FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Authentication, authorization, and session management in the HTTP protocol
Student:	Bc. Klára Drhová
Supervisor:	RNDr. Daniel Joščák, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2018/19

Instructions

Review the authentication, authorization, and session management methods in the HTTP protocol. Focus on the security of these methods and present their general security weaknesses.
Get acquainted with the Burp Suite tool that serves as a web application security assessment tool. Find out how this tool addresses authentication, authorization, and session management. Learn how to extend this tool with additional features.
Create an extension to the Burp Suite that will help with authentication, session management, and management of authorization tokens during penetration testing of web applications. Test the resulting extension.

References

Will be provided by the supervisor.

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

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

Prague January 12, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Authentication, authorization, and session management in the HTTP protocol

Bc. Klára Drhová

Department of Computer Systems
Supervisor: RNDr. Daniel Joščák, Ph.D.

April 29, 2018

Acknowledgements

I would like to thank my supervisor RNDr. Daniel Joščák, Ph.D., for his valuable advices and comments and for the time he spent helping me with this thesis. I am grateful to my colleagues for their feedbacks on the created extension. A very special gratitude goes out to Goran Pogačić, mag. ing. comp., for the fruitful discussions he provided me so many times and for his faith in the extension.

I am grateful to my family for providing me with unfailing support and continuous encouragement throughout all my life. Finally, I must express my very profound gratitude to my partner for all his love, patience, and support he provides me every day. Thank you.

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. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on April 29, 2018 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2018 Klára Drhová. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Drhová, Klára. *Authentication, authorization, and session management in the HTTP protocol.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato diplomová práce se zabývá způsoby autentizace, autorizace a správy relací, které jsou dennodenně používány uživateli Internetu v rámci protokolu HTTP. Důraz je kladen především na bezpečnost a bezpečnostní slabiny jednotlivých metod. Dále byl v rámci této diplomové práce vytvořen doplněk pro nástroj Burp Suite. Tento nástroj slouží k testování webových aplikací a je využíván bezpečnostními specialisty po celém světě. Hlavním přínosem vytvořeného doplňku je snadná správa autentizačních, autorizačních a dalších tokenů obsažených v HTTP zprávách, požadavcích a odpovědích, a dále pak možnost spravovat více relací najednou a tím několikanásobně urychlit automatizované skenování webových aplikací.

Klíčová slova autentizace, autorizace, správa relací, protokol HTTP, Burp Suite, doplněk Authentication Master, automatizace penetračních testů

Abstract

This master's thesis deals with methods of authentication, authorization, and session management in the HTTP protocol that are used every day by Internet users. The main emphasis is placed on security of individual methods and their security weaknesses. Furthermore, an extension to the Burp Suite tool was created. This tool is used for web application testing by many security specialists worldwide. The main benefit of the created extension is easy management of authentication, authorization, and other tokens contained in HTTP messages, requests and responses, as well as the ability to manage multiple sessions at the same time, speeding up the automated web application scanning several times.

Keywords authentication, authorization, session management, HTTP protocol, Burp Suite, Authentication Master extension, penetration test automation

Contents

List of Figures

List of Tables

Introduction

The Internet has become a part of our lives. We use it every day and almost everywhere – when chatting with friends using a mobile application, when updating our computer, or when downloading some film. Today, even many household appliances are able to connect to the Internet. Our television can play videos stored on the Internet, we can turn on the heating at our homes while being at work, and our washing machines notify us once the wash program is finished. Wireless networks, such as Wi-Fi, 3G, and LTE networks, are almost omnipresent. The HTTP protocol is one of the main application protocols used in the Internet today. It enables data transfers across the Internet.

Security is the main topic in most companies recently. Users began to take care of their personal data and public data leakage is a perfect recipe for destroying the reputation of a company. To preserve confidentiality of our data, data encryption is used. To verify identity of users and devices on the Internet, authentication methods are used. Once the entity is authenticated, it is important to check whether the entity is allowed to perform actions it requests. Authorization methods serve this purpose. It is also necessary to maintain some information about clients (for example, their language preferences). To do so, session management is used. All these techniques are used daily by billions of people without knowing it – when shopping in an online store, accessing internet banking, or sharing posts on a social network.

Today, great emphasis is placed also on security of web applications. Many tools for web application testing have been created and companies invest more and more money in security testing in general. Consequences of an attack on a vulnerable application may be serious. Data and information may be disclosed, modified, or deleted. An attacker may gain access to company's devices. Customers may be tricked to perform some unintended, malicious actions. It is thus important to test web applications regularly. The Burp Suite is one of the most popular web application testing tools used by security experts worldwide.

This thesis describes methods of authentication, authorization, and session management that are used together with the HTTP protocol. Primarily, their security aspects are analysed. Furthermore, an extension to the Burp Suite tool created within the thesis is described.

Main purpose of the created extension is to manage authentication tokens, session identifiers, and other similar tokens that can be found in HTTP messages. There are many possibilities where the token can be located – in an HTTP header, in a URL address, in a message body, etc. To make the testing and extension settings as easy as possible, many options were implemented. Testers may parse and update tokens located, for example, in HTTP cookies, headers, GET and POST parameters, XML bodies, and URL addresses.

Another desired feature of the extension is to enable testers to maintain multiple sessions at the same time. Thus, web applications may be scanned by multiple threads without any conflicts – something that is not possible in the Burp Suite itself. This feature helps to speed up testing of web applications, especially active scanning and fuzzing. The extension also allows to create new sessions automatically and to define under what circumstances sessions expire. Every expired session is discarded and replaced by a new one if necessary. To avoid repetitive setting of the extension, it is possible to export user settings to an XML file and import them anytime later.

The structure of this master's thesis is as follows: Chapter 1 describes the HTTP protocol in general, as well as methods of authentication, authorization, and session management used in this protocol. Security considerations for every such method are mentioned. Section 1.2 contains a brief description of the SSL and TLS protocol.

The Burp Suite tool along with some its features is described in the Chapter 2. The way this tool addresses authentication, authorization, and session management is also mentioned in this chapter.

Chapter 3 deals with questions related to the design of the extension. Functional and non-functional requirements are mentioned. Some basic concepts used in subsequent chapters are explained.

Chapter 4 describes implementation choices, such as the choice of the programming language, that were made. This chapter also includes a list of implemented classes, a description of how to use the extension, and several example use cases are mentioned.

Details related to testing of the created extension, as well as a comparison with other similar available extensions can be found in Chapter 5.

CHAPTER 1

Analysis

This chapter describes theory necessary for understanding authentication, authorization, and session management methods used on the Internet today. These methods are then examined with focus on their security aspects.

1.1 HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is a stateless application protocol. Its purpose is to enable data transfers across the Internet. Development of HTTP was initiated in 1989 by Tim Berners-Lee. The first standard defining HTTP/1.1, the most common version of HTTP in use today, occurred in 1997 [1]. This standard was soon replaced by a new one, RFC 2616 [2], and then again in 2014 by a group of standards, RFC 7230 to RFC 7235 [3–8]. A nice overview of HTTP protocol can be found on Mozilla web page [9].

HTTP is a request/response protocol and we distinguish HTTP clients and HTTP servers. An HTTP client, a web browser for example, is an application that establishes a connection to a server and sends requests to it. An HTTP server, a web page application for example, is an application that accepts this connection, processes the requests and returns responses that usually contain some resources. Such a resource can be an HTML file, a picture, some data, etc., and is identified by a string called Uniform Resource Identifier (URI). HTTP provides clients with a generic, uniform interface that is independent of application implementation and independent of available resources.

The following example illustrates a typical client request and server response. The string `"http://www.example.com/hello.txt"` is URI of the resource and `"Hello World! My payload includes a trailing CRLF.\n"` is its content. It was taken from [3].

```
Client Request:
     GET /hello.txt HTTP/1.1
     User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l
```

```
        zlib/1.2.3
    Host: www.example.com
    Accept-Language: en, mi

Server Response:
    HTTP/1.1 200 OK
    Date: Mon, 27 Jul 2009 12:28:53 GMT
    Server: Apache
    Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
    ETag: "34aa387-d-1568eb00"
    Accept-Ranges: bytes
    Content-Length: 51
    Vary: Accept-Encoding
    Content-Type: text/plain

    Hello World! My payload includes a trailing CRLF.
```

Requests and responses, also called messages, have a fixed format. They consist of a *start-line* followed by any number of *header fields* (collectively referred to as the *headers*). Then an empty line follows indicating the end of the header section and the last is a message body that is optional.

Requests and responses differ only in the start-line. Format is the following:

```
Request start-line:
    Method Request-target HTTP-version

Response start-line:
    HTTP-version Status-code Reason-phrase
```

There are several methods (`Method` part) with different purposes that are commonly used in HTTP. Some of them are standardized in the RFC 7231 [4], others are not:

- *GET* – this method requests a current representation of the target resource; it does not modify the resource
- *HEAD* – this method is the same as *GET*, but requests only the start-line and header section (response is the same but without body)
- *POST* – this method is used to supply the server with some data that are then processed depending on the requested resource – the data can be for example used for modification of the resource, for creation of a new resource or can define a format of a server response
- *PUT* – this method replaces all current representation of the target resource

- *DELETE* – this method removes all current representations of the target resource
- *CONNECT* – this method establishes a tunnel to the server that is identified by the target resource
- *OPTIONS* – this method is used to describe the communication options for the target resource
- *TRACE*[1] – this method performs a message loop-back test along the path to the target resource; if enabled, the web server responds to a request by echoing the whole request in the response body
- *PATCH* (not specified in the mentioned RFC document) – this method is used to apply partial modifications to a resource

Also, the status codes are standardized (`Status-code` part). It is a 3-digit integer code that indicates whether a specific HTTP request has been successfully completed. There are five classes: informational responses, successful responses, redirects, client errors, and server errors. Table 1.1 contains some of the most commonly used status codes and corresponding recommended reason phrases.

The HTTP headers allow to pass additional information with the message – a request or a response. A header consists of a case-insensitive name followed by a colon, optional space, and its value. The header must not contain any line break. Some headers are used by both requests and responses, some of them are applicable only for requests and some of them only for responses. User can also use his own, non-standard headers. Their names were originally prefixed with "*X-*" but this convention was deprecated in 2012. The reason was that some originally non-standard headers had become standardized.

Some HTTP headers carry information about the client[2], other contain information about the context of request[3]. Information about the server can be sent via headers[4], too. Of course, the HTTP headers can also carry information about the request, response or the message body itself[5].

[1]The HTTP TRACE method is designed for diagnostic purposes, but can lead to the disclosure of sensitive information such as cookies. Attack using this method is called *Cross Site Tracing attack* and detailed information can be found on the OWASP web page [10]. Thus, the TRACE method should be disabled on production web servers.

[2]A good example of such header is *User-Agent* header that contains information about application type, operating system, etc.

[3]For example, *Cookie* header contains stored HTTP cookies previously sent by the server and *Referer* header contains an address of the web page from which a link to the currently requested page was followed.

[4]The *Server* header contains information about the software and software versions used by the origin server.

[5]Headers like *Content-Length*, *Content-Language*, and *Content-Encoding* are good examples.

Status code	**Reason phrase**
Information responses	
100	Continue
101	Switching Protocols
Successful responses	
200	OK
201	Created
202	Accepted
204	No Content
Redirection messages	
301	Moved Permanently
302	Found
304	Not Modified
307	Temporary Redirect
Client error responses	
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
408	Request Timeout
415	Unsupported Media Type
Server error responses	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

Table 1.1: Status codes and recommended reason phrases

There are many headers that are commonly used. Here you can find some examples that are related to security:

- *Cache-Control*, *Expires* and *Pragma* – These headers can be used to prevent the browser from storing a local cached copy of content received from the web server. Caching is used for optimizing performance but this behaviour can be risky if there is some sensitive or confidential information in the server response. Thus, it is recommended to instruct the browser to avoid caching such data.

- *Content-Security-Policy (CSP)* – This header is used to inform the client browser about expected behaviour of the application and to whitelist content sources. For example, sources of scripts and images or plugin types can be defined. This header is complex in nature but allows the browser to enforce security constraints more intelligently. If set correctly, it can significantly decrease the chances of successful *cross-site scripting attacks.*

- *Cross-origin resource sharing (CORS) headers* – These headers are used to make interactions with other websites secure. Such an interaction can include loading scripts or fonts from different websites. A header *Access-Control-Allow-Origin* is used to define which websites are allowed to access resources on the target website. If the CORS header are missing, requests from other websites are forbidden by default.

- *Public-Key-Pins* – This header instructs the browser which certificate to trust and for how long time. When the browser meets the header for the first time, it will save that specific pinned certificate and will double-check it next time. This header is not widely used due to its complicated implementation and possible errors it can cause. Another alternative is a *Public-Key-Pins-Report-Only* header which reports problems but doesn't lock users out. This header helps to prevent forged X.509 certificates and *rogue attacks* in case the certificate authority is compromised.

- *Referer*[6] and *Referrer-Policy* – The *Referrer-Policy* header is used to specify when the browser should set the *Referer* header. The *Referer* header allows the browser to specify the address of a web page from which the target website or resource was loaded. This information should not be used for security checks, although it often is, but can be useful for analytics.

- *Strict-Transport-Security* – This header is sometimes called *HSTS* header[7] and is used to force the browser to use a secure HTTPS connection

[6] *Referer* is a misspelling of the word "*referrer*".

[7] HTTP Strict Transport Security

when communicating with the server. If this header is properly used, the browser will not allow to access the website over HTTP and will prevent the client from overriding an SSL certificate warning[8]. Thus, the *SSL strip attacks* are prevented and the *man-in-the-middle attacks* are more difficult to perform.

- *X-Content-Type-Options* – This header serves as a defence of *MIME type confusion attacks*. Most of the browsers try to guess content type of the server response instead of trusting the response header. This technique, called *MIME sniffing*, can cause that the browser is intentionally confused by the attacker and executes some malicious code. This header prevents the browser from guessing the type and instructs it to use the value of the *Content-Type* header instead.

- *X-Frame-Options* – This header prevents the attacker from embedding the website in an *iframe* HTML element on her malicious website. If missing, the attacker can perform *clickjacking attack*. In this attack, the victim is directed to the attacker's website and manipulated to perform unintentional actions on the target application. This header is used to define which websites are allowed to frame the website. It has been deprecated and should be replaced by the *frame-options* directive in the *Content-Security-Policy* header.

- *X-XSS-Protection* – This header is used to inform the browser that browser's prevention against *cross-site scripting attacks* should be enabled or disabled. This header has been deprecated and should be replaced by the *reflected-xss* directive in the *Content-Security-Policy* header.

1.1.1 HTTP Cookies

As the HTTP protocol is mostly stateless, there was a necessity for some mechanism for storing stateful information. Two headers, the response *Set-Cookie* header and the request *Cookie* header, were introduced. The former header serves for setting a small piece of data, called a cookie, that stores stateful information. This cookie is sent by the server to the browser which remembers it and later sends it back within the request to the same server using the *Cookie* header. An HTTP cookie is specified in RFC 6265 [11].

Cookies are sent with every request and can be divided into two groups, *session cookies* and *permanent cookies*. Session cookies should be deleted when the browser shuts down but nowadays, many browsers use a technique called *session restoring* which causes that cookies are restored after start of the browser. *Permanent cookies* expire at a specific date and time (*Expires*

[8]SSL certificate warning appears when the website certificate is invalid or if it is faked by an attacker

attribute) or after a specific length of time (*Max-Age* attribute). Cookies are usually used for these purposes:

- **Session management** – maintaining information about logins, items added in the shopping cart, etc.
- **Personalization** – storing user preferences and settings
- **Tracking** – recording and analyzing user actions and behaviour

There are many security considerations associated with cookies. Cookies are often used to identify some particular user and his session. Stealing a cookie can thus lead to hijacking the authenticated session of the user (*session hijacking attack*). Attacker can use it for getting access to some confidential information or performing some malicious actions. Cookies are usually stolen using *social engineering* techniques or *cross-site scripting (XSS) vulnerability* in the application. Also, the *cross-site request forgery (CSRF, XSRF) attack* makes use of cookies.

To prevent these attacks, some flags were specified. These flags restrict the way the browser can manipulate with user's cookies. There is a *Secure* flag that limits the scope of the cookie to secure channels and protects thus its confidentiality. The cookie with this flag set will be only sent over the HTTPS protocol and never over the HTTP protocol that does not offer any encryption. To prevent *cross-site scripting attack*, *HttpOnly* flag is used. It instructs the browser to omit this cookie when providing access to cookies via some "non-HTTP" APIs, for example, JavaScript's `document.cookie` API. Such cookies can be only sent to the server automatically within the request. There is also a new, experimental *SameSite* flag that instructs the browser not to send the cookie within cross-site requests. Such a cookie can only be sent in requests originating from the same origin as the target domain. This flag mitigates *cross-site request forgery attacks.*

For more information about cookies, refer to the Section 1.5.

1.1.2 HTTPS

HTTPS (HTTP Secure) is a protocol that enables secure communication over a computer network. It makes use of the HTTP protocol together with SSL or TLS protocol which serves for encryption. HTTPS URL addresses begin with "*https://*" and use port 443 by default as opposed to URL addresses beginning with "*http://*" and using port 80 as in the case of HTTP. Generally, it is recommended to use HTTPS instead of insecure HTTP for all web pages. Both *SSL* and *TLS* are discussed in the Section 1.2.

1.2 SSL and TLS

Transport Layer Security (TLS) and Secure Sockets Layer (SSL) are security protocols that provide communications privacy over the Internet. Their purpose is to ensure data confidentiality, message integrity, and authentication. Thus, eavesdropping, tampering, and message forgery should be prevented. These protocols are application protocol independent – a higher level protocol can layer on top of them transparently.

There are several versions of these protocols. SSL is today prohibited from use by Internet Engineering Task Force (IETF) and it is recommended to use its successor, TLS. Current version of the TLS protocol, TLS 1.2, is specified in RFC 5246 [12]. At the time of writing this thesis, there is a public draft of a standard for TLS 1.3 that has gone through 28 versions. It is approved now, and some companies have already begun to implement it. The new version should provide improved security and faster speed due to use of newer encryption methods and elimination of unnecessary steps in the TLS handshake.

The TLS protocol is composed of two layers. The lower layer, that is on the top of some reliable transport protocol (usually the TCP protocol), is the *TLS Record Protocol*. Its main purpose is to provide connection security. Such a connection is private and symmetric cryptography is used for data encryption. The shared key necessary for this symmetric encryption must be unique for each connection and is generated from some secret negotiated by another protocol (for example, the TLS Handshake Protocol). This connection is also reliable. Integrity of messages is checked using a keyed Message authentication code (MAC).

The higher layer is the *TLS Handshake Protocol*. It allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the applications on the client and the server begin to communicate. It provides authentication of the peers' identity using asymmetric cryptography and the negotiation of a shared secret is secure and reliable. The communication cannot be modified by an attacker without being detected and the negotiated secret cannot be eavesdropped.

Today, the TLS protocol is essential to keeping content of HTTP messages private. This is important especially for transmission of user's credentials. However, there are still many security risks. If a user is redirected to a fraudulent web page, he may insert his credentials there and provide them thus unintentionally to an attacker. *Phishing attacks* are used very often for this purpose. As TLS certificates are issued to almost any user (including attackers), a valid certificate is not a guarantee of security.

There are several attacks against the SSL and TLS protocols that have appeared during their existence. Many of them are summarized in RFC 7457 [13]. Matthew Green, a cryptographer and professor at Johns Hopkins University, has published detailed descriptions of some attacks on his blog [14]. You can find a brief description of chosen attacks in the Appendix B.

1.3 Authentication

Authentication is a process that verifies the identity of a user – who a user is. Authentication precedes authorization described in Section 1.4. The user usually provides credentials to an application or a server that are then compared to already known ones[9]. If the credentials match, the user is successfully authenticated and is granted authorization for access. HTTP authentication is defined in RFC 7235 [8].

In HTTP protocol, if a request sent to a server lacks valid authentication credentials for the target resource, a response with the 401 (Unauthorized) status code and a *WWW-Authenticate* header is returned. This header should contain at least one authentication method (also called authentication scheme, or authentication type) that should be used to gain access to the resource. Example of a response can be found below. If the request contains valid credentials but these are not adequate to gain access to the resource, the server should respond with the 403 (Forbidden) status code.

```
HTTP/1.1 401 Unauthorized
Date: Mon, 27 Jul 2009 12:28:53 GMT
WWW-Authenticate: Newauth realm="apps", type=1,
    title="Login to \"apps\"", Basic realm="simple"
```

The *WWW-Authenticate* header can contain more than one acceptable authentication method, and each this method can contain a comma-separated list of authentication parameters. Furthermore, the header itself can occur several times. A list of HTTP authentication schemes is maintained by IANA [15].

There is a *realm* authentication parameter that is used by almost all HTTP authentication schemes. Its value is an arbitrary string defined by a server. In combination with the root URI of the server, it defines the protection space. The protected resources on a server can be partitioned into several protection spaces, each with its own allowed authentication schemes and own database of known credentials.

After receiving a 401 (Unauthorized) response, a client should use *Authorization* header in order to authenticate itself. The value of this header contains a method of the authentication and credentials of the user. How the credentials should be encoded prior to the transmission is defined in the authentication scheme specification.

[9]It is recommended that only hashes of the passwords are stored and that the passwords are hashed together with some random data (called *salt*) that is also stored. There are some hash functions that are specially designed for key derivation and password hashing – for example, *Argon2*, *bcrypt*, or *scrypt*.

There is a similar mechanism when sending a request to a proxy in order to use this proxy. In that case, the 407 (Proxy Authentication Required) status code is returned and the response contains a *Proxy-Authenticate* header. A client identifies itself using a *Proxy-Authorization* header.

This general HTTP authentication framework does not provide any mechanism to maintain confidentiality of the credentials. The HTTP protocol depends on the security properties of the underlying layers (especially the transport or session layer) which should provide connection that ensures confidentiality of the headers. For example, the TLS protocol can be used to secure the connection, as mentioned in the Section 1.2.

An important security consideration is that many HTTP clients (e.g., browsers) store authentication information indefinitely. This is especially a problem if the validity of the authentication information is not limited somehow – Basic authentication scheme is a good example. In the general HTTP authentication framework, there is no mechanism for the server how to instruct the client to discard this information. However, the expiration and revocation of the credentials can be specified in the authentication scheme definition itself. Common circumstances for the expiration or revocation may include an inactivity of a client for some period of time or a session termination indication (as a "*logout*" action).

If the server hosts resources belonging to multiple parties and if they are hosted under the same root URI, the credentials can also be at risk. Successfully authenticated users which requested some resource can use the same credentials when requesting another resource on the same origin server. This another resource can be a malicious one and can harvest these valid credentials. Possible mitigation technique is to separate resources by using a different host name or a port number for different parties.

Servers that implement authentication and need to store users' credentials in order to authenticate users should store these credentials in such a way that even a leak of the stored data does not allow an attacker to easily recover the credentials. This is especially important in cases when the users are allowed to choose their own passwords, as they usually tend to choose weak passwords or to use the same password for different applications. There are some hash functions recommended especially for password hashing. These hash functions should be used together with method called "*salting the password*". Still, a leakage or a theft of the database of passwords that are poorly secured is very serious and frequent problem today.

Users' passwords should have a reasonable amount of entropy – human-memorable passwords are usually vulnerable to *dictionary attacks* independently of the authentication algorithm used. Every failed authentication attempt should be logged, since many repeated login failures caused by a single client may indicate an attacker that attempts to guess users' credentials. However, the server should never log any sensitive information (e.g., entered passwords in cleartext). Another security problem associated with passwords is that users tend to reuse their passwords for many different applications and web sites. Thus, if an attacker gains a password of a user in one application, she is very often able to gain access to other accounts of this user in different applications.

If an attacker is able to perform the *man-in-the-middle attack* and to modify the list of authentication schemes allowed by the server in server response, then she can add some weak authentication scheme, such as the Basic authentication, to the list and hope that the client will choose this one. If this happens, the user's credentials can get exposed to the attacker. Therefore, the client should always choose the strongest available authentication scheme. But this is not a flawless solution, as the attacker can also completely replace all the offered schemes by the weak one and then easily harvest the credentials. A hostile or compromised proxy server can be used to perform such an attack. To detect these attacks, the user agent can be configured to remember the strongest authentication scheme offered by the server in the past and if there will be only weaker options sometime in the future, a warning message should be thrown. To completely prevent similar attacks, it is recommended to use the HTTPS protocol.

The Basic authentication scheme is one of the weakest ones, as it transmits user's credentials in an encoded form. The security level is the same as if they were transmitted in the plaintext. The Digest authentication scheme uses hash functions to hide the raw password. However, it is not easy to implement this authentication scheme correctly and it is vulnerable to *offline dictionary attacks*. Thus, it is not commonly used today. As password-based methods suffer from many security problems, some authentication schemes, such as the HOBA authentication scheme, aim to avoid them by using some pre-deployed strong secret keys (e.g., digital signatures). In some cases, the authentication method also offers authentication of the server. The Mutual and SCRAM authentication schemes are such examples. There is also the VAPID authentication scheme that is focused on authentication of an application server to a push service within the Web Push protocol. You can find detailed descriptions of the individual authentication schemes in the following subsections. The non-standard form-based authentication technique is mentioned at the end of this section. It is probably the most common authentication technique used on the Internet today. The main reason may be that it allows to control appearance and exact behaviour during the authentication process unlike the default browser pop-up dialogues used by the HTTP authentication schemes.

1.3.1 Basic Authentication

Basic authentication is a simple HTTP authentication scheme defined in the RFC 7617 [16]. In this scheme, the user's credentials (both the user name and the password) are transmitted together in an encoded form using Base64 encoding.

Base64 encoding is a method that represents binary data in form of an ASCII string. Every character of the output string represents 6 bits of the input binary data. As encodings do not serve to ensure the confidentiality and can be reverted without knowledge of any secret data, this scheme cannot be considered secure unless it is used with some other method that ensures the confidentiality – for example the TLS protocol mentioned above.

The exact name of this scheme used in requests and responses is "*Basic*". A server should always append the *realm* authentication parameter and optionally also the *charset* authentication parameter in the *WWW-Authenticate* header. A client processes user's credentials in the following form:

1. The user name is concatenated with a single colon character ("*:*") and the password.

2. Then, the output string from the first step is encoded into an octet sequence – the default encoding is not defined.

3. Finally, the byte data generated in the second step is encoded into a string of ASCII characters using the Base64 encoding.

In this authentication scheme, user's credentials must not contain any control[10] characters and the user name must not contain a colon character. Text before the first colon is supposed to be the user name and the text after it is supposed to be the password during validation of the credentials by the server. An example of the *Authorization* header used by the client for user name `"user"` and password `"password"` is shown below.

The following example demonstrates a communication between a client and a server. The client requests some resource. The server responses with a 401 response, meaning that the client has to authenticate itself in order to obtain the resource from the server. The client finally authenticates itself using the Basic authentication scheme.

```
Client Request:
     GET /resource HTTP/1.1
     Host: www.example.com

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: Basic realm="example"

Client Request:
     GET /resource HTTP/1.1
```

[10]Non-printing characters, also called control characters, do not represent any written symbol. For example, a *null* character, a *line feed* character, or a *horizontal tab* character are examples of control characters in ASCII that are still in common use today.

```
     Host: www.example.com
     Authorization: Basic dXNlcjpwYXNzd29yZA==

Server Response:
     HTTP/1.1 200 OK
     ...
```

It is supposed that resources in the same folder (or in sub-folders of this folder) are within the same protection space and a client may send the corresponding *Authorization* header together with a request for resource without waiting for the 401 response from the server.

As for the security aspects of this authentication scheme, the most serious security problem is the previously mentioned way of processing the user's credentials. As the credentials are only encoded, the security level is the same as if they were transmitted over the network in the cleartext. If there is no additional security measure, such as HTTPS, the credentials can be easily overheard by an attacker. Thus, the scheme itself is not sufficient to protect sensitive data.

This scheme is also vulnerable to spoofing the credentials by a hostile server. The user can be misled to believe that he is connecting to a target server containing required resources, and instead, he is connecting to the hostile server. This server can request user's credentials and store them for later use.

Basic authentication can be used as identification of a user for some insensitive purposes – for example, for gathering of statistics. In this case, the user's credentials should be created by the server and the user should not have the option to change his password, as many users tend to use the same password for multiple different applications.

Basic authentication scheme can also be used with one-time passwords. In such a case, capturing of the credentials is not so problematic, as the credentials should not be valid anymore after the first use of them. But still, an attacker performing a *man-in-the-middle attack* can misuse the credentials.

1.3.2 Bearer Authentication

Bearer authentication, sometimes also called token authentication, is an HTTP authentication scheme defined in the RFC 6750 [17]. This scheme makes use of security tokens, called bearer tokens, which are strings usually generated by an authorization server. A bearer token is an access token that has the property that anybody who owns the token (a "*bearer*") can use it in any way as any other bearer can. The bearer does not have to prove possession of any cryptographic key to use the bearer token.

The Bearer authentication scheme was defined within the OAuth 2.0 protocol described in the Subsection 1.4.2. It is probably the most common way of using the OAuth 2.0 API.

Clients firstly need to obtain an approval of the resource owner, also called an *authorization grant*. Bearer tokens are then issued to the clients by an authorization server in response to the acquired *authorization grant*. The clients can use the access token to gain access to the resource stored on a resource server. The resource server validates the access token, and if the token is valid, the server returns the requested resource to the clients.

The access token replaces various authorization constructs (e.g., user's credentials) and can have different validity periods. The exact content of the token is not specified in the previously mentioned RFC document; the token can contain, for example, some authorization information. An advantage of this approach is that the resource server does not need to implement a wide range of authentication schemes but only the Bearer authentication scheme.

There are three standardized methods of sending the bearer token to the resource server. Clients must not use more than one of these methods at once. The most common method is sending the access token within the *Authorization* request header. Another method is sending the token in the request body as a parameter called "*access_token*". This method is used in cases when browsers do not have access to the *Authorization* request header. The last method is sending the token within the request URI as the "*access_token*" parameter. In this case, the client should also send the *Cache-control* header containing the "*no-store*" option. This method should not be used unless the previous methods are not feasible.

The exact name of this scheme used in requests and responses is " *Bearer*". A server may append the *realm* authentication parameter or the *scope* authentication parameter in the *WWW-Authenticate* header. If a client request contained an access token and the authentication failed, the server should include the *error* parameter to provide the reason why the request was rejected. In such a case, also the *error_description* parameter with a human-readable explanation and the *error_uri* parameter can be appended.

The following example demonstrates a communication between a client and a resource server. The client requests some resource. The server responses with a 401 response, meaning that the client has to authenticate itself in order to obtain the resource from the server. The client then authenticates itself using the Bearer authentication scheme. The bearer token is however not valid and the resource server returns an error. The client finally authenticates itself using a new, valid token.

```
Client Request:
     GET /resource HTTP/1.1
     Host: www.example.com

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: Bearer realm="example"

Client Request:
     GET /resource HTTP/1.1
     Host: www.example.com
     Authorization: Bearer qT-6+A1j.9/4KpS

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: Bearer realm="example",
          error="invalid_token",
          error_description="The access token expired"

Client Request:
     GET /resource HTTP/1.1
     Host: www.example.com
     Authorization: Bearer Ut8_7Df49+73a4v

Server Response:
     HTTP/1.1 200 OK
     ...
```

There are multiple security considerations related to this authentication scheme. The bearer tokens may be misused and may include some sensitive information. Thus, it is necessary to protect them from disclosure during any transport and when saved in any storage. When sending the token in a request or a response, it is necessary to use an additional security protocol, such as TLS, with this authentication scheme.

Another possible security issue can be caused by sending the access token within the URI parameter. Depending on the server or the client, the full URI including the parameters may be logged – in server's log files, browser's history, etc. If an attacker gets access to these records, she will be able to find out the used access token and potentially to reuse it. The page URI is also sent in the *Referer* header of all requests made by that page, even if the requested resources are third party resources. The access token can be thus disclosed this way.

Depending on the way how the tokens are generated, an attacker may try to generate a bogus token or to modify some existing token (for example, to extend the validity period). Thus, she may be able to gain access to a resource without having appropriate privileges. The attacker may also attempt to simply reuse some access token that has been already used in the past (*token replay attack*). Another possible attack is to use some token to gain access to a resource on a different resource server that incorrectly supposes that the token was generated for it (*token redirect attack*).

Many security issues listed above can be mitigated by protecting the content of the access token. A digital signature or a Message Authentication Code (MAC) can be used for that. The integrity protection must be strong enough to prevent an attacker from modifying the token.

A reference to authorization information can be used instead of encoding the information itself into the token. In this case, the references must be infeasible to guess. To prevent the *token redirect attack*, it is recommended to include identification of the resource server in the token. To prevent or limit the *token replay attack*, the validity period of the token must be limited. It is recommended to restrict the validity period to one hour or less.

The authorization server and the resource server must implement TLS in order to protect confidentiality and integrity of the token (to prevent its disclosure or modification). The client must verify the identity of the resource server by validation of the TLS certificate chain and check of the list of revoked certificates[11]. Otherwise, a counterfeit server can request the access token and use it for gaining inappropriate access – similarly as in the case of the Basic authentication.

If cookies are used to store the bearer tokens, there is an additional security concern. The token must not be stored in a cookie that can be sent in the cleartext over HTTP or stolen using a *cross-site scripting attack*.

1.3.3 Digest Authentication

Digest authentication is another authentication scheme specified in the RFC 7616 [18]. A server response contains a *nonce* value, an arbitrary number that should be used only once. A valid client request contains a digest of the user name, the password, the received *nonce* value, the HTTP method, and the requested URI. Hash functions are used to create the digest – user's credentials are not transmitted in the cleartext as in the Basic authentication scheme. The length of the digest depends on the used algorithm and the digest is represented by a hexadecimal string.

The exact name of this scheme used in requests and responses is "*Digest*". A server may append the *realm* authentication parameter and many other parameters in the *WWW-Authenticate* header. The *nonce* parameter contains a string that should be uniquely generated each time the *WWW-Authenticate* header is used. Usually it is a Base64-encoded string or a hexadecimal string. The exact content is not specified and may depend, for example, on the current time, IP address of the client, *ETag*[12] of the resource, and many other factors. Validity period of the nonce can be limited by the server. The *opaque* parameter is a string of data generated by the server. The client should return

[11]Certificate Revocation List (CRL) is a list of issued certificates that are not valid anymore. A possible reason could be a disclosure of the corresponding private key.

[12]The *ETag* included in the nonce value prevents a *replay attack* for an updated version of the resource.

the value of this parameter within the *Authorization* header. The *stale* authentication parameter is optional and describes if the used nonce value was invalid or if the user's credentials were invalid. The *algorithm* parameter indicates an algorithm, a hash function, that should be used to create the digest. The *qop* parameter stands for the "*quality of protection*" – the value "*auth*" indicates authentication and the value "*auth-int*" indicates authentication with integrity protection.

The client should respond with the *Authorization* header. The values of the *algorithm* and *opaque* parameters must be the same as those sent by the server. The *response* parameter is used to prove that the client knows the password – this parameter contains the computed digest. The *username* parameter contains the user name either in the plaintext or rather as a hexadecimal hash code. The *qop* parameter must contain one of the options offered by the server. The compulsory *cnonce* parameter is an ASCII string provided by the client. It provides mutual authentication, message integrity protection, and avoids *chosen plaintext attacks.* The compulsory *nc* parameter stands for the "*nonce count*" and indicates the overall number of requests made with the same nonce. The *uri* parameter contains the URI address of the requested resource.

After the server receives a request from the client with *Authorization* header, it needs to check the validity of the digest. It must perform exactly the same operation as the client did and compare the result to the received digest. Because of the way the digest is computed (defined in the RFC 7616 [18]), the server does not need to know the user's password in cleartext.

The server can include multiple *WWW-Authenticate* headers using Digest authentication in the response. In such a case, each one of these headers must use a different algorithm. The first header must contain the most preferred algorithm, the last one the less preferred algorithm. The following algorithms are defined: *SHA2-256*, *SHA2-512/256*, and *MD5.* The *MD5* algorithm is not recommended and is defined only for backward compatibility, as *MD5* hash function is considered to be broken.

The following example demonstrates a communication between a client and a server. It was taken from the previously mentioned RFC 7616 [18]. The client requests some resource. The server responses with a 401 response, meaning that the client has to authenticate itself in order to obtain the resource from the server. Two possible algorithms are proposed by the server. The client finally authenticates itself using the Digest authentication scheme with the *SHA2-256* algorithm.

```
Client Request:
    GET /dir/index.html HTTP/1.1
    Host: www.example.org

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: Digest
        realm="http-auth@example.org",
        qop="auth, auth-int",
        algorithm=SHA-256,
        nonce="7ypf/xlj9XXwfDPEoM4URrv/xwf94BcCAzFZH4GiTo0v",
        opaque="FQhe/qaU925kfnzjCev0ciny7QMkPqMAFRtzCUYo5tdS"
     WWW-Authenticate: Digest
        realm="http-auth@example.org",
        qop="auth, auth-int",
        algorithm=MD5,
        nonce="7ypf/xlj9XXwfDPEoM4URrv/xwf94BcCAzFZH4GiTo0v",
        opaque="FQhe/qaU925kfnzjCev0ciny7QMkPqMAFRtzCUYo5tdS"

Client Request:
     GET /dir/index.html HTTP/1.1
     Host: www.example.org
     Authorization: Digest username="Mufasa",
        realm="http-auth@example.org",
        uri="/dir/index.html",
        algorithm=SHA-256,
        nonce="7ypf/xlj9XXwfDPEoM4URrv/xwf94BcCAzFZH4GiTo0v",
        nc=00000001,
        cnonce="f2/wE4q74E6zIJEtWaHKaf5wv/H5QzzpXusqGemxURZJ",
        qop=auth,
        response="753927fa0e85d155564e2e272a28d1802ca10daf449
          6794697cf8db5856cb6c1",
        opaque="FQhe/qaU925kfnzjCev0ciny7QMkPqMAFRtzCUYo5tdS"

Server Response:
     HTTP/1.1 200 OK
     ...
```

Digest authentication should always be used over a secure channel like the HTTPS channel using the TLS protocol. Otherwise, *man-in-the-middle attacks* are quite easy to perform and the confidentiality of the message content is not ensured.

The server needs to store some data derived from the user's credentials in order to be able to authenticate the user. For Digest authentication, a user name and a hash of concatenated user name, realm, and password are sufficient to be stored in the password file. This approach has one advantage and one disadvantage. As for the advantage, the user's password does not need to be stored in plaintext and if the password file is compromised, an attacker has to conduct a *brute-force attack* or a *dictionary attack* to obtain the user's password. The disadvantage is that if an attacker gets data from the password file, she immediately gains access to any document on the server that is in the corresponding realm. The user name together with the hash of the user name, the realm, and the password stored in the password file is sufficient to authenticate the user. Thus, the attacker is able to spoof his identity. However, as the realm is part of the stored hash, the attacker should not be able to gain access to documents in other realms. There are some security consequences of this – the password file must be well protected and the realm string should be unique (e.g., a name of the host can be used).

In this scheme, a client does not have any option how to authenticate the server. Nevertheless, the server is able to authenticate the client and in the case of the Digest authentication, the method is much more secure than in the case of the Basic authentication, as the user name and the password is protected by a hash function. Even if there is no HTTPS in use, if the used hash function is secure enough, the password should be protected.

As for the *nonce* parameter, its value is generated by the server and can be restricted in many ways. For example, the server may generate a different nonce value for every client, for a particular resource (and its versions), or after some period of time. These restrictions strengthen the overall security and prevent some types of attacks (e.g., *replay attacks*). On the other hand, they can be very performance consuming and may lead to some failures. For example, if there is a new nonce value in every single response, authentication failures will appear for any pipelined requests.

Since the URI address of the resource is incorporated into the digest, *replay attacks* for GET requests are very limited. If an attacker overhears some client request, she will be only able to exploit its information to obtain the same resource that was demanded by the original request. However, in such a case, it is very likely that if she can overhear client requests, she can also overhear server responses. Thus, she will be able to eavesdrop the requested resource and will not need to perform the *replay attack* to get it. This is a very different situation from Basic authentication. In the Basic authentication scheme, if the attacker is able to overhear user's request together with user's credentials, she is able to obtain any resource that is protected by these credentials.

As for the requests that perform some action, the security largely depends on the algorithm for nonce generation. If the nonce value depends on the client IP, current time, version of the resource, and server identification, the *replay attacks* are very complicated. The attacker must spoof his IP address and perform the attack before the validity of the nonce expires. To completely eliminate *replay attacks*, one-time nonce values can be used.

If the *qop* parameter equals to the "*auth-int*", then the integrity of parameters used for the calculation of the *WWW-Authenticate* and *Authorization* header values are protected. Otherwise, they are not. Even if this option is used, the content of POST and PUT requests itself is not protected. Thus, this is not a sufficient defence of *replay attacks*.

The *cnonce* parameter is used to prevent *chosen plaintext attacks.* A malicious server or an attacker performing the *man-in-the-middle attack* may arbitrarily choose a value of the nonce parameter and give it to the client that will use it for the digest computation and send the digest back to the malicious entity. In general, the ability to choose some input value makes the cryptanalysis easier. Even if there is no known method how to exploit that for the hash functions offered by this scheme at the moment, the creators wanted to make these attacks even much harder by introducing the *cnonce* parameter. Value of this parameter is not controlled by the attacker and is used as an additional input into the hash function.

The *cnonce* parameter also prevents a *dictionary attack* when the attacker chooses one particular nonce value, computes the expected client's digest for this nonce value and many different common passwords, and stores the values in a database. If there is no *cnonce* parameter in use, she can use the *man-in-the-middle attack* to change the server's nonce value to the one she used for the precomputation and compare the digest computed by the client with those stored in her database. If there is a match, then the corresponding password to the matching digest is the user's password.

Another possible attack prevented by the *cnonce* parameter is the attack in which the attacker gathers digests computed by many different clients in response to the same nonce value. The attacker then tries to find some set of passwords within the harvested set of clients' digests. In this attack, the attacker tests a few passwords for many different clients, as opposed to the previous attack where she tested many different passwords for a single client.

A chain is only as strong as its weakest link. The security of this scheme largely depends on the randomness of the *nonce* and *cnonce* parameters. Therefore, if there is some weak random generated in use, then the protocol is not secure.

There is also a possibility of *denial-of-service attacks.* If there is some malicious client that sends out many unauthenticated requests, the server creates many new structures for storing the *nonce* value, etc. Thus, some resources may get depleted. There are numerous solutions for mitigation of these attacks. For example, only one structure may be generated for one particular client.

1.3.4 HTTP Origin-Bound Authentication (HOBA)

HOBA authentication is an HTTP authentication method based on digital signatures that is specified in the RFC 7486 [19]. It is still under examination. Its main goal is to offer an authentication scheme that avoids usage of passwords and security problems related to them. HOBA also adds some useful features such as a logout functionality. A client generates a new key pair, a public key and a private key, for every server and realm on which it authenticates. Note that it is not necessary to use public key certificates with their overhead, as the key pair itself is sufficient for this scheme.

A server binds the Client Public Key (CPK) with a user's identifier (e.g., a user name). Of course, multiple Client Public Keys may be used by a user – one for every client (e.g., browser) he uses. As the computation of a digital signature is quite expensive operation, HOBA defines a way for servers to determine a challenge of sufficient length. In this scheme, the client has an option to renew the challenge at any time by fetching it from the server.

The exact name of this scheme used in requests and responses is "*HOBA*". A server must append the *challenge* and the *max-age* authentication parameter in the *WWW-Authenticate* header. The *challenge* parameter is a string that should be signed by the client in its request. To prevent *replay attacks*, its value must be unique for every 401 HTTP response. The *max-age* parameter specifies for how many seconds the *challenge* value is valid. If set to zero, then it is valid only for one signature. The *realm* parameter is optional. For the client, there is only one available parameter, the *result* parameter.

The HOBA scheme may be used with applications in the following way. The user is firstly redirected to a page with HOBA authentication. After he successfully authenticates, a new session cookie is set by the server. This cookie is used for further interactions with the application. This is a common way of session management today. There is however one security concern; cookies are basically equivalent to the bearer tokens that are weaker that the HOBA scheme.

General process of HOBA registration and authentication is not complicated. A client connects to a server and sends a request. The server responses with a *WWW-Authenticate* header that includes the HOBA authentication scheme together with parameters, such as the *challenge* parameter. If the client has never communicated with the server, a new key pair is generated by the client. The Client Public Key is sent to the server which attaches the key to the particular user name. The client then uses the parameters obtained in the server response to compute the *client-result* string which is sent in the next request as the *result* authentication parameter within the *Authorization* header. The server checks the value of the *result* parameter. If the value is valid (the signature can be verified by the client's public key stored on the server), a session cookie indicating that the client is authenticated is usually returned by the server.

Every client that uses HOBA authentication maintains pairs of host names (web origins) and realms. For every such pair, user's credentials, the key pair, are stored. Upon receiving a server response, the client combines the web origin, the realm, a value of the challenge parameter, the user name, and a client generated nonce and signs it with his private key corresponding to the particular Client Public Key. The structure that is signed by the client contains also lengths of the individual items to prevent ambiguity. The *client-result* string is a string compound of these parameters separated by a dot:

the user name, the server's challenge, the nonce, and the signature – all the parameters encoded using Base64URL encoding.

The following example demonstrates a communication between a client and a server. At the beginning, the used parameters are mentioned. The example was taken from the previously mentioned RFC 7486 [19]. The client requests some resource. The server responses with a 401 response, meaning that the client has to authenticate itself in order to obtain the resource from the server. The client finally authenticates itself using the HOBA authentication scheme.

```
Client's Public Key (maintained by the client and the server):
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAviE8fMrGIPZN9up94M28
6o38B99fsz5cUqYHXXJlnHIi6gGKjqLgn3P7n4snUSQswLExrkhSr0TPhRDuPH_t
fXLKLBbh17ofB7t7shnPKxmyZ69hCLbe7pB1HvaBzTxPC2KOqskDiDBOQ6-JLHQ8
egXB14W-641RQt0CsC5nXzo92kPCdV4NZ45MW0ws3twCIUDCH0nibIG9SorrBbCl
DPHQZS5Dk5pgS7P5hrAr634Zn4bzXhUnm7cON2x4rv83oqB3lRqjF4T9exEMyZBS
L26m5KbK860uSOKywI0xp4ymnHMc6Led5qfEMnJC9PEI90tIMcgdHrmdHC_vpldG
DQIDAQAB
-----END PUBLIC KEY-----

Origin:             https://example.com:443
User Name:          vesscamS2Kze4FFOg3e2UyCJPhuQ6_3_gzN-k_L6t3w
Server's Challenge: pUE77w0LylHypHKhBqAiQHuGC751GiOVv4/7pSlo9jc=
Signature Alg.:     RSA-SHA256
Client's Nonce:     Pm3yUW-sW5Q

Signature:
VD-0LGVBVEVjfq4xEd35FjnOrIqzJ2OQMx5w8E52dgVvxFD6R0ryEsHcD31ykh0i
4YIzIHXirx7bE4x9yP-9fMBCEwnHJsYwYQhfRpmScwAz-Ih1Hn4yORTb-U66miUz
q04ZgTHm4jAj45afU20wYpGXY2r3W-FRKc6J6Glv_zI_ROghERalxgXG-QVGZrKP
tG0V593Yf9IPnFSpLyW6fnxscCMWUA9T-4NjMdypI-Ze4HsC9J06tRTOunQdofr9
6ZJ2i9LE6uKSUDLCD2oeEeSEvUR--4OGtrgjzYysHZkdVSxAi7OoQBK34EUWg9kI
S13qQA43m4IMExkbApqrSg


Client Request:
     GET /resource HTTP/1.1
     Host: example.com

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: HOBA
        max-age=0,
        challenge="pUE77w0LylHypHKhBqAiQHuGC751GiOVv4/7pSlo9jc="
```

```
Client Request:
     GET /resource HTTP/1.1
     Host: example.com
     Authorization: HOBA
        result="vesscamS2Kze4FFOg3e2UyCJPhuQ6_3_gzN-k_L6t3w.pUE7
        7wOLylHypHKhBqAiQHuGC751GiOVv4/7pSlo9jc=.Pm3yUW-sW5Q.VD-
        OLGVBVEVjfq4xEd35FjnOrIqzJ2OQMx5w8E52dgVvxFD6ROryEsHcD31
        ykhOi4YIzIHXirx7bE4x9yP-9fMBCEwnHJsYwYQhfRpmScwAz-Ih1Hn4
        yORTb-U66miUzqO4ZgTHm4jAj45afU2OwYpGXY2r3W-FRKc6J6Glv_zI
        _ROghERalxgXG-QVGZrKPtGOV593Yf9IPnFSpLyW6fnxscCMWUA9T-4N
        jMdypI-Ze4HsC9JO6tRTOunQdofr96ZJ2i9LE6uKSUDLCD2oeEeSEvUR
        --4OGtrgjzYysHZkdVSxAi7OoQBK34EUWg9kIS13qQA43m4IMExkbApq
        rSg"

Server Response:
     HTTP/1.1 200 OK
     ...
```

The client has possibility to inform the server that it wishes to log out. To do so, it sends a POST request to a special, registered URL `".well-known/hoba/logout"`. In that case, the server should delete or invalidate the session cookie and the client should also delete it.

The client can request a new challenge from the server by sending a POST request to a registered URL `".well-known/hoba/getchal"`. If successful, the server sends a new HOBA challenge within the response body. This allows for pre-calculation of signatures and can help with making the user interface more responsive.

The HOBA scheme does not offer a way how to invalidate a key pair (how to delete the user's Client Public Key from the server). This option is useful when the user's private key gets disclosed or is stolen. This mechanism can be implemented at the application layer.

Registration of a new client must be secured, as registering own Client Public Key with someone else's account would be unacceptable. Thus, all the parameters, user names, and other values generated by the server must be infeasible to guess. This requires a good random generator.

If the *max-age* parameter has a non-zero value, *replay attacks* are possible within the time window specified by its value – the situation is very similar to the Digest and Bearer authentication. It is therefore wise to reduce the value of the *max-age* parameter as much as possible. More than a few minutes is usually not a justified choice. Servers can attempt to detect the *replay attacks* and react to them by sending a new challenge value within a 401 HTTP response.

As the HOBA scheme does not provide confidentiality or integrity of the message body and the header fields, it is highly recommended to use an additional mechanism, as the TLS protocol, together with this authentication scheme. It is important to protect the user name against attackers, as it usually does not change over time. Furthermore, if a session cookie is used, it is necessary to protect its value.

Another security consideration concerns a way how to store and manage the key pairs generated by the client. A user should have a possibility to set a certain amount of time after which a particular key pair is deleted by the client automatically or to delete the key pair anytime he decides to. Also, the keys have to be protected against other users in some appropriate way.

The *challenge* parameter must be infeasible to guess, should have at least 128 bits, and should be indistinguishable from a random string. As it is sent in the server response and returned by the client, there is no necessity to store the value on the server. However, in such a case, some reliable mechanism for checking the validity of the challenge has to be implemented. For example, the challenge may be an encrypted string of a specific format that is decrypted and checked by the server after being returned by the client.

1.3.5 Mutual Authentication

Mutual authentication is an authentication scheme that provides both the authentication of a client by a server and the authentication of a server by a client. This authentication protocol is password-based and assures the user that the particular server knows his encrypted password. A detailed specification can be found in the RFC 8120 [20]. It is still under examination.

This scheme has two important features that contribute to the overall security. Although this scheme is based on passwords, there is no password information exchanged during the communications. Thus, there is no possibility of sniffing the password (as in the case of the Basic authentication), *offline dictionary attacks* are prevented, and the user's password is not divulged even if some phishing website is used. The second feature is that the server also owns its valid credentials in order to successfully authenticate. Thus, the client can always check the identity of the server and the attacks performed using a counterfeit server are prevented.

There are three types of messages sent between the client and the server. First type includes messages that are sent by the server to instruct the client to start mutual authentication. These are called *authentication request messages.* Second type includes messages that are used by both peers for sharing of a cryptographic secret. These are called *authenticated key exchange messages.* The last group includes messages used by both peers for verification of the mutual authentication. These are called *authentication verification messages.* The messages are used in the order in which they are mentioned. Firstly, an *authentication request message* is used by the server to start the authentication protocol. Then, *authenticated key exchange messages* are used to perform authentication and to share the secret. Finally, *authentication verification messages* are used to verify the identity of both peers.

The exact name of this scheme used in requests and responses is "*Mutual*". The server uses two headers – the *WWW-Authenticate* header is used in responses with a 401 status code and the *Authentication-Info* header is used in all other responses. The client uses the *Authorization* header.

The *authentication request messages* are used by the server. They should always have a 401 status code and contain the *WWW-Authenticate* header with many available mandatory or obligatory authentication parameters. The *algorithm* parameter specifies the authentication algorithm to be used. Acceptable algorithms are defined in a separate RFC document – discrete-logarithm or elliptic-curve settings are used together with hash functions. The *validation* parameter specifies the method of host validation. Acceptable values are "*host*" for hostname validation, "*tls-server-end-point*" for TLS endpoint (certificate) validation, and "*tls-unique*" for TLS shared-key validation. The *realm* parameter is mandatory. The *reason* parameter describes a possible reason for the failed authentication. Description of some available parameters is omitted.

The *authenticated key exchange messages* used by the client contain the *Authorization* header. The *algorithm*, *validation*, and *realm* parameters must be the same as those received from the server. The mandatory *user* parameter contains a user name. The *kc1* parameter contains a client-side key exchange value which depends on the algorithm used.

The *authenticated key exchange messages* used by the server should have a 401 status code and contain the *WWW-Authenticate* header. The *algorithm*, *validation*, and *realm* parameters must be the same as those received from the client. A value of the *sid* parameter is a session identifier which is a random integer. The *ks1* parameter contains a server-side key exchange value which depends on the algorithm used.

The *authentication verification messages* used by the client contain the *Authorization* header. The *algorithm*, *validation*, and *realm* parameters must be the same as those received from the server. The *sid* parameter must contain the value obtained from the server for the particular realm. The *vkc* parameter contains a client-side authentication verification value which depends on the algorithm used.

The *authentication verification messages* used by the server contain the *Authentication-Info* header and do not have a 401 status code. The *sid* parameter must contain the value received from the client. The *vks* parameter contains a server-side authentication verification value which depends on the algorithm used.

The following example demonstrates a communication between a client and a server. The client requests some resource. The server responses with a 401 response, meaning that the client has to authenticate itself in order to obtain the resource from the server. The client then sends a message with an authenticated key exchange to start the authentication – the user name and the user password is used for that. The server then checks the user's credentials using its password database. If the credentials are valid, it generates a new session identifier (*sid*) and sends it together with a server-side authenticated key exchange value to the client. Both peers then compute a shared *session secret* using the authenticated key exchange values. These match only if both the server and the client use secret credentials generated from the same password. The *session secret* will be used in every subsequent request and response for access authentication. The client then sends a request with a client-side authentication verification value which is calculated from the *session secret*. The server checks its validity and if correct, the client is successfully authenticated. Otherwise, a 401 HTTP response is returned. Finally, the server sends its own authentication verification value and the client checks it. If it has some unexpected value or if the value is not present at all, the client must not process the rest of the response, as the response is most likely created by an attacker using the *man-in-the-middle attack* and a *phishing attack* is also possible.

```
Client Request:
     GET /resource HTTP/1.1
     Host: www.example.com

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: Mutual realm="example".
        ...

Client Request:
     GET /resource HTTP/1.1
     Host: www.example.com
     Authorization: Mutual user="john",
        kc1="...",
        ...

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: Mutual sid=...,
        ks1="...",
        ...

A "session secret" calculation

Client Request:
     GET /resource HTTP/1.1
     Host: www.example.com
     Authorization: Mutual sid=...,
        vkc="...",
        ...

Server Response:
     HTTP/1.1 200 OK
     Authentication-Info: Mutual vks="..."
     ...
```

There are also some alternative courses of the authentication.

- The client may omit the first request and immediately send the request containing the authenticated key exchange.
- If both peers previously shared a session secret, the client may directly send the request containing the authentication verification value. The server either accepts it or sends a 401 response, indicating that the key is expired and a new key exchange is required.

This protocol, as well as the protocols mentioned above, relies on security of the underlying layers. The TLS protocol should be used to provide data confidentiality and integrity. The protocol itself is secure against *passive eavesdropping* and *replay attacks*.

If the TLS server certificates are reliably verified, the *man-in-the-middle attacks* are prevented. Otherwise, there is a threat of rigging the mutually authenticated content by a JavaScript code (or other similar script) that is not authenticated by this mechanism.

As this is a password-based protocol, all the security concerns related to the passwords apply also to it. The user's passwords must not be disclosed. In contrast with the Basic authentication, the user's password is in this authentication protocol safe, even if an attacker uses a counterfeit server or a phishing web site. The mutual authentication further enables the detection of such servers and web pages.

The shared session secret also must not be disclosed, as this would lead to a possibility of *session hijacking attack* until the session expires.

There is also a possibility of *denial-of-service attacks*. The server maintains a table of active session. If there is some malicious client that sends many requests for the key exchange, the server creates many new sessions. Thus, some resources may get depleted. There are many solutions how to mitigate these attacks. For example, number of pending key exchange requests made by one client may be limited, a minimal time delay between two key exchange requests made by one client may be set, or a time limit for validity of the sessions may be set. However, the situation is better that in the case of the

Digest authentication; in this case, the session is created after the client sends out his user name and starts a key exchange.

As the password is not directly transported during the communication, it cannot be sniffed and obtained using an *offline dictionary attack*. Still, *active dictionary attacks* are possible. In these attacks, the attacker sends many authentication requests, every time trying a different password. The server may prevent these attacks by limiting the number or rate of unsuccessful authentication requests.

1.3.6 OAuth Authentication

As OAuth is rather an authorization protocol than an authentication protocol, its description can be found in the section called *Authorization*, Subsection 1.4.1.

1.3.7 Salted Challenge Response Authentication Mechanism (SCRAM)

Salted Challenge Response Authentication Mechanism is a family of authentication mechanisms that are still under examination. Their main purpose is to provide a more robust way of authentication than in the case of mechanisms transmitting a plaintext password[13] but with easier implementation and deployment than in the case of earlier challenge-response authentication mechanisms[14]. Specification can be found in the RFC 7804 [21].

This authentication mechanism has several advantages. Information stored on the server in the password database is not sufficient to impersonate a client and an attacker has to perform a *dictionary attack* to get the client's password. Furthermore, the information in the database is salted[15], so pre-computed password tables, called rainbow tables, are not applicable for an attacker. This authentication mechanism also supports mutual authentication and supports reauthentication requiring only one client request and one server response.

The exact name of this scheme used in requests and responses is a string "*SCRAM-*" followed by the name of the underlying hash function in uppercase. For example, "*SCRAM-SHA-256*" and "*SCRAM-SHA-1*" are valid authentication scheme names. All HTTP servers and clients should implement the "*SCRAM-SHA-256*" authentication mechanism for interoperability reasons.

Similarly, as in the case of the Mutual authentication, two headers are used by the server: the *WWW-Authenticate* header in responses with a 401

[13]For example, the Basic authentication

[14]For example, the Digest authentication

[15]A *salt* is random data that is in this case generated by the server. It is used as an input into a hash function together with the client's password. It serves as a defence against *dictionary attacks* and *pre-computed rainbow table attack*. Its value is also stored in the password database.

status code and *Authentication-Info* in other server responses. The *realm* parameter is optional and appears only in the first SCRAM server response and first SCRAM client request. The *data* parameter is a Base64-encoded string described below. The *sid* parameter is a unique session identifier.

The following example demonstrates a communication between a client and a server. It was taken from the previously mentioned RFC 7804 [21]. The client requests some resource. The server responses with a 401 response, meaning that the client has to authenticate itself in order to obtain the resource from the server. Two possible realms are proposed by the server. The client selects one of the realms and authenticates itself using the SCRAM authentication scheme with the *SHA-256* hash function. The server also authenticates itself in the last server response.

```
Client Request:
     GET /resource HTTP/1.1
     Host: server.example.com

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate:
        SCRAM-SHA-256 realm="realm3@example.com",
        SCRAM-SHA-256 realm="testrealm@example.com"

Client Request:
     GET /resource HTTP/1.1
     Host: server.example.com
     Authorization:
        SCRAM-SHA-256 realm="testrealm@example.com",
        data=biwsbj11c2VyLHI9ck9wck5HZndFYmVSV2diTkVrcU8K

Server Response:
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: SCRAM-SHA-256
        sid=AAAABBBBCCCCDDDD,
        data=cj1yT3ByTkdmd0ViZVJXZ2JORWtxTyVodllEcFdVYTJSYVRDQWZ
             1eEZJbGopaE5sRixzPVcyMlphSjBTTlk3c29Fc1VFamI2Z1E9PS
             xpPTQwOTYK

Client Request:
     GET /resource HTTP/1.1
     Host: server.example.com
     Authorization: SCRAM-SHA-256 sid=AAAABBBBCCCCDDDD,
        data=Yz1iaXdzLHI9ck9wck5HZndFYmVSV2diTkVrcU8laHZZRHBXVWE
             yUmFUQ0FmdXhGSWxqKWh0bEYscD1kSHpiWmFwV0lrNGpVaE4rVX
```

```
                    Rl0Xl0YWc5empmTUhnc3FtbWl6N0FuZFZRPQo=


Server Response:
       HTTP/1.1 200 OK
       Authentication-Info: sid=AAAABBBBCCCCDDDD,
           data=dj02cnJpVFJCaTIzV3BSUi93dHVwK21NaFVaVW4vZEI1bkxUSlJ
                  zamw5NUc0PQo=
       ...
```

The *data* parameter contains information necessary for the mutual authentication. Its value is a Base64-encoded string that is composed of several fields depending on the current step.

In the first value of the *data* parameter in the example above, the client sends the following encoded string:

```
n,,n=user,r=rOprNGfwEbeRWgbNEkqO
```

The first field "n" is a flag denoting that HTTP channel binding is not supported. The client's user name ("n=") and a random, unique nonce value ("r=") follows. The server responses with this encoded string:

```
r=rOprNGfwEbeRWgbNEkqO%hvYDpWUa2RaTCAfuxFIlj)hNlF,
s=W22ZaJ0SNY7soEsUEjb6gQ==,i=4096
```

The first field ("r=") contains a nonce value generated by the server and appended to the client's nonce. The client's salt ("s=") and a number of iterations ("i=") is also sent. These fields are necessary for further parameter generation. The client then sends this string to the server:

```
c=biws,r=rOprNGfwEbeRWgbNEkqO%hvYDpWUa2RaTCAfuxFIlj)hNlF,
p=dHzbZapWIk4jUhN+Ute9ytag9zjfMHgsqmmiz7AndVQ=
```

The first field ("c=") contains encoded information about the channel binding, the second field ("r=") has the same value as in the previous case, and the last field ("p=") contains the *client proof* that is necessary for the client authentication. The server verifies the *proof* and responds with a 200 response containing this encoded string:

```
v=6rriTRBi23WpRR/wtup+mMhUZUn/dB5nLTJRsjl95G4=
```

It contains only one field, the *server verifier* ("v="), that enables the client to authenticate the server. Exact formulas for computation of the *proof* and *verifier* fields can be found in the RFC document.

If an attacker is able to sniff messages (e.g., there is no HTTPS in use), then she is able to perform *offline dictionary* or *brute-force attacks* to recover the client's password. This may be problematic especially when some weak hash function, such as MD5, is used. The complexity of the password and increasing iteration count improves the security. A similar situation occurs if the password database is stolen. In such a case, the utilized *salt* makes the attacks more difficult – a separate attack is necessary for every single entry in the database.

As long as the used password, hash function, salt, or iteration count is unique across the servers, the original server is not able to impersonate any of his clients.

The selection of the used hash function is not directly specified. If possible, the client should choose the strongest available hash function to secure the authentication process and protect the confidentiality of his password as much as possible. On the other hand, the computational difficulty is also often taken into account.

There is a possibility of *denial-of-service attacks* on clients. A hostile server or another attacker able to modify messages can change the iteration count to some big value. That may lead to depletion of computing resources of the client. As a defence, the client may choose a maximum admissible iteration count and fail the authentication if the number supplied by the server is higher.

1.3.8 Voluntary Application Server Identification Authentication (VAPID)

VAPID authentication is a mechanism focused on authentication of an application server to a push service. Today, many applications use Web Push protocol to deliver messages to a user automatically, without being requested by the user. This behaviour can be used for example for notifications. In the Web Push protocol, three different roles are used: an application server, a push service, and a client (a user agent or an application). An application server can send a message, called a push message, at any time to a push service that ensures reliable delivery to the target client. The VAPID authentication protocol was defined in the RFC 8292 [22] and enables the application servers to authenticate themselves to the push services to better distinguish between legitimate and bogus traffic.

The server identity has many applications. The push service can use the stable identity of the application server to create a behavioural model and to detect deviations from it. A software error or an attack may cause a significant deviation – in this case, it may be desirable to inform an application server administrator about the current situation. The server identity can be also used for message priority determination. The clients are able to subscribe to some application servers using their identifiers, while not to the others. Besides the server identity, some additional information, such as contact information, can be provided by the application server.

In this authentication method, the application server needs to generate and store a single key pair that can be used for the Elliptic Curve Digital Signature

Algorithm. When sending messages to the push service, the server includes a *JSON Web Token* that is signed using the private key. The token includes information about the origin address and expiration time. The validity period must not be greater than 24 hours. If the signature or some parameter in the token is invalid, the push service may reject the request (a 403 (Forbidden) status code should be returned) and the information from the request must not be used.

The exact name of this scheme used within the *Authorization* header is "*vapid*". There are two possible authentication parameters. The *t* parameter contains the signed token which is composed of three Base64-encoded parts separated by a dot character – a token header, a token payload (body), and the signature. The *k* parameter contains the corresponding Base64-encoded public key of the application server in uncompressed form that serves as a stable identifier of the server.

The following example demonstrates a message sent by an application sever to a push service. It was taken from the RFC document mentioned above.

```
Application Server Request:
     POST /p/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
     Host: push.example.net
     TTL: 30
     Content-Length: 136
     Content-Encoding: aes128gcm
     Authorization: vapid
        t=eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9.eyJhdWQiOiJodHRwc
          zovL3B1c2guZXhhbXBsZS5uZXQiLCJleHAiOjE0NTM1MjM3NjgsInN
          1YiI6Im1haWx0bzpwdXNoQGV4YW1wbGUuY29tIn0.i3CYb7t4xfxCD
          quptFOepC9GAu_HLGkMlMuCGSK2rpiUfnK9ojFwDXb1JrErtmysazN
          jjvW2L9OkSSHzvoD1oA,
        k=BA1Hxzyi1RUM1b5wjxsn7nGxAszw2u61m164i3MrAIxHF6YK5h4SDY
          ic-dRuU_RCPCfA5aq9ojSwk5Y2EmClBPs

     { encrypted push message }
```

As mentioned before, the *t* parameter consists of three parts separated by a dot character – a token header, a token payload, and a signature. The encoded and decoded first two parts can be found here:

```
Token Header Encoded:
     eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9
Token Header Decoded:
     {"typ":"JWT","alg":"ES256"}

Token Payload Encoded:
     eyJhdWQiOiJodHRwczovL3B1c2guZXhhbXBsZS5uZXQiLCJleHAiOjE0NTM
```

```
    1MjM3NjgsInN1YiI6Im1haWx0bzpwdXNoQGV4YW1wbGUuY29tIn0
Token Payload Decoded:
    {"aud":"https://push.example.net","exp":1453523768,
    "sub":"mailto:push@example.com"}
```

In the Web Push protocol, URI addresses of the push service intended for receiving requests from application servers should stay secret. The VAPID authentication protocol adds another layer of security – the push message is processed only if it includes a valid signed token.

There is a possibility of *replay attacks* if an attacker gains a token that is still valid. Thus, the HTTPS protocol should be used with this authentication method to provide confidentiality and to prevent *eavesdropping*. The restricted validity period of the token reduces the impact of similar attacks.

The contact information provided by application servers is not proven and may be falsified before inserted into the token. It is important to keep it in mind when handling some critical security issues.

As signature validation is a computationally intensive operation, there is a threat of *denial-of-service attacks*. It is recommended that application servers reuse tokens which permits the push service to utilize a cache of already validated signatures.

1.3.9 Form-based Authentication

Form-based authentication is a technique used on the Internet that makes use of a web form into which the user enters his credentials. Actually, this non-standardized technique generally does not use the HTTP authentication framework at all and is not an HTTP authentication scheme. However, it is probably the most common authentication technique used on the Internet today. According to some sources, over 90% of web applications use this mechanism [23]. That is the reason why it is mentioned in this thesis.

In general, the term "*form-based authentication*" refers to any authentication technique that uses a form for credentials, no matter how the credentials are transmitted. For example, they can be transmitted using the HTTP *Authorization* header and Basic authentication scheme. More specifically, the term "*form-based authentication*" is used for an HTML form that sends user's credentials in a message body using the HTTP protocol and POST method. In the following description, the term "*form-based authentication*" is used in this sense.

The whole process consists of several steps. First, a user requests some resource via the HTTP protocol and because he is not authenticated, he is redirected to a login page containing the login form. Second, the user fills in his credentials and sends them to the server. The credentials are sent in the plaintext within the POST request body – usually as POST parameters or within some structure, such as a JSON structure. Finally, the server verifies the credentials and if valid, creates a new session identifier which is transmitted in the form of an HTTP cookie. This cookie identifies the user in the subsequent requests. Usually, the user is redirected back to the requested resource.

The following example demonstrates a communication between a client and a server during the whole process. POST parameters are used for the transmission of the credentials.

```
Client Request:
     GET /resource.html HTTP/1.1
     Host: www.example.com

Server Response:
     HTTP/1.1 302 Found
     Location: https://www.example.com/loginpage.html

Client Request:
     GET /loginpage.html HTTP/1.1
     Host: www.example.com

Server Response:
     HTTