets use their own set of rules, including the encapsulation into frames.

2.2.2.1 Framing

A frame is the raw format of the data. The WebSocket standard defines the frame as shown in Figure 2.1. Note that the sizes specified there are in bits. In a group of bits, the leftmost one is the most significant bit (MSB). [35]

Now to the meaning of individual fields [35]:

FIN The frames can be fragmented into several smaller ones. The protocol guarantees the delivered order of each frame. This `FIN` flag marks the final fragment from a message.

RSV1–RSV3 Reserved bits for future definition. The values must be zero if the application does not know their meaning.

Opcode Numeric code specifying the type of the frame.

- `0x0`: continuation frame
- `0x1`: text frame
- `0x2`: binary frame
- `0x3`–`0x7`: reserved (non-control frames)
- `0x8`: connection close
- `0x9`: ping
- `0xA`: pong
- `0xB`–`0xF`: reserved (control frames)

[10] As defined in section 4 of RFC 4648.
[11] As defined in FIPS.180-3. The produced hash is 160 bits long.
[12] This string is a Globally Unique Identifier (GUID), as explained in RFC 4122.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Figure 2.1: WebSocket data frame

Mask The payload data sent from the client must be masked (XOR with a masking key). If this bit is set, the mask was used. Otherwise, this bit must be clear.

Payload length Length of the payload data in bytes. If the value is lower or equal to 125, that is the length. The value 126 indicates that the length is in the following two bytes (as an unsigned integer). The value 127 indicates that the following 8 bytes are the payload length (again, as an unsigned integer).

Masking-key Random value, present only if the mask flag is set.

Payload data The combination of *Extension data*, if an extension was negotiated, and *Application data*.

2.2.2.2 Control frames

The control frames have the *opcode* with a value `0x8` and higher. [36]

The *close* frame (*opcode* `0x8`) indicates the end of a connection and the payload data. If not empty, it may contain the reason. No further frames should follow. If an endpoint receives the *close* frame first, it must also respond with a *close* frame. [36]

Ping (*opcode* `0x9`) and *pong* (*opcode* `0xA`) are control frames for detecting if the other side of the connection is still alive. An endpoint receiving a *ping* frame must respond with a *pong*, containing the same *payload data*. [36]

2.2.2.3 Data frames

Data frames are carriers of the actual data. At the moment, only two types of *data* frames are defined [37]:

Text The payload is a UTF-8 encoded text.

Binary The payload are binary data, without any more specific restrictions. The data representation is up to the developers of individual applications.

2.2.2.4 Masking

Masking the payload data is required when sending a message from a client to a server. When masking is applied, the *mask* flag must be set. The masking key is randomly generated four bytes, sent as part of the frame. [38]

The conversion between masked and unmasked data is the same in both directions, according to the following formula:

$$T_i = D_i \oplus K_{i \bmod 4}$$

where

- T_i is the transformed data at index i
- D_i is the original data at index i
- $\oplus$ is an XOR operation
- K_i is the masking key at index i mod 4

The index i goes from 0 to the (length of the payload minus 1), and it is an index to individual bytes. This masking process preserves the length of the data[13]. [38]

2.2.2.5 Examples

To give a realistic idea, how such frame may look like, let us have a look at two examples.

The first one is a text data frame from a server (unmasked data), sending a short message “server”. The *FIN* flag will be 1, because the message is short and no fragmentation is needed. *RSV1–3* has to be unset (0); *opcode* for a text data frame is `0x1`; length of the payload data is 6 bytes[14], and finally, the data are `0x73 65 72 76 65 72`. The visual representation of this frame can be seen in Figure 2.2. The final binary representation is `0x81 06 73 65 72 76 65 72`.

[13] XOR operation may only flip individual bits but does not change the length of the data.

[14] Six characters in UTF-8 representation could lead to more than six bytes. However, in this case, only ASCII characters are present, which are always one byte long.

```
0                   1                           6
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6        ...    4
+-+-+-+-+-------+-+-------------+---------------------+
|F|R|R|R| opcode|M| Payload len |    Payload data     |
|I|S|S|S|  (4)  |A|     (7)     |                     |
|N|V|V|V|       |S|             |                     |
| |1|2|3|       |K|             |                     |
+-+-+-+-+-------+-+-------------+---------------------+
|1|0|0|0|0 0 0 1|0|0 0 0 0 1 1 0|        ...          |
+-+-+-+-+-------+-+-----+-------+---------------------+
|  0x8  |  0x1  |  0x0  |  0x6  | 0x73 65 72 76 65 72 |
+-------+-------+-------+-------+---------------------+
```

Figure 2.2: WebSocket example – *server* message

```
0                   1                           4                   9
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6      ...      8        ...        6
+-+-+-+-+-------+-+-------------+---------------+---------------------+
|F|R|R|R| opcode|M| Payload len |  Masking key  |    Payload data     |
|I|S|S|S|  (4)  |A|     (7)     |     (32)      |                     |
|N|V|V|V|       |S|             |               |                     |
| |1|2|3|       |K|             |               |                     |
+-+-+-+-+-------+-+-------------+---------------+---------------------+
|1|0|0|0|0 0 0 1|1|0 0 0 0 1 1 0|      ...      |        ...          |
+-+-+-+-+-------+-+-----+-------+---------------+---------------------+
|  0x8  |  0x1  |  0x8  |  0x6  | 0x1a 2b 3c 4d | 0x79 47 55 28 74 5f |
+-------+-------+-------+-------+---------------+---------------------+
```

Figure 2.3: WebSocket example – *client* message

The second example switches the roles. It is coming from a client and thus needs to apply the mask. The text message is "client" – `0x63 6c 69 65 6e 74` in UTF-8 representation and the masking key was for demonstration chosen as `0x1a 2b 3c 4d`. The payload data can be computed as (`0x63 XOR 0x1a`) for the first byte, (`0x6c XOR 0x2b`) for the second byte, up until the fourth one. Then the index over the masking key resets to the beginning – (`0x6e XOR 0x1a`) for the fifth byte, and so forth. The final payload data is `0x79 47 55 28 74 5f`. The updated visualization can be seen in Figure 2.3. The binary representation of this frame is `0x81 86 1a 2b 3c 4d 79 47 55 28 74 5f`.

Even though the second message has the same length and *opcode* as the first one, the fact that masking needs to be applied makes it four bytes longer (of the masking key field).

Chapter **3**

Practical part

In this chapter, we will first search for Burp's web endpoints (including REST API, which is turned off by default), manually investigate the suspicious places, and run vulnerability scans. Then, we will develop an application for fuzzing the WebSockets protocol implementation. In the end, we will analyse the network traffic generated by Burp.

3.1 Burp's web endpoints

Although the most user interaction with Burp happens in the GUI part of the application, it also provides several web endpoints.

By default, there is a running[15] web interface on `http://burpsuite` address, with aliases `http://burp`, and `http://localhost:8080` (or other port, depending on your proxy configuration). The first two URLs are top-level domains, however not included in the DNS root zone overseen by IANA[16]. It can work this way because the request does not reach the Internet. Instead, it is served locally from Burp. The page content is a welcome message, and a link to the public part of Burp's certificate used to serve HTTPS traffic to clients (endpoint `/cert`). Once we add this certificate between the trusted ones, it is possible to access also HTTPS version of all previously mentioned sites.

The entire content of the page `http://burpsuite`, as rendered by browsers, can be seen in Figure 3.1.

3.1.1 Finding other endpoints

During a security assessment, finding any error messages may turn out to be quite useful. In our case, we can try to access a site that does not exist

[15]It can be turned off in the *Proxy – Options* configuration.

[16]Internet Assigned Numbers Authority is responsible for the management of the top-level domains (like `cz`, `com`, and `org`), besides other things. [39]

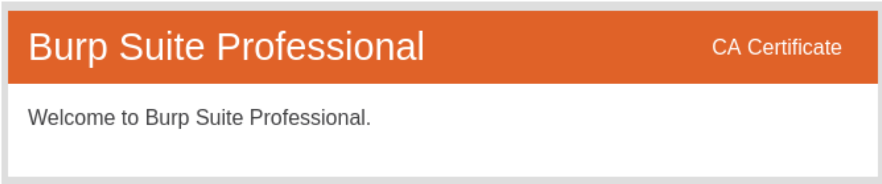

Figure 3.1: Burp's index page

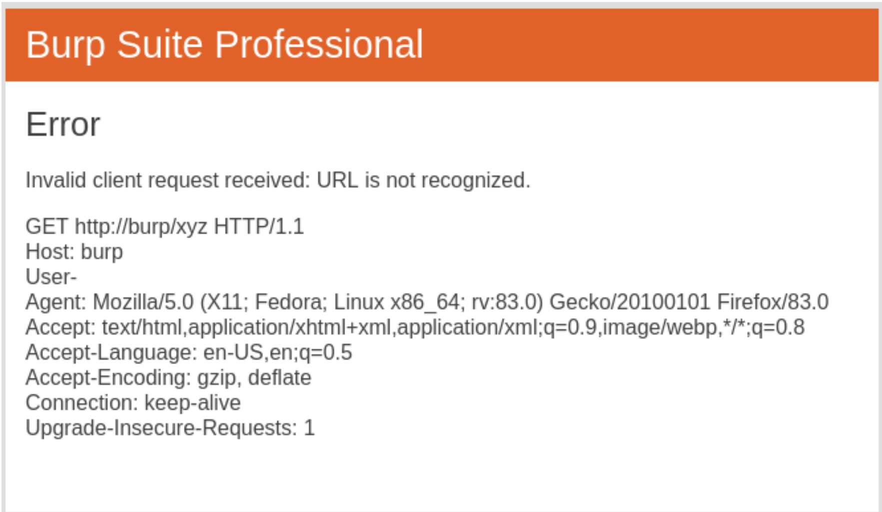

Figure 3.2: Burp – `xyz` site does not exist

– `http://burp/xyz`. And indeed, that led the application into giving us something more interesting, as shown in Figure 3.2.

We can see that the raw version of the request is included in the error output. That immediately raises suspicion about reflected cross-site scripting (XSS) vulnerability.

3.1.1.1 What is cross-site scripting (XSS)

Cross-site scripting arises when the user input validation is not done correctly by the application developers, and that leaves enough space to an attacker to run arbitrary code in the browser. This method is classified as an injection attack, just like SQL injection. In addition to customizing the site for himself, which is usually not the goal, an attacker can[17] impact other users by steal-

[17] If no other defence is in place, such as character encoding, HttpOnly cookie flag, and Content Security Policy. [40, 41]

ing their session tokens or sensitive information from the page content, and redirecting them to other location controlled by him. [42]

We can distinguish between several types of XSS[18] based on the conditions needed to execute the code:

Reflected XSS happens when a user's input is included right away in the response, and the injected payload is executed. To effectively exploit this vulnerability, an attacker may try to deliver a malicious link to his victim. [42]

Stored XSS usually requires more steps of user interaction (or multiple actors). In the initial phase, the payload is saved to the database, probably by an attacker himself. The code execution is triggered later when other action results in including this data into a response's content.

This type can also be *wormable*, which means it can copy itself and infect others when executed. One such famous case from the past is a worm Samy (alternatively called JS.Spacehero) used on MySpace social network in 2005. It managed to infect over one million users in twenty hours. [43, 44]

DOM-based XSS is other, lesser-known type, taking advantage of the specific properties of a site. An example might be the usage of a `window.location` (current URL) value in an `eval` function. [45]

3.1.1.2 Check for reflective XSS

Back to our case, instead of `xyz` site, we can insert the script tags with some JavaScript code inside and see if anything happens. The most common payload to try, if any given script on input gets executed, is `<script>alert(1) </script>`. The `alert` function is easy to spot when executed – displays a pop-up message in the browser with the text given as a parameter. In this case, a window with number 1[19] should appear.

Accessing the site `http://burp/<script>alert(1)</script>` from a web browser results in a similar error message as with the `xyz` site, only this time containing leading line of the included request:

```
GET http://burp/%3Cscript%3Ealert(1)%3C/script%3E HTTP/1.1
```

[18] Some applications may be vulnerable to multiple XSS types simultaneously – e.g. reflected and stored XSS at once.

[19] We can insert any string (e.g. "pwned by XSS") to the `alert` function, however using a number has a great benefit that we do not have to use another special character – quotation marks. Moreover, the goal is only to find out if **any** script can get executed. Thus the payload should be as simple as possible.

The characters %3C and %3E indicate a URL encoding. That happens automatically when sending the request from a web browser. To bypass this behaviour, we can send the request e.g. from other Burp's instance, which was made for such purposes. This attempt is shown in Figure 3.3.

An alternative to the repeater feature in Burp might be a command-line tool cURL:

```
$ export http_proxy=http://localhost:8080/
$ curl 'http:/burp/<script>alert(1)</script>'
```

which can achieve the same results:

```
[...]
<h1>Error</h1><p>Invalid&#32;client&#32;request&#32;received&#58;&#32;URL&#
↪   32;is&#32;not&#32;recognized&#46;</p>
<div class="request">GET http://burp/&lt;script&gt;alert(1)&lt;/script
↪   &gt; HTTP/1.1<br>
[...]
```

As a result, the text slightly changed; this time, the special characters reached the server. And we can see that the malicious action was handled gracefully with an HTML encoding ("<" as "<", ">" as ">"), which is an effective defence against XSS attacks. [46]

There are other techniques for achieving the same results, even without using the `script` tag if certain conditions are met. E.g. if input from a user is included in some attribute, we can try to escape from it with a quotation mark and then inject custom attribute like `onload`, `onerror`, `onmouseover`, or similar, which accept JavaScript function as its value. However, no such place is present here.

3.1.1.3 Replay/show in browser feature

The functionality of the web interface does not end with downloading of the Burp's CA certificate. It has at least two more use cases.

It is possible to show a specific response or replay the same request in a web browser from the proxy history. It is achieved through generating a special link that should be copied to a web browser (which has to be connected through the proxy). The format of a link to show a specific response in a web browser is `http://burpsuite/show/1/ptb0jdoa7jsbjmibrea7ny4758hhj6zo`. The part after `show` is a sequential number, incremented with each new link generated. The remaining part is 32 characters, probably randomly generated for each action.

Replaying a selected request from the history has a very similar procedure, only the generated URL looks like the following example – `http:`

Figure 3.3: Burp – testing reflective XSS

```
{
    "project_options":{
        "connections":{
            "upstream_proxy":{
                "servers":[
                    {
                        "destination_host":"*",
                        "enabled":true,
                        "proxy_host":"127.0.0.1",
                        "proxy_port":8899
                    }
                ],
                "use_user_options":false
            }
        }
    }
}
```

Figure 3.4: Configuration file for an upstream proxy

`//burpsuite/repeat/1/teuhnu9cit72mm5akt1v9sswrklr46cg`. The location changed from `show` to `repeat`, followed again by the sequential number of the action and 32 character long nonce[20].

3.1.1.4 Enumerating sites

We already know about several sites in the `http://burpsuite` web interface. Nevertheless, we can try to enumerate more of them, if any additional exist.

First of all, since this will be more resource-intensive job, we can this time start Burp in a headless mode (running in the background, without the GUI part):

```
$ java -Djava.awt.headless=true \
    -jar Downloads/burpsuite_pro_v2020.11.jar \
    --disable-extensions \
    --use-defaults \
    --project-file=[...]/project.burp
```

Optionally, a parameter `--config-file` can point to a JSON file with a custom configuration. Example setting up an upstream proxy is shown in Figure 3.4.

From the previous part, we found an easy way to determine if a site does not exist. If that is the case, the response contains the following string:

[20] Unpredictable (pseudo-) random value, which can be used only once. [47]

```
Invalid&#32;client&#32;request&#32;received&#58;&#32;URL&#32;is&#32;not&#32;recognized&#46;
```

We can now use this knowledge to enumerate all[21] the sites. There are many tools to perform this task. Even Burp has an Intruder feature that can be used for such purpose. However, after playing a bit around, Ffuf was chosen as the right tool for this task. It is a command-line fuzzer written in Go, aiming for a great performance[22]. [48]

Ffuf takes a wordlist on the input and using a multithreaded approach tries all the entries. In the language of Ffuf, we can define *matchers*, which are conditions for the entry to be printed (e.g. specific status code, response length, regular expression, ...), and *filters*, which have the same condition, but if met, the entry is silently skipped.

The first test was made with a famous dictionary *rockyou* based on the RockYou company breach in 2009. The incident leaked passwords (stored in plaintext) of 32,603,388 accounts. When the duplicates were removed, the count was 14,341,564 unique passwords. Despite its origin in a passwords leak, it is useful also for other cases, since it contains all the most common words and phrases. This test revealed one so far unknown location `/caset`. However, in contrast to other sites, this one acts exactly as the index page (at least a response to a GET request). We worked with the hypothesis that it might allow importing custom certificate through a POST request (the word *caset* can be split to form phrase "CA set"). But we did not manage to verify it. [49, 50]

However, the detected location was a motivation to be more throughout with the enumeration and generate a brute-force dictionary. The first one is assembled from lower alpha-numerical characters of length 1–5 characters. The size of the result is 371 MiB, and the brute-forcing process took around 7.5 hours. The test confirmed the two already known locations – `/cert`, and `/caset`, but did not discover anything new. We continued with a second dictionary containing only alphabetic characters and the length set to exactly six characters. The result was a text file with an approximate size of 2.2 GiB. Despite its huge size and significant time to test all the entries (38 hours), nothing new was discovered.

Since increasing the length further would be even more space and time-consuming, and chances to uncover some hidden parts were negligible, we decided to end up here.

[21] Theoretically, if we had an unlimited amount of time.

[22] This specific task's result was the speed in the range between 2000 and 3000 requests per second, including matching each of them against a regular expression to decide whether to print it or not. Against the Burp's Intruder was a fact that with the increased number of requests, it was becoming unresponsive.

Verb	Endpoint	Parameters	Response
GET	/knowledge_base/issue_definitions	{ }	200 OK, [IssueDefinition]
POST	/scan	{"scan": Scan}	201 Created, Location header
GET	/scan/[task_id: String]	{"after": String, "issue_events": Integer}	200 OK, ScanProgress

Figure 3.5: REST API page

3.1.2 REST API

Another part of the web interface is disabled by default – the REST API for integration with other applications. After enabling it (*User options – Misc – REST API*), the default address is `http://127.0.0.1:1337/` followed by an API key[23] and version, e.g. `http://127.0.0.1:1337/1iLrHvHQlQWs7POJSgquTTz4h7j7aXQz/v0.1/`. The content of this page is shown in Figure 3.5. It summarises all available endpoints, plus the image in the top right corner is a link to detailed specification in an OpenAPI [51] format.

There are only three REST API endpoints available in the current version `v0.1`. The `/knowledge_base/issue_definitions` returns a list of all issues known to Burp, with a description, possible remediations, references, and other information. The `/scan` starts a new scan with provided parameters. Finally, the `/scan/[task_id]` returns the progress of a scan with provided ID, containing the current status (paused, crawling, succeeded, . . .), detected issues, and other information.

The page with the REST API endpoints summary also offers one additional feature, which is slightly hidden. After making a click on any of the endpoints in the table, a modal window appears (as shown in Figure 3.6), allowing to choose values for the parameters and try out the request straight away from the web browser. Additionally, the page contains a cURL command of the same request (including the configurable dynamic parameters). It can help to understand all the details in the API quickly.

A “Send Request” button directly below the cURL line may invoke an idea that it is this line that is executed in a shell on the host system, and that could lead to OS command injection, if the input was not sanitized properly.

[23] The REST API keys can be created and deleted in the Burp’s configuration, specifically in the *REST API* section in *User options*. Once created, the keys cannot be displayed again.

A hypothetical exploit of the case shown in Figure 3.6 would be to use the following payload in the `task_id` text input:

```
'; cat /etc/passwd #
```

The first apostrophe character ends the current string. A semicolon is a delimiter between commands in shell (alternatives are `&&` and `||` operators, that run the next command depending on the previous one's return value). The part that follows is a command we would like to execute – `cat /etc/passwd` prints all the users on the system, together with their properties. The last character `#` introduces a comment, ensuring the rest of the command does not cause any syntax problems.

However, this cURL line is not the command that gets executed. Instead, the GET and POST requests are sent straight away from the web browser, and the cURL line is only informational. That means no OS command injection is possible in this place.

3.1.3 Found web endpoints summary

Summary of all the identified web endpoints follows:

```
http[s]://burp[suite] ≡ http[s]://localhost:<proxy_port>
 ├── /
 ├── /caset
 ├── /cert
 ├── /repeat
 └── /show
```

The `/` contains a welcome message and a link to `/cert`, which contains a Burp's CA self-signed certificate. The `/caset` acts similarly as `/`, however, no deeper meaning was discovered. The `/repeat` and `/show` sites are used in a web browser for replaying a request, respectively showing a response from the history.

On the REST API part:

```
http[s]://localhost:<rest_api_port>/<api_key>/v0.1
 ├── /
 ├── /knowledge_base/issue_definitions
 ├── /scan [POST]
 └── /scan/<id>
```

There is a summary page in the root (`/`); definitions of all the issues under `/knowledge_base/issue_definitions`; it is possible to start a new scan with

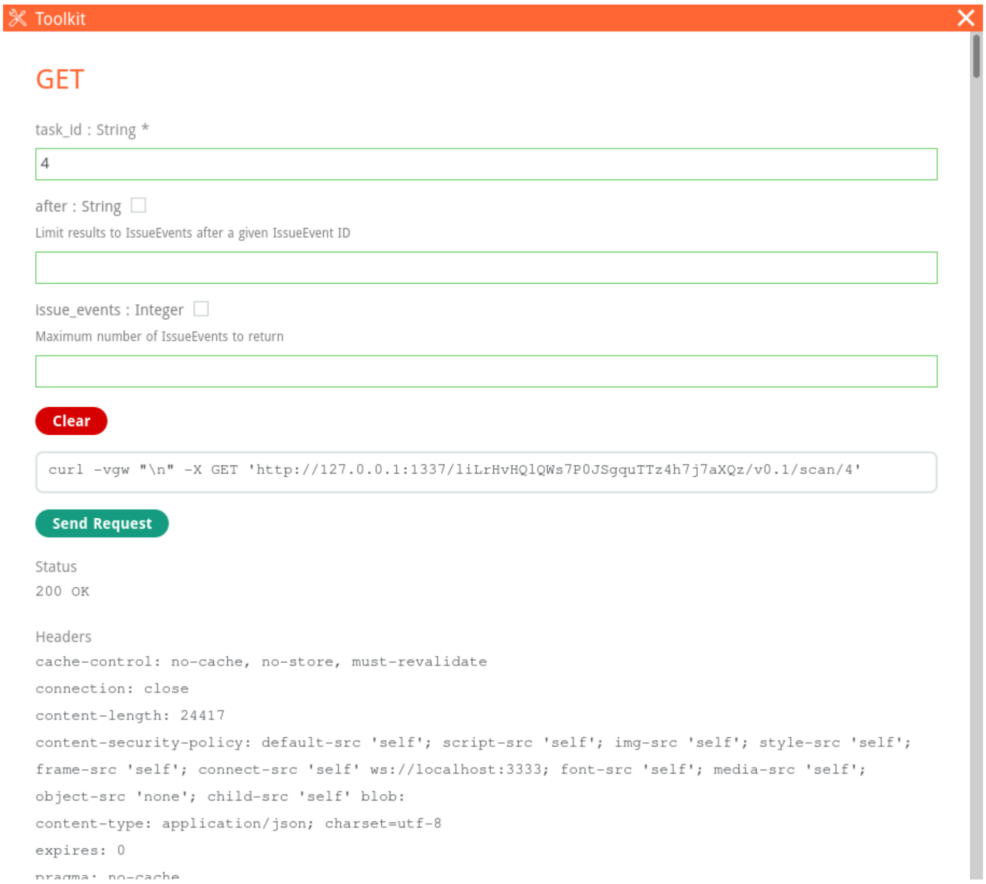

Figure 3.6: REST API – try out the request from web

a POST request to `/scan`, and to get back the status of a specific scan under `/scan/<id>`.

3.2 Scanning

In section 3.1.3, we made a summary of the identified endpoints. We can now run some automated scans to examine the security properties of the webserver. First, we will conduct an overall webserver scanning using Nikto. Later, we will use Burp itself to scan other running instance of Burp.

3.2.1 Nikto

We can start with *Nikto* – an open-source (GNU GPLv2) project. Nikto mostly detects well-known locations on the server or specific strings from responses. These often indicate particular (vulnerable) software being present, or a site not intended for a general public (administrator interface, load balancer configuration, and similar). [52, 53]

If we simply run:

```
nikto -h localhost:8080
```

the tool returns a huge amount of results. A snippet follows:

```
+ /forums//adm/config.php: PHP Config file may contain database IDs and
↪  passwords.
+ /forums//administrator/config.php: PHP Config file may contain database
↪  IDs and passwords.
+ /forums/config.php: PHP Config file may contain database IDs and
↪  passwords.
+ /guestbook/guestbookdat: PHP-Gastebuch 1.60 Beta reveals sensitive
↪  information about its configuration.
+ /guestbook/pwd: PHP-Gastebuch 1.60 Beta reveals the md5 hash of the admin
↪  password.
```

We can try to access some of the reported locations. The result is an error page (same as in Figure 3.2). Nikto is probably confused because all the responses have a status code 200 (including pages which were not found and should have a status code 404). Luckily, there is a parameter for this situation (`-404string`), which tells Nikto that responses containing provided string (or regular expression) should be considered as not found:

```
nikto -h http://localhost:8080 -404string '<h1>Error</h1>'
```

This time, the output seems more sane:

```
- Nikto v2.1.6
---------------------------------------------------------------------------
+ Target IP:          127.0.0.1
+ Target Hostname:    localhost
+ Target Port:        8080
+ Start Time:         2020-12-23 13:51:33 (GMT1)
---------------------------------------------------------------------------
+ Server: No banner retrieved
+ The X-XSS-Protection header is not defined. This header can hint to the
↪  user agent to protect against some forms of XSS
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ Web Server returns a valid response with junk HTTP methods, this may cause
↪  false positives.
+ OSVDB-3092: /3rdparty/phpMyAdmin/Documentation.html: phpMyAdmin is for
↪  managing MySQL databases, and should be protected or limited to
↪  authorized hosts.
+ OSVDB-3092: /phpMyAdmin/Documentation.html: phpMyAdmin is for managing
↪  MySQL databases, and should be protected or limited to authorized hosts.
+ 4933 requests: 0 error(s) and 4 item(s) reported on remote host
```

```
+ End Time:           2020-12-23 13:53:25 (GMT1) (112 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Manually checking all the reported items, the `X-XSS-Protection` header is really not included in the server responses. However, this header is only an additional layer of protection against XSS attacks. Its absence does not introduce the vulnerability, and there might be other sufficient defences in place – e.g. character encoding and nowadays more popular header Content Security Policy. The remaining items were assessed as a false positive. [54]

Running Nikto for the REST API part:

```
nikto -h http://127.0.0.1:1337/PGD6ZD4VvdiTzvWqwoUpNh8MIYDCnBuq/v0.1/
```

did return:

```
- Nikto v2.1.6
---------------------------------------------------------------------------
+ Target IP:          127.0.0.1
+ Target Hostname:    127.0.0.1
+ Target Port:        1337
+ Start Time:         2020-12-23 15:06:30 (GMT1)
---------------------------------------------------------------------------
+ Server: No banner retrieved
+ Uncommon header 'x-burp-version' found, with contents: 2020.12-5207
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ Allowed HTTP Methods: DELETE
+ OSVDB-5646: HTTP method ('Allow' Header): 'DELETE' may allow clients to
↪  remove files on the web server.
+ 5073 requests: 0 error(s) and 3 item(s) reported on remote host
+ End Time:           2020-12-23 15:06:38 (GMT1) (8 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

The first part is rather informational. However, the reported DELETE HTTP method is much more interesting. We managed to verify that the server accepts this method – after sending a request with OPTIONS method, the response started with:

```
HTTP/1.1 405 Method Not Allowed
Allow: DELETE
```

The `Allow` header specifies a set of HTTP methods acceptable by an endpoint. [55] Probably something similar was what Nikto detected. Our initial thought was that this might allow deleting a specific scan with provided ID, but this is not the case. After playing a bit around, we managed to find out the meaning – it is another (undocumented) REST API call for shutting down

Figure 3.7: Burp – undocumented REST API call

the application. This request can be seen in Figure 3.7. The appropriate response sent before shutting down the application indicates that this behaviour is intentional (not an application crash).

In our opinion, all the REST API calls should be documented and listed in the summary page. Some people may, for example, turn off the authorization with an API key in a belief that only the listed actions are available and based on incomplete information evaluate, that no harm is possible. Moreover, in case of issuing the API keys to multiple people, the responsible person should have all the information about possible actions with this key.

In the response from Figure 3.7, there is one additional detail worth mentioning. The `Content-Security-Policy` header contains an address `ws://localhost:3333`. But simply connecting to it did not work, and we did not manage to find the purpose of this address for the REST API.

3.2.2 Burp

To test the web interface from Burp, we need to start two instances at once and set one as an upstream proxy for the other.

The first scan was targeted towards the web interface – `http://localhost:8080`. The generated sitemap from the scan can be seen in Figure 3.8. Not all the endpoints were automatically detected, so we needed to give the tool some hints. The reported items were:

- *unencrypted communication* – while true, it is only running locally and making it possible to use TLS, if we desired to have the connection encrypted
- *input returned in response (reflected)* – we already investigated this in section 3.1.1.2, when we noticed that an error message contains the original request

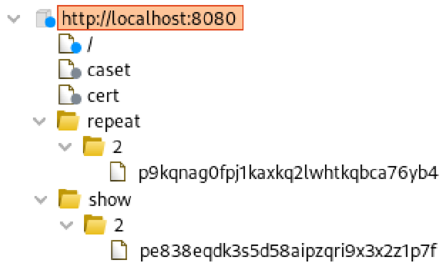

Figure 3.8: Burp's web interface sitemap

- several other items, which are rather informational

The next scan was of the REST API interface. We provided only the base address with an API key, and the tool managed to identify all the endpoints (including various resources like fonts, images, and JS files), as shown in Figure 3.9. No severe issue was detected even this time; the reported items were again rather informational. In addition to the *unencrypted communication* and *input returned in response (reflected)*, we were notified that an email address was detected in one of the files (`helpdesk@example.com`), that the site is using `Underscore.js` in version 1.8.3, located at `/static/js/bundle.js`, and few other similar cases.

3.3 Non-compliance with the HTTP protocol

While trying various HTTP GET requests for the proxy's welcome page, we noticed that the server is quite flexible and accepts even invalid requests according to the HTTP Protocol. [56] For example, the following request sent to the target `http://127.0.0.1:8080` was accepted (as shown in Figure 3.10):

```
NonExistingMethod / InvalidHTTPVersion
```

The response is "*Welcome to Burp Suite Professional*" message with status code 200, the same as in Figure 3.1.

Note that despite Burp is mainly a web proxy, the specific parts of the application discussed in the following part act as a regular webserver – directly serving an HTTP content, rather than being an intermediate between two endpoints.

3.3.1 Host header

According to the Hypertext Transfer Protocol – HTTP/1.1, which the server understands and uses (as we can see in its responses, e.g. in Figure 3.10),

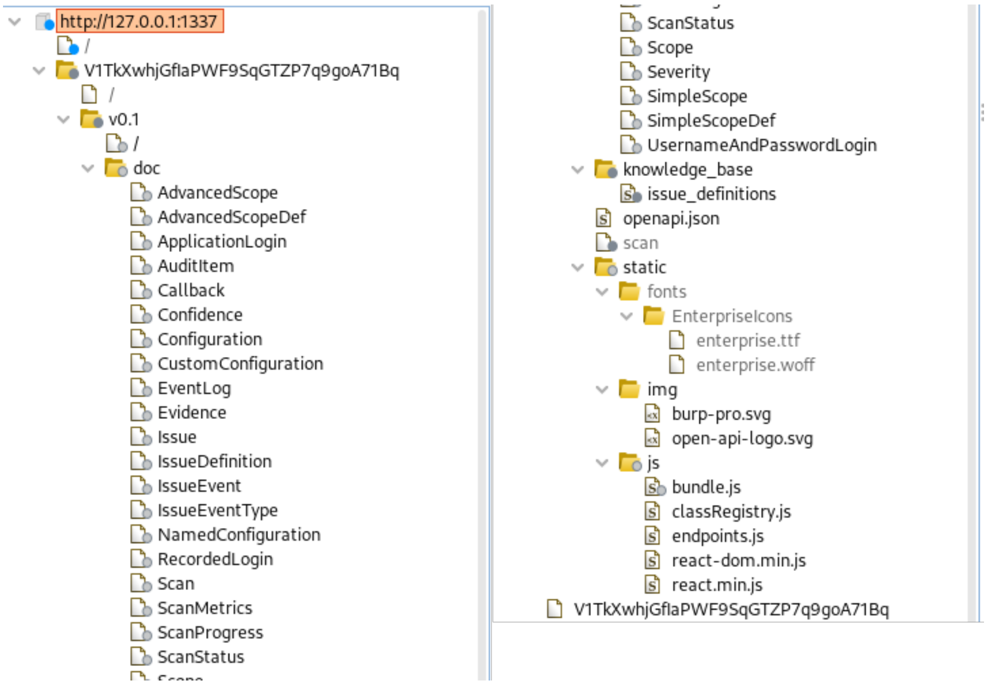

Figure 3.9: Burp's REST API sitemap

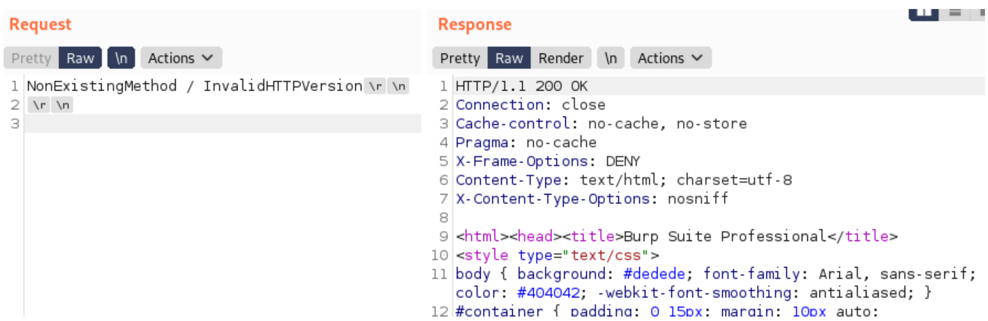

Figure 3.10: Invalid HTTP request being accepted

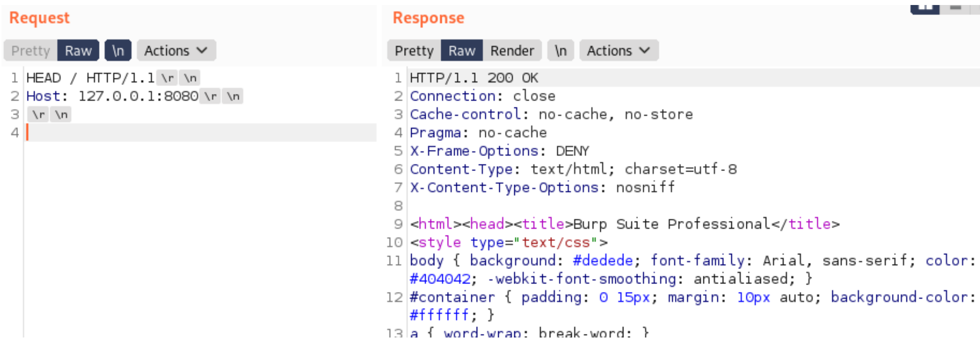

Figure 3.11: Sending a request with a HEAD method

`Host` header must be included in the request. Quoted from section 14.23 – "*A client MUST include a Host header field in all HTTP/1.1 request messages*". That makes the request sent by us to the server invalid, according to this protocol. [57]

The protocol also defines what should be the response in that case. From the same section, few sentences further – "*All Internet-based HTTP/1.1 servers MUST respond with a 400 (Bad Request) status code to any HTTP/1.1 request message which lacks a Host header field.*" [57]

We could see (e.g. in Figure 3.10) that this is not the case. The response had a status code 200.

3.3.2 HEAD method

There is another violation of the HTTP protocol in the handling of the HEAD request method. Section 4.3 of the RFC 2616 contains the following statement: "*All responses to the HEAD request method MUST NOT include a message-body [...]*". The response to a request with the HEAD method containing a message body is shown in Figure 3.11. [58]

3.3.3 Expect header

The `Expect` header (used commonly with the value "100-continue") is used to check if the server will accept the request, e.g. before sending a huge message body. An example from Mozilla's MDN Web Docs [59]:

```
PUT /somewhere/fun HTTP/1.1
Host: origin.example.com
Content-Type: video/h264
Content-Length: 1234567890987
Expect: 100-continue
```

In section 8.2.3, the HTTP protocol demands that the server responds to

Figure 3.12: Sending a request with an `Expect` header

the "`Expect: 100-continue`" header with either status code 100 (while it must not wait for the request body before sending it), or final status code. If the latter is the case, it must not perform the requested method. [60]

Sending a request with an `Expect` header to Burp is shown in Figure 3.12. Despite the presence of this header, the server processed the requested method (GET).

While the non-compliance with standards is not a security issue, it can confuse people and tools when interacting with the software. Moreover, the misunderstanding of the functionality caused by this deviation from standards could cause severe problems. We recommend following the standards if there is no rational reason against it.

3.4 Writing a WebSocket fuzzer

From the research of existing work in WebSocket fuzzing, it seems that the vast majority of projects focus on testing the applications on the other side of the WebSocket connection, rather than evaluating the implementation of the underlying protocol. The process is to take the payload data field, decompose the data representation used by the application (JSON is a commonly used format) into individual tokens, replace them with values that might cause some trouble on the other side (if sanitisation was not done properly), and monitor the responses. The same principle is used with regular dynamic web application testing. Only the underlying connection is substituted from WebSockets to HTTP.

The following section describes a similar high-level approach, only concentrating on the payload data (either text or binary). But later, we will make a step further and start fuzzing individual fields (some of them with a length of a single bit) of the WebSocket frame itself. The topology used for the fuzzing is shown in Figure 3.13.

Figure 3.13: Burp's fuzzing topology

The two versions of a fuzzer application and a simple server were written in Python programming language. This language is not a rare choice among security professionals, due to its ability to quickly prototype the testing applications thanks to a high-level syntax [61] and significant extensibility through libraries [62].

3.4.1 Creating a local WebSocket server

Although some public WebSocket servers are available (e.g. `wss://echo.websocket.org`), allowing us to test and debug an application, our traffic will be higher and contain malformed frames, which is not suitable for public service. Fortunately, Python's *websockets* library [63] allows us to build a local WebSockets server quickly:

```
#!/usr/bin/env python
# Based on examples from https://pypi.org/project/websockets/

import asyncio
import websockets

LISTEN_ADDRESS = ('localhost', 8050)


async def msg_handler(websocket, path):
    async for message in websocket:
        await websocket.send(message)


if __name__ == '__main__':
    start_server = websockets.serve(msg_handler,
                                    host=LISTEN_ADDRESS[0],
                                    port=LISTEN_ADDRESS[1],
                                    compression=None)

    asyncio.get_event_loop().run_until_complete(start_server)
    asyncio.get_event_loop().run_forever()
```

Now, we have a local server running on `ws://localhost:8050` that repeats back the same data we sent there[24].

[24]For other than text and binary frames (e.g. control frames), it acts according to the

3.4.2 High-level fuzzing

There is one problem with Python's *websockets* library – it currently does not support connection through an HTTP proxy, which is the component we would like to test. This feature might be coming in the future[25]. However, in the meantime, we need to solve it in another way.

One possible workaround is to open the desired connection on our own (or using another library):

```
def proxy_connect(self, target, proxy):
    self._http_conn = http.client.HTTPConnection(proxy[0], proxy[1])
    self._http_conn.set_tunnel(f"{target[0]}:{target[1]}")
    self._http_conn.connect()
```

and then feed the underlying socket to the websockets library when starting the connection.

As for the fuzzer input, we used random data limited to printable characters for a text frame, and all binary characters for a binary frame. The following data generator was used:

```
def data_generator():
    while True:
        yield ''.join(random.choice(string.printable)
                      for _ in range(random.randint(0,
                      ↪  MAX_MSG_SIZE))).encode('utf-8')
        yield random.randbytes(random.randint(0, MAX_MSG_SIZE))
```

The decision logic, whether some unexpected behaviour happened, was based on comparing the sent and received data and manually checking for exceptions from the Burp application, which were configured to be saved into a local directory. No exception related to the testing was spotted.

The implementation of the fuzzing function, as described above, follows:

```
async def ws_fuzz(socket, ws_uri):
    count = 0
    errors = 0

    async with websockets.connect(ws_uri, sock=socket,
    ↪  max_size=MAX_MSG_SIZE, compression=None) as websocket:
        for data in data_generator():
            await websocket.send(data)
            resp = await websocket.recv()

            if data != resp:
```

protocol.

[25]There is an unresolved issue on GitHub [61] and corresponding pull request [64], which does not seem functional.

```
                errors += 1
                print(f"MISMATCH\ndata: {data}\nresp: {resp}\n")
                continue

            count += 1
            if (count % 1000) == 0:
                print(f"INFO: {count} OK; {errors} errors\n")
```

The complete source code can be found on the enclosed media in the file *client.py*.

Using this high-level fuzzing client, we tested over one million requests with payload data lengths between 0 and 2^{15}, and no failure was detected.

3.4.3 Low-level fuzzing

From the previous section, we should be confident that no sequence of bytes in the payload data will cause any harm. Nevertheless, there are more fields to be tested in the frame itself (presented earlier in section 2.2.2.1).

Several frameworks were considered for this part. Boofuz [65, 66], a more actively maintained fork of previously quite popular framework Sully [67], did seem like the best candidate. However, after digging deeper into the details and trying it out for our specific case, we noticed that it is not sufficiently low-level. In several cases, we need to work with individual bits in the frame, and even though the framework does have a primitive `BitField`, it can produce only whole bytes from fields with an unaligned count of bits. To demonstrate it on an example, we used fields from the first byte of the websocket frame:

```
test = Request("Alignment-Test", children=(
    BitField("FIN", default_value=1, width=1),
    BitField("RSV1", default_value=0, width=1),
    BitField("RSV2", default_value=0, width=1),
    BitField("RSV3", default_value=0, width=1),
    BitField("Opcode", default_value=1, width=4)))
```

Calling `test.render()` returns `b'\x01\x00\x00\x00\x01'` instead of a single byte `b'\x81'`. With a few more minor inconveniences, the idea of using a fuzzing framework was rejected. However, we can use at least some parts of the *websockets* library.

First, we implemented a helper class `SimpleWebsocketConnection`[26] that manages the connection – providing public methods `connect`, `close`, `read_frame`, `write_frame`, and `proxy_alive`. The `read_frame` simply calls the library's implementation – `Frame.read` [68]. The `write_frame` method is a bit more tricky, as the library has some sanity checks and understandably does not allow us to create malformed frames. The easiest solution was to

[26] On the enclosed media, it is inside *simple_websocket_connection.py* file.

copy the code of `Frame.write` [69] function and modify it according to our needs. The licence [70] of the library, fortunately, allows us to do this. And the `proxy_alive` is a check if the proxy did not crash. It is useful when we expect that the connection will be terminated (e.g. after sending a frame with one of the reserved bits set), but we still need some confirmation that the proxy is alive.

For simple usage, we also defined an asynchronous context manager:

```
@asynccontextmanager
async def websocket(target, proxy):
    """Context manager for the SimpleWebsocketConnection class.

    Allows to use the syntax:
    async with websocket(...) as ws:
        ...
    """
    ws = SimpleWebsocketConnection(target, proxy)

    try:
        await ws.connect()
        yield ws
    finally:
        await ws.close()
```

The last part of the *simple_websocket_connectin.py* file is a `RawFrame` class for representation of individual fields and containing one method, allowing validation of the whole frame.

The rest of the logic that takes advantage of this prepared infrastructure is contained in the *client_low_level.py* file. First, we define some base `RawFrames`[27]:

```
test_cases = [
    # Text frame
    RawFrame(fin=True, rsv1=False, rsv2=False,
             rsv3=False, opcode=OP_TEXT, mask_flag=True, payload_len=None,
             ↪ mask_key=None, data=b"text"),

    # Binary frame
    RawFrame(fin=True, rsv1=False, rsv2=False,
             rsv3=False, opcode=OP_BINARY, mask_flag=True, payload_len=None,
             ↪ mask_key=None, data=b"binary"),

    # Reserved (non-control frames)
    RawFrame(fin=True, rsv1=False, rsv2=False,
             rsv3=False, opcode=0x3, mask_flag=True, payload_len=None,
             ↪ mask_key=None, data=b"reserved 0x3"),
    [ ... ]
]
```

[27] There is one frame for each opcode.

These base frames then serve as an input for a `mutations` generator, which returns new frames with various mutations of all the fields:

```
def mutations(base_frame):
    """Generator of frames with various mutations from a given base frame.
    """
    # Unchanged
    yield base_frame

    # Switch mask_flag
    mutation = copy.deepcopy(base_frame)
    mutation.mask_flag = not mutation.mask_flag
    yield mutation

    # Switch rsv1 flag
    mutation = copy.deepcopy(base_frame)
    mutation.rsv1 = not mutation.rsv1
    yield mutation

    [...]

    # Modify data and/or payload_len
    for _ in range(100):
        mutation = copy.deepcopy(base_frame)
        mutation.payload_len = random.randint(0, 2**15)

        data_len = random.randint(mutation.payload_len, 2**15)
        if mutation.opcode == OP_TEXT:
            mutation.data = ''.join(random.choice(string.printable)
                                    for _ in
                                    ↪  range(data_len)).encode('utf-8')
        else:
            mutation.data = random.randbytes(data_len)

        yield mutation
```

The payload data from this generator may contain more data than indicated. Sending less data is contra-productive because the server is waiting for the remaining part, leading to a closed connection on timeout error.

Every frame generated this way is then tested using the `test_mutation` function:

```
async def test_mutation(frame):
    async with websocket(TARGET, HTTP_PROXY) as ws:
        await ws.write_frame(frame)

        # Don't expect response for pong
        if frame.opcode == OP_PONG:
            if ws.proxy_alive():
                return
            else:
                raise RuntimeError("Proxy died after pong")
```

```python

        # Read response frame
        for _ in range(3):
            try:
                resp = await asyncio.wait_for(ws.read_frame(),
                ↪ timeout=READ_RESPONSE_TIMEOUT)
            except Exception:
                if frame.is_valid():
                    raise
                elif ws.proxy_alive():
                    return

            if not ws.proxy_alive():
                raise RuntimeError("Proxy is not responding")

            if resp.opcode == OP_PING or (resp.opcode == OP_PONG and
            ↪ frame.opcode != OP_PING):
                click.secho(f"Got unexpected frame: {resp}", fg="yellow")
                click.echo("Reading again\n")
                continue
            else:
                break

        if (not frame.is_valid() or frame.opcode == OP_CLOSE) and
        ↪ resp.opcode == OP_CLOSE:
            return

        if resp.opcode not in [OP_CONT, OP_TEXT, OP_BINARY, OP_PONG]:
            raise ValueError(f"Response contains bad opcode: {resp}")

        # Expecting echo server on the other side sending the same data back
        if frame.opcode in [OP_TEXT, OP_BINARY, OP_PONG] and frame.data !=
        ↪ resp.data:
            # If we faked payload_len, compare only the shorter length
            incorrect_len = frame.payload_len is not None and
            ↪ frame.payload_len < len(
                frame.data)
            if incorrect_len and frame.data[:frame.payload_len] ==
            ↪ resp.data[:frame.payload_len]:
                return
            raise ValueError(f"Data missmatch: {resp}")
```

If any unexpected situation happens[28], this frame is reported as suspicious and should be examined manually.

Even in this low-level testing of the WebSocket protocol implementation, we did not find any bugs or vulnerabilities. Overall, the tested part of the application seems reliable. We used plenty of different messages, and none of the responses raised any suspicion.

[28] E.g. returning different data than we have sent for text/binary/ping frames, or getting a timeout on the connection.

If anyone is conducting similar fuzzing activity in the future, they might want to include one additional test-case. We limited ourselves to operating with single frames. But the protocol allows sending a message decomposed into several frames, using the `FIN` flag (unset if another frame will follow), and opcode `0x0` – continuation frame (from the second frame to the final one). This additional test-case could use our prepared infrastructure for building, sending, and receiving frames.

3.5 Examine Burp's communication

3.5.1 Motivation

In this section, we would like to investigate the Burp's background communication. Not the usual traffic expected from a proxy. We are interested in messages like performance feedback to the company's servers, or other suspicious outbound traffic. Although the company claims that all the user's data are anonymized, it is always better to verify such claims, even though this analysis might be a tough challenge.

Moreover, recent security incidents are teaching us a lesson not to underestimate supply-chain attacks. An example is a recent incident with SolarWinds Orion, described in brief further.

3.5.1.1 SolarWinds Orion incident

SolarWinds Orion is a suite of products helping companies with IT management. It provides a huge range of operations, including various monitoring capabilities, network configuration, and virtualization. [71]

In December 2020, a famous cybersecurity company Fireeye stated that their infrastructure was compromised using highly sophisticated offensive capabilities. The entry point was presumably an Orion platform by SolarWinds. Later, SolarWinds company said that 18000 of their customers might have been affected by this supply-chain attack. [72, 73]

The attackers managed to inject their code into the product's codebase while doing various measures not to be detected. Even there, they spent a considerable effort to remain hidden. There might have been a second actor taking advantage of a zero-day vulnerability CVE-2020-10148 in SolarWinds Orion API, allowing to execute unauthenticated API calls. [73, 74]

Some news suggested that the hackers might have gained access to *some* systems of high-value targets such as the US Energy Department, and the National Nuclear Security Administration. For example, The Guardian claims "*The attackers gained access to an extraordinary array of potential targets in the US alone: more than 425 of the Fortune 500 list of top companies; all of the top 10 telecommunications companies; all five branches of the military; and all of the top five accounting firms.*" [75]

Recent update from Microsoft serves as another confirmation of this story. Microsoft, similarly to other companies, detected presence of malicious SolarWinds applications in their environment, and said "*We detected unusual activity with a small number of internal accounts and upon review, we discovered one account had been used to view source code in a number of source code repositories.*" [76]

We should not draw preliminary conclusions while the investigation is ongoing. However, the so far uncovered information shows the severity of such incidents and that nobody should take security lightly. The analysis executed in this section could potentially discover if the application was malicious, and the approach developed to inspect the network traffic (3.5.2–3.5.5) can be used in any future security audit.

3.5.2 Prepare a network namespace

Right at the beginning of the inspection of Burp's background communication, we have identified two problems that would need to be solved. The first one is simpler – we need to capture all the traffic coming out of Burp, but preferably without too much noise from other applications. We could set an upstream proxy in Burp and monitor it from there, but that would not guarantee that no further connection was made or the application behaviour did not change. Filtering the relevant packets from a whole local network traffic dump is also not an ideal solution, since there is no easy way to describe the Burp's connections. The other challenge is that the traffic is most probably encrypted using a TLS layer.

An elegant approach to separate an application from the rest of the system is by using namespaces[29]. In our case, we only need to separate the network part; thus, using a network namespace will be sufficient. In the past, a Linux kernel shared one space across the entire OS for all the network interfaces and routing tables. This concept has changed with the introduction of network namespaces, allowing several separate networks to operate independently of each other. [78]

Note that some of the commands used in this section need to run with root privileges.

A new `net` namespace with name *burp_ns* can be created with:

```
# ip netns add burp_ns
```

To check all available (named) network namespaces on the system:

```
$ ip netns list
```

[29]Containers are using namespaces as the underlying technology (together with cgroups, and UnionFS in case of Docker) [77].

In our case, we see `burp_ns (id: 0)` in the output.

Although we have created a network namespace, at the moment, it is not much useful since we cannot even ping to the localhost address from within. Running any command from inside of the `burp_ns` namespace is possible with a command `ip netns exec burp_ns <command>`.

First, we can fix the ping to the localhost address by bringing up a loopback (`lo`) interface:

```
# ip netns exec burp_ns ip link set lo up
```

The ping command should start working (only for localhost):

```
# ip netns exec burp_ns ping localhost
```

The next task is to bring inside the Internet connection. There are several things we need to configure. First, let us create a *veth*[30] pair. To create such pair with names `veth-h` (on the host), and `veth-burp` (inside of the namespace), run:

```
# ip link add veth-h type veth peer name veth-burp
```

To insert the `veth-burp` into the namespace, run:

```
# ip link set veth-burp netns burp_ns
```

Next, assign private IPv4 addresses to the veth devices:

```
# ip addr add 10.9.8.1/24 dev veth-h
# ip netns exec burp_ns ip addr add 10.9.8.2/24 dev veth-burp
```

and bring the devices up:

```
# ip link set veth-h up
# ip netns exec burp_ns ip link set veth-burp up
```

At this stage, ping between the endpoints should be possible:

```
$ ping 10.9.8.2
# ip netns exec burp_ns ping 10.9.8.1
```

To get access to the Internet, we will need to enable packet forwarding and start using NAT. To verify if packet forwarding is enabled on the system, run:

```
$ sysctl net.ipv4.ip_forward
```

[30] Veth is a virtual ethernet interface.

The value needs to be 1. Otherwise run:

```
# sysctl -w net.ipv4.ip_forward=1
```

for a temporary assignment or use an appropriate `sysctl` configuration file. Then allow the forwarding of packets, e.g. with `iptables` utility:

```
# iptables -A FORWARD -o enp0s31f6 -i veth-h -j ACCEPT
# iptables -A FORWARD -i enp0s31f6 -o veth-h -j ACCEPT
```

where *enp0s31f6* is our ethernet interface. And finally, start the masquerade:

```
# iptables -t nat -A POSTROUTING -s 10.9.8.2/24 -o enp0s31f6 -j MASQUERADE
```

Sometimes, it might also be needed to add additional firewall rules (e.g. include the veth-h into a trusted zone).

The last missing piece is routing. We can fix it by specifying a default route inside the namespace:

```
# ip netns exec burp_ns ip route add default via 10.9.8.1
```

Finally, there is Internet access in the namespace.

3.5.3 Breaking the TLS

The next big issue to be solved is that nowadays most of the traffic is encrypted with TLS. Some applications like Chrome, Firefox, and cURL allow saving the per-session secrets to a local log file (configured with an environment variable `SSLKEYLOGFILE`) [79]. However, there is no such option in Burp. [80]

Luckily, a project *extract-tls-secrets* [81] exists. If attached to a Java application on either side of the connection, it allows making similar action (dump the per-session secrets into a log file). The attachment is made in the following manner:

```
$ java \
  -javaagent:<path>/extract-tls-secrets-4.0.0.jar=/<out_path>/secrets.log \
  -jar app.jar
```

Then we only need to point Wireshark to the secrets log file, which can afterwards achieve automatic on-the-fly decryption of the TLS. This can be configured in *Preferences – Protocols – TLS (SSL for older versions) – (Pre)-Master-Secret log filename.*

3.5.4 Record Burp's traffic

We already prepared a `burp_ns` network namespace and found out how to decrypt the TLS traffic sent from Burp. Now, it is time to start recording the traffic.

First, start Wireshark, so no traffic is missed during the tested application launch:

```
# ip netns exec burp_ns wireshark
```

We can start recording on the *any* interface since we are inside of the namespace. Now, we can finally launch Burp. There is a problem with detecting the user's licence because the command `ip netns exec burp_ns` needs to run with root privileges (we used `sudo` utility in front of the command). The workaround is to specify the user for the next part of the command with another `sudo` call. The launch command is:

```
# ip netns exec burp_ns sudo -u tom \
  /home/tom/BurpSuitePro/jre/bin/java \
    -javaagent:<path>/extract-tls-secrets-4.0.0.jar=/tmp/secrets.log \
    -jar <path>/burpsuite_pro_v2020.12.1.jar
```

We let the application run for some time and then started analyzing the captured traffic.

3.5.5 Analyze the captured traffic

Fortunately, the TLS decryption is working as expected, so we can see what Burp is sending outside. Note that we did not list below messages originating from extensions (e.g. downloading new string patters for finding vulnerable software versions).

3.5.5.1 Check for updates

The first connection made from the application right after launch is a check for updates. The corresponding packets in Wireshark can be seen in Figure 3.14. The extracted HTTP content follows.

Request:

```
GET /Burp/Releases/CheckForUpdates?product=pro&version=2020.12.1&license=<censored_license_key> HTTP/1.1
Host: portswigger.net
Accept: */*
Accept-Language: en
Connection: close
```

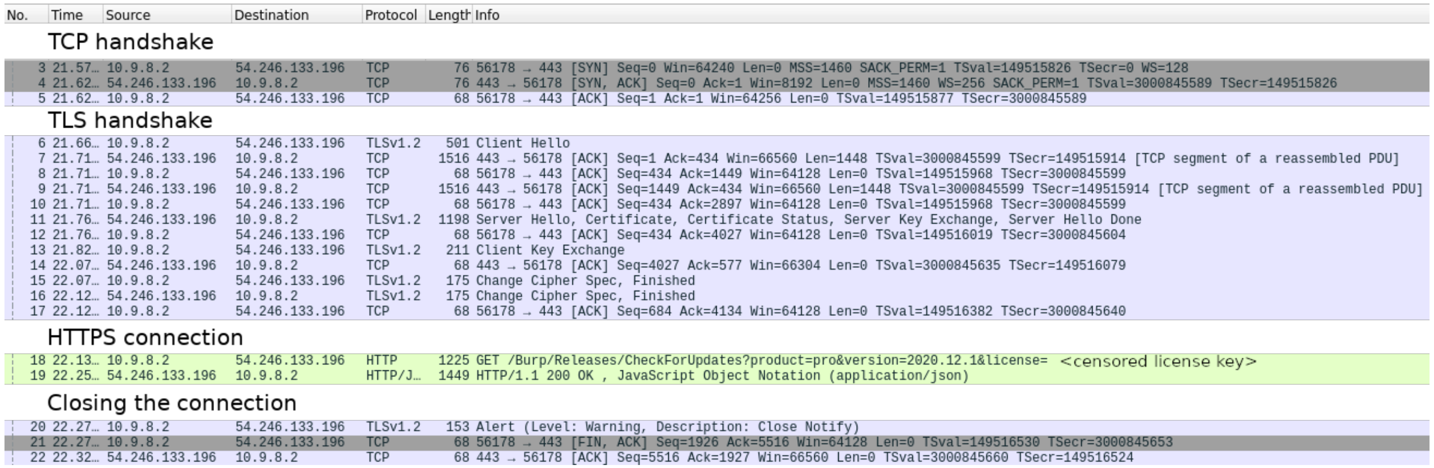

Figure 3.14: Wireshark – check for updates

Response:

```
HTTP/1.1 200 OK
Cache-Control: private, s-maxage=0,no-store, no-cache
Content-Type: application/json; charset=utf-8
Content-Security-Policy: default-src 'none';base-uri 'none';child-src 'self'
↪ https://www.youtube.com/embed/;connect-src 'self'
↪ https://www.google-analytics.com/collect
↪ https://www.google-analytics.com/r/collect
↪ https://www.google-analytics.com/j/collect
↪ https://www.googletagmanager.com
↪ https://www.google.com/recaptcha/;font-src 'self';frame-src 'self'
↪ https://www.youtube.com/embed/ https://www.google.com/recaptcha/;img-src
↪ 'self' data:;media-src 'self' https://d21v5rjx8s17cr.cloudfront.net/
↪ https://d2gl1b374o3yzk.cloudfront.net/;script-src 'self'
↪ 'nonce-6uAs5bHTLOktc8CP96OD8VbbtFdw/VRe' 'strict-dynamic';style-src
↪ 'self';
Set-Cookie: SessionId=B15461[snipped]; domain=portswigger.net; expires=Sat,
↪ 22-Dec-2040 13:14:43 GMT; path=/; secure; HttpOnly; SameSite=Lax
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000; preload
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Date: Sun, 27 Dec 2020 13:14:42 GMT
Connection: close
Content-Length: 127

{"result":"up_to_date","licenseId":"<censored
↪ ID>","manualDownloadUrl":"","autoDownloadUrl":"","updates":[]}
```

We can notice that the request contains the license key, sent as a GET parameter. The response was that the software is up-to-date.

3.5.5.2 BApp Store current list

The next message sent soon after the application started, is a request for current BApp Store list of extensions:

```
GET /bappstore/currentlist HTTP/1.0
Host: portswigger.net
Accept-Encoding: gzip, deflate
```

The response contains many base-64 encoded strings, where each (except one) correspond to one extension in the store. The whole response is available on the enclosed media in file `bappstore_currentlist.txt`. Example of a decoded entry follows:

```
Type: 1000
Version: 0
Uuid: e2a137ad44984ccb908375fa5b2c618d
ExtensionType: 1
```

```
Name: .NET Beautifier
ScreenVersion: 0.3
SerialVersion: 3
MinPlatformVersion: 0
ProOnly: False
Author: Nadeem Douba
Description: <base64-encoded description>
Rating: 464
Revoked: False
DownloadUrl: https://portswigger.net/bappstore/bapps/download/e2a137ad44984ccb908375fa5b2c618d
ExecutionScore: 8255
LastUpdated: 1485173990550
RepoUrl: https://github.com/portswigger/dotnet-beautifier
```

We decoded and inspected all of the entries received. Except for the last one, they all describe extensions. The last entry contains 128 bytes of binary data, which we were unable to understand. A similar entry also appears in other messages. A hypothesis is that it might be a signature of the message.

3.5.5.3 Burp Collaborator polling

The limitation of today's Internet is the lack of public IPv4 addresses. Burp Collaborator is a public service (or you can run it on your infrastructure) that mitigates this shortage and allows testing of external service interaction even from behind of NAT.

The Collaborator server implements several network protocols (such as DNS, HTTP, and SMTP) and lets you know when it receives any request. Burp is configured to use `burpcollaborator.net` by default, and there is a polling every 10 seconds to check if any interaction happened. [82]

Request:

```
GET /burpresults?biid=GUHoCPoG0xgbgnB5DiVNLc3wkHu360Np2wMDfw33dws%3d HTTP/1.1
Host: polling.burpcollaborator.net
Accept: */*
Accept-Language: en
Connection: close
```

Response:

```
HTTP/1.1 200 OK
Server: Burp Collaborator https://burpcollaborator.net/
X-Collaborator-Version: 4
X-Collaborator-Time: 1609074943031
Content-Type: application/json
Content-Length: 2

{}
```

To see how a notification about executed interaction looks like, we generated a unique link from Burp's Collaborator client and made a DNS query for that address (with a `dig` command-line utility). This time, the HTTP response data were:


```
{
  "responses":[
    {
      "protocol":"dns",
      "opCode":"1",
      "interactionString":"kl9xd7p9i31wrxvnb3semtwvrmxcl1",
      "clientPart":"0y",
      "data":{
        "subDomain":"Kl9XD7P9i31wrXVnB3SeMTwVRMXcL1.BUrpCOLLaboRatOR.NeT",
        "type":1,
        "rawRequest":"KdgAEAABAAAAAAABHktsOVhEN1A5aTMxd3JYVm5CM1NlTVR3VlJNW
        ↪ GNMMRBCVXJwQ09MTGFib1JhdE9SA05lVAAAAQABAAApBNAAAIAAAAA="
      },
      "time":"1609103932712",
      "client":"<ip_address>"
    }
  ]
}
```


3.5.5.4 Performance anonymous feedback

Burp application is sending another message outside, to the company's servers – anonymized performance feedback. If we have a look at the request (at least its printable representation), it seems a bit strange:


```
POST /feedback/submit HTTP/1.0
Content-Length: 566

PK.........q.Q................ipc-envelope.....@...>..T..!j.*........#.C..2
↪  .DE...YP.e.UWwuw}B..i..C40.?..2....Qrp..a.<.F......P.C...}.lg.@.W.....>
↪  ....S.@.....2..Q*..........y..........tCkU2....=^.{>.w...s..?M.C:...#..
↪  .2.^^I.....E.E.\...%"..q.gS.F./......PE#....N.b....?h&...F.T.....f..QD8
↪  0.&..b....
YO...E.
..A.....e.@4S...ncs..T.........@>..F_..:....&.g...VV....?..he.ho.v...F.&...
↪  ..
.G.Y.HrXz..).&Q...v"u......."....s}V.:\...6..#4u..6.y5.=.....?.sV.K..)?jD.
↪  ?.....[4..~.PK..,...........PK..........q.Q,.........................
↪  ...ipc-envelopePK.........:.........
```


The exact data sent can be seen in appendix B.

For some time, we were struggling to understand the meaning of these binary data. But then we realized that it is a ZIP archive, from which we can extract a single file named "ipc-envelope".

An example of the ipc-envelope file content follows:

```
Version: 0
Channel: stable
ProductExecutionMode: 1
ProductType: Pro
ProductVersion: 2020.12.1-5278
UniqueIdentifier: <censored id>

```

```
VHlwZTogNTAwMApWZXJzaW9uOiAwCkZFQVRVUkVfVFlQRTogVEFSR0VUX1NJVEVfTUFQX1ZJRVdfT1BUSU9OX0xFRlRfUklHSFRfU1BMSVQKVFlQRTogRkVBVFVSRV9VU0UKCg==
VHlwZTogNTAwMApWZXJzaW9uOiAwClFVQU5USVRZX1RZUEU6IFBST1hZX0hJU1RPUllfRURJVEVEX1ZJRVdfRklMVEVSX1RJTUVSClRZUEU6IFFVQU5USVRZCgowCg==
VHlwZTogNTAwMApWZXJzaW9uOiAwClFVQU5USVRZX1RZUEU6IFBST1hZX0hJU1RPUllfRURJVEVEX1ZJRVdfRklMVEVSX1RJTUVSClRZUEU6IFFVQU5USVRZCgowCg==
F/Ovx8HRzj4JmcgRhXTSeZfgzeuhpadoand04GEFUzpdKATjAx8c6ssVjUlRxTePUgScheFODZTUeWkE6fglQ42WFwqnlorHoYLf0diUcqF/HYlGwdBODYzgbNf09XyhKiUejCaN1qPvqxrsqYkqSbnc8iTP2Few3+G5qDTERdI=
```


The four lines at the end of the file seem[31] to be base64 encoded. If we decode them, the first entry contains the following text:

```
Type: 5000
Version: 0
FEATURE_TYPE: TARGET_SITE_MAP_VIEW_OPTION_LEFT_RIGHT_SPLIT
TYPE: FEATURE_USE
```

The second and third entries are the same, and decoded contain:

```
Type: 5000
Version: 0
QUANTITY_TYPE: PROXY_HISTORY_EDITED_VIEW_FILTER_TIMER
TYPE: QUANTITY

0
```

However, the last line is significantly different – contains 128 bytes of binary data, which we could not decode/decrypt. It is a similar case as with the BApp Store current list's last entry.

After examining more of the feedback messages, we can say that they all follow the same pattern. Some of them may have a significantly higher count of the base64-encoded entries (e.g. the final one sent during a shutdown), but the text entries are similar. The one last entry always differs and contains some binary data.

3.5.5.5 Background traffic summary

During the traffic inspection, we managed to identify and understand the format of several messages:

[31] Because of the character set used and the typical '=' characters at the end.

- Check for updates
- BApp store current list
- Burp Collaborator polling
- Performance anonymous feedback

Except for the mysterious last entry (128 bytes of binary data) of the BApp store current list, and performance feedback messages, we did not notice any sensitive or suspicious data sent outside (or received) by the application. However, we cannot guarantee that Burp does not send any additional messages, especially in longer time window or if different conditions occur. In our case, the network dump covered two hours, during which we generated some light activity like opening a website from a web browser or starting a scan with configured credentials. The binary data might be a signature, adding another layer of security.

Conclusion

This thesis's first two goals were to study the current state of the Burp Suite penetration testing tool or Owasp ZAP penetration testing tool (at least one of them) and describe its functionality. Chapter 1 contains the description of the penetration testing tool of our choice – Burp Suite, the way it is placed in the man-in-the-middle position, what issues it brings (e.g. broken TLS) and how to address them (use the Burp's certificate). Then follows a description of several most widely used features.

The third goal was to manually examine security aspects of the application, look for weak spots and focus on new functionalities of the application (e.g. WebSockets). The manual examination was done in the first part of chapter 3 (sections 3.1, 3.2, and 3.3). First, we identified the endpoints of the web interface and the REST API. During the process, we examined all the parts of the application that looked suspicious. Next, we started automated scans of the web interface. And last, we manually processed and verified findings from these scans. Although no severe flaw was detected in the application, we managed to show that not everything is as it should be. E.g. the webserver is not compliant with the RFC 2616 (Hypertext Transfer Protocol – HTTP/1.1), or an undocumented REST API call, allowing to shut down the application, was discovered. All these discrepancies will be reported to the Portswigger company for their own appraisal.

Continuing in the third goal and moving towards the fourth one, we turned our attention towards WebSockets. Chapter 2 described the theory of this protocol. That includes protocol overview, a handshake for both – clients and servers, and data transfer explanation with all the pieces like framing and masking. In section 3.4, we describe our approach writing fuzzing applications for the WebSockets component. We ended up with three applications. The first one is a WebSocket server for keeping all this testing locally in an environment controlled by us. The second one is a high-level fuzzer that tests the data payload being transferred over the WebSocket connection. The third one goes deeper and breaks down the WebSocket frame into individual pieces and operates on them. In the final section 3.5, we inspected the Burp's communication. We succeeded in separating the Burp's traffic from the rest of the system and breaking into the TLS layer, and we manually examined the content of the messages.

The final goal was to discuss and analyze the results with a focus on their security aspects. This was achieved throughout the text, following the description of individual findings. Overall, we did not discover any major security flaw of the components tested or any sensitive information leakage, and the implementation of WebSockets seems reliable. However, we found several minor issues.

Future work might add support for sequences of frames into the fuzzer application, analyze an up-to-date traffic dump using our methodology, and audit other Burp's components.

Bibliography

[1] Oyetoyan, T. D.; Milosheska, B.; et al. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In *Agile Processes in Software Engineering and Extreme Programming*, edited by J. Garbajosa; X. Wang; A. Aguiar, Lecture Notes in Business Information Processing, Springer International Publishing, ISBN 978-3-319-91602-6, pp. 86–103, doi:10.1007/978-3-319-91602-6_6.

[2] Positive Technologies. SAST, DAST, IAST, and RASP [online]. Available from: `https://www.ptsecurity.com/ww-en/analytics/knowledge-base/sast-dast-iast-and-rasp-how-to-choose/` [accessed 2020-10-04]

[3] Whitehorn-Gillam, M. What Tools Are Used When Penetration Testing a Web Application [online]. Available from: `https://secureideas.com/knowledge/what-tools-are-used-when-penetration-testing-a-web-app` [accessed 2020-12-18]

[4] Luotonen, A.; Altis, K. World-Wide Web Proxies. volume 27, no. 2: pp. 147–154, ISSN 01697552, doi:10.1016/0169-7552(94)90128-7. Available from: `https://linkinghub.elsevier.com/retrieve/pii/0169755294901287` [accessed 2020-09-20]

[5] PortSwigger Ltd. Features – Burp Suite Professional [online]. Available from: `https://portswigger.net/burp/pro/features` [accessed 2020-09-20]

[6] OWASP Foundation, Inc. OWASP ZAP [online]. Available from: `https://www.zaproxy.org/docs/desktop/start/features/` [accessed 2020-12-18]

[7] PortSwigger Ltd. Web Application Security, Testing, & Scanning [online]. Available from: `https://portswigger.net` [accessed 2020-09-20]

[8] OWASP Foundation, Inc. Zaproxy [online]. Available from: `https://github.com/zaproxy/zaproxy` [accessed 2020-10-02]

[9] PortSwigger Ltd. Launching Burp Suite from the Command Line [online]. Available from: `https://portswigger.net/burp/documentation/desktop/getting-started/launching/command-line` [accessed 2020-10-15]

[10] Mitchell, J. C. Portability and Safety: Java. In *Concepts in Programming Languages*, Cambridge University Press, ISBN 978-0-521-78098-8, pp. 384–428, doi:10.1017/CBO9780511804175.014. Available from: `https://www.cambridge.org/core/books/concepts-in-programming-languages/portability-and-safety-java/9950A3084F8D8F2CFA1D21AEF4E79E45`

[11] PortSwigger Ltd. Professional / Community 2020.9.2 [online]. Available from: `https://portswigger.net/burp/releases/professional-community-2020-9-2` [accessed 2020-10-15]

[12] PortSwigger Ltd. Burp Proxy Options [online]. Available from: `https://portswigger.net/burp/documentation/desktop/tools/proxy/options` [accessed 2020-10-16]

[13] Walfield, N. H.; Koch, W. TOFU for OpenPGP. In *Proceedings of the 9th European Workshop on System Security - EuroSec '16*, ACM Press, ISBN 978-1-4503-4295-7, pp. 1–6, doi:10.1145/2905760.2905761. Available from: `http://dl.acm.org/citation.cfm?doid=2905760.2905761` [accessed 2020-10-12]

[14] Google Inc. HSTS Preload List Submission [online]. Available from: `https://hstspreload.org/` [accessed 2020-10-12]

[15] Jackson, C.; Barth, A.; et al. HTTP Strict Transport Security (HSTS) [online]. Available from: `https://tools.ietf.org/html/rfc6797` [accessed 2020-10-12]

[16] Fielding, R.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. , no. RFC 2616: pp. 61–62, ISSN 2070-1721. Available from: `https://www.rfc-editor.org/info/rfc2616` [accessed 2020-12-20]

[17] Apache Software Foundation. RedirectSSL - HTTPD [online]. Available from: `https://cwiki.apache.org/confluence/display/HTTPD/RedirectSSL` [accessed 2020-10-16]

[18] Mauro, T. How to Create NGINX Rewrite Rules [online]. Available from: `https://www.nginx.com/blog/creating-nginx-rewrite-rules/` [accessed 2020-10-16]

[19] W3Schools. How To Redirect to Another Webpage [online]. Available from: https://www.w3schools.com/howto/howto_js_redirect_webpage.asp [accessed 2020-12-18]

[20] Dierks, T.; Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.1. , no. RFC 4346, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc4346 [accessed 2020-12-22]

[21] PortSwigger Ltd. Intercepting Messages [online]. Available from: https://portswigger.net/burp/documentation/desktop/tools/proxy/intercept [accessed 2020-12-18]

[22] OWASP Foundation, Inc. OWASP Top Ten Web Application Security Risks [online]. Available from: https://owasp.org/www-project-top-ten/ [accessed 2020-10-17]

[23] PortSwigger Ltd. Burp Proxy History [online]. Available from: https://portswigger.net/burp/documentation/desktop/tools/proxy/history [accessed 2020-12-18]

[24] PortSwigger Ltd. Audit Options [online]. Available from: https://portswigger.net/burp/documentation/desktop/scanning/audit-options [accessed 2020-12-18]

[25] DVWA team. DVWA – Damn Vulnerable Web Application [online]. Available from: http://www.dvwa.co.uk/ [accessed 2020-10-18]

[26] PortSwigger Ltd. Burp Suite Extensibility [online]. Available from: https://portswigger.net/burp/extender [accessed 2020-12-18]

[27] Fielding, R.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. , no. RFC 2616: pp. 12–14, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc2616 [accessed 2020-12-20]

[28] Wilkins, G.; Salsano, S.; et al. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP [online]. Available from: https://tools.ietf.org/html/rfc6202 [accessed 2020-10-29]

[29] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: pp. 4, 11, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[30] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: p. 14, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[31] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: pp. 5–9, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[32] McManus, P. Bootstrapping WebSockets with HTTP/2 [online]. Available from: https://tools.ietf.org/html/rfc8441 [accessed 2020-12-19]

[33] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: pp. 4–9, 14–25, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[34] Fielding, R.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. , no. RFC 2616: p. 39, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc2616 [accessed 2020-12-20]

[35] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: pp. 27–39, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[36] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: pp. 36–38, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[37] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: p. 38, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[38] Fette, I.; Melnikov, A. The WebSocket Protocol. , no. RFC 6455: pp. 32–33, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc6455 [accessed 2020-12-19]

[39] IANA. Internet Assigned Numbers Authority [online]. Available from: https://www.iana.org/ [accessed 2020-12-25]

[40] Mozilla Foundation. Using HTTP Cookies [online]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies [accessed 2020-12-20]

[41] Mozilla Foundation. Content Security Policy (CSP) [online]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP [accessed 2020-12-20]

[42] OWASP Foundation, Inc. Cross Site Scripting (XSS) Software Attack [online]. Available from: https://owasp.org/www-community/attacks/xss/ [accessed 2020-12-20]

[43] Grossman, J. Cross-Site Scripting Worms and Viruses: The Impending Threat and the Best Defense [online]. Available from: https://web.archive.org/web/20110104191201/http://net-security.org/dl/articles/WHXSSThreats.pdf [accessed 2020-12-20]

[44] Kamkar, S. MySpace Worm Explanation [online]. Available from: https://samy.pl/myspace/tech.html [accessed 2020-12-20]

[45] PortSwigger Ltd. What Is DOM-Based XSS (Cross-Site Scripting) [online]. Available from: https://portswigger.net/web-security/cross-site-scripting/dom-based [accessed 2020-12-20]

[46] OWASP Foundation, Inc. Cross Site Scripting Prevention – OWASP Cheat Sheet Series [online]. Available from: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html [accessed 2020-12-21]

[47] Rogaway, P. Nonce-Based Symmetric Encryption. In *Fast Software Encryption*, edited by B. Roy; W. Meier, Lecture Notes in Computer Science, Springer, ISBN 978-3-540-25937-4, pp. 348–358, doi:10.1007/978-3-540-25937-4_22.

[48] Haywood, S.; Skelton, M.; et al. Ffuf [online]. Available from: https://github.com/ffuf/ffuf [accessed 2020-12-21]

[49] Burns, W. J. Common Password List (Rockyou.Txt) [online]. Available from: https://kaggle.com/wjburns/common-password-list-rockyoutxt [accessed 2020-12-21]

[50] Cubrilovic, N. RockYou Hack: From Bad To Worse [online]. Available from: https://social.techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/ [accessed 2020-12-21]

[51] The Linux Foundation. OpenApi Initiative [online]. Available from: https://www.openapis.org/ [accessed 2020-11-29]

[52] g0tmi1k. Nikto [online]. Available from: https://tools.kali.org/information-gathering/nikto [accessed 2020-12-23]

[53] Sullo, C. Nikto v2.1.5 – The Manual [online]. Available from: https://cirt.net/nikto2-docs/ [accessed 2020-12-23]

[54] Mozilla Foundation. X-XSS-Protection [online]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection [accessed 2020-12-23]

[55] Mozilla Foundation. Allow [online]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Allow [accessed 2020-12-24]

[56] Fielding, R.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. , no. RFC 2616, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc2616 [accessed 2020-12-20]

[57] Fielding, R.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. , no. RFC 2616: pp. 128–129, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc2616 [accessed 2020-12-20]

[58] Fielding, R.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. , no. RFC 2616: pp. 32–33, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc2616 [accessed 2020-12-20]

[59] Mozilla Foundation. Expect [online]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Expect [accessed 2020-12-04]

[60] Fielding, R.; Gettys, J.; et al. Hypertext Transfer Protocol – HTTP/1.1. , no. RFC 2616: pp. 48–50, ISSN 2070-1721. Available from: https://www.rfc-editor.org/info/rfc2616 [accessed 2020-12-20]

[61] Martelli, A. *Python in a Nutshell*. "O'Reilly Media, Inc.", ISBN 978-0-596-10046-9, pjqbAgAAQBAJ.

[62] Python Software Foundation. PyPI · The Python Package Index [online]. Available from: https://pypi.org/ [accessed 2020-12-15]

[63] Augustin, A. Websockets: An Implementation of the WebSocket Protocol (RFC 6455 & 7692) [online]. Available from: https://github.com/aaugustin/websockets [accessed 2020-12-15]

[64] Shadura, A. HTTP Proxy Support Reworked by Andrewshadura · Pull Request #751 · Aaugustin/Websockets [online]. Available from: https://github.com/aaugustin/websockets/pull/751 [accessed 2020-12-15]

[65] Amini, P.; Portnoy, A.; et al. Boofuzz: Network Protocol Fuzzing for Humans [online]. Available from: https://boofuzz.readthedocs.io/en/stable/ [accessed 2020-12-16]

[66] Pereyda, J. Jtpereyda/Boofuzz [online]. Available from: https://github.com/jtpereyda/boofuzz [accessed 2020-12-16]

[67] Amini, P.; Portnoy, A.; et al. OpenRCE/Sulley [online]. Available from: https://github.com/OpenRCE/sulley [accessed 2020-12-16]

[68] Augustin, A. Websockets – Frame.read [online]. Available from: https://github.com/aaugustin/websockets/blob/139085fe2624192a5a6c72b1e5db211dcec6ced1/src/websockets/framing.py#L79 [accessed 2020-12-17]

[69] Augustin, A. WebSockets – Frame.write [online]. Available from: https://github.com/aaugustin/websockets/blob/139085fe2624192a5a6c72b1e5db211dcec6ced1/src/websockets/framing.py#L148 [accessed 2020-12-17]

[70] Augustin, A. WebSockets – Licence [online]. Available from: https://github.com/aaugustin/websockets/blob/8.1/LICENSE [accessed 2020-12-17]

[71] SolarWinds Worldwide, LLC. Orion Platform [online]. Available from: https://www.solarwinds.com/solutions/orion [accessed 2020-12-30]

[72] FireEye, Inc. Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor [online]. Available from: https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html [accessed 2020-12-30]

[73] Lakshmanan, R. New Evidence Suggests SolarWinds' Codebase Was Hacked to Inject Backdoor [online]. Available from: https://thehackernews.com/2020/12/new-evidence-suggests-solarwinds.html [accessed 2020-12-30]

[74] Lakshmanan, R. A New SolarWinds Flaw Likely Had Let Hackers Install SUPERNOVA Malware [online]. Available from: https://thehackernews.com/2020/12/a-new-solarwinds-flaw-likely-had-let.html [accessed 2020-12-30]

[75] Hern, A. Orion Hack Exposed Vast Number of Targets – Impact May Not Be Known for a While. Available from: http://www.theguardian.com/world/2020/dec/14/solarwinds-breach-orion-hacked-cyber-espionage [accessed 2020-12-30]

[76] Microsoft Security Response Center. Microsoft Internal Solorigate Investigation Update [online]. Available from: https://msrc-blog.microsoft.com/2020/12/31/microsoft-internal-solorigate-investigation-update/ [accessed 2021-01-02]

[77] Docker Inc. Docker Overview [online]. Available from: https://docs.docker.com/get-started/overview/ [accessed 2021-01-03]

[78] Lowe, S. Introducing Linux Network Namespaces [online]. Available from: https://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/ [accessed 2020-12-28]

[79] Wireshark Foundation. TLS · Wiki [online]. Available from: https://gitlab.com/wireshark/wireshark/-/wikis/TLS [accessed 2020-12-27]

[80] PortSwigger Ltd. Integrating Burp and Wireshark – Burp Suite User Forum [online]. Available from: https://forum.portswigger.net/thread/integrating-burp-and-wireshark-816abb78 [accessed 2020-12-27]

[81] Neykov, S. Extract-Tls-Secrets [online]. Available from: https://github.com/neykov/extract-tls-secrets [accessed 2020-12-29]

[82] PortSwigger Ltd. Burp Collaborator [online]. Available from: https://portswigger.net/burp/documentation/collaborator [accessed 2020-12-27]

Appendix A

Acronyms

API Application Programming Interface

CA Certificate Authority

CRLF Carriage Return (`U+000D`) Line Feed (`U+000A`)

DAST Dynamic Application Security Testing

DVWA Damn Vulnerable Web Application

GUI Graphic User Interface

GUID Globally Unique Identifier

HSTS HTTP Strict Transport Security

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IANA Internet Assigned Numbers Authority

IAST Interactive Application Security Testing

JS JavaScript

OS Operating System

RCE Remote Code Execution

SAST Static Application Security Testing

SP Space character (U+0020)

TLS Transport Layer Security

URI Uniform Resource Identifier

URL Uniform Resource Locator

XSS Cross-Site Scripting

Appendix B

Performance feedback


```
0000    50 4b 03 04 14 00 08 08 08 00 fa 71 9b 51 00 00    PK.........q.Q..
0010    00 00 00 00 00 00 00 00 00 00 0c 00 00 00 69 70    ..............ip
0020    63 2d 65 6e 76 65 6c 6f 70 65 cd 8e cb b2 9a 40    c-envelope.....@
0030    18 84 f7 3e 85 fb 54 ce 99 21 6a 90 2a 17 8a 8c    ...>..T..!j.*...
0040    8a f1 c2 cc fc 23 b2 43 18 2e 32 07 44 45 90 a7    .....#.C..2.DE..
0050    0f 59 50 d9 65 9d 55 57 77 75 77 7d 42 de 1f 69    .YP.e.UWwuw}B..i
0060    91 1b 43 34 30 13 3f cf a5 32 86 8f a7 7f 51 72    ..C40.?..2....Qr
0070    70 bc 17 61 15 3c ad 46 06 d5 b3 eb ec 8a 50 1a    p..a.<.F......P.
0080    43 dc e7 fc 7d eb 6c 67 fa 40 f4 57 1a d2 d0 07    C...}.lg.@.W....
0090    d6 3e f0 f7 b1 f6 53 1f 40 9e 96 95 dc 84 32 7f    .>....S.@.....2.
00a0    a6 51 2a ef c6 f0 9d bd f5 1b f2 15 96 79 1a be    .Q*..........y..
00b0    f2 f2 19 ab cc b8 8c be 74 43 6b 55 32 18 88 b5    ........tCkU2...
00c0    aa 3d 5e c4 7b 3e af 77 f3 db c9 73 ed d6 3f 4d    .=^.{>.w...s..?M
00d0    ab 43 3a af cd cc 23 8e a0 02 32 11 09 a2 1c da    .C:...#...2.....
00e0    f5 84 45 18 45 02 5c bc b7 85 25 22 0e c4 71 b1    ..E.E.\...%"..q.
00f0    67 53 11 46 1c 2f 80 c1 f4 e0 a2 86 50 45 23 c8    gS.F./......PE#.
0100    d4 9a 91 4e f1 62 c7 84 b3 ed 3f 68 26 16 82 08    ...N.b....?h&...
0110    46 c5 54 00 82 ad 19 cf 66 ff 06 51 44 38 30 06    F.T.....f..QD80.
0120    26 a8 e7 62 ea 81 05 93 0d 59 30 8e 13 cf 45 89    &..b.....Y0...E.
0130    0d 98 1e 41 a9 88 02 fd 03 65 f5 40 34 53 bb ce    ...A.....e.@4S..
0140    b3 6e 63 73 10 cc 54 fd f6 ef 9f 19 17 f5 ff 01    .ncs..T.........
0150    40 3e 0f af 46 5f d3 f6 3a b2 bf 82 98 26 2e 67    @>..F_..:....&.g
0160    d2 8b e2 56 56 c9 cd 0f 0b 3f 0f d1 68 65 11 68    ...VV....?..he.h
0170    6f e1 76 ce af f3 46 0f 26 8f 87 b8 82 a2 0d 97    o.v...F.&.......
0180    47 88 59 90 48 72 58 7a 1c e4 29 b3 26 51 ac 9c    G.Y.HrXz..).&Q..
0190    91 76 22 75 99 ab e2 be 2e ce bf 22 14 a6 10 94    .v"u......."....
01a0    e4 73 7d 56 ab 3a 5c 1c 96 e7 36 be ec 23 34 75    .s}V.:\...6..#4u
01b0    df c9 36 05 79 35 fd 3d 2e 8f af b2 b9 3f ca 73    ..6.y5.=.....?.s
01c0    56 b2 4b 1e e8 29 3f 6a 44 d6 3f be ad c6 e5 92    V.K..)?jD.?.....
01d0    5b 34 dc cc 7e 03 50 4b 07 08 2c 92 15 06 ac 01    [4..~.PK..,.....
01e0    00 00 cc 02 00 00 50 4b 01 02 14 00 14 00 08 08    ......PK........
01f0    08 00 fa 71 9b 51 2c 92 15 06 ac 01 00 00 cc 02    ...q.Q,.........
0200    00 00 0c 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0210    00 00 00 00 69 70 63 2d 65 6e 76 65 6c 6f 70 65    ....ipc-envelope
0220    50 4b 05 06 00 00 00 00 01 00 01 00 3a 00 00 00    PK..........:...
0230    e6 01 00 00 00 00                                  ......
```

Appendix C

Contents of enclosed CD

Also available from `https://gitlab.stdin.cz/ts/dp-security-assessment`.

```
README.md ............................ the file with CD contents description
DP_Stefan_Tomas_2021.pdf .............. the thesis text in PDF format
DP_Stefan_Tomas_2021.zip .............................. XeLaTeX source
bappstore_currentlist.txt ........ response for /bappstore/currentlist
burp_fuzzer ...................... source code of the fuzzer applications
    client.py ............................. high-level WebSocket fuzzer
    client_low_level.py ..................... low-level WebSocket fuzzer
    server.py ........................................ WebSocket server
    websockets_licence.txt ........... licence of the websockets library
```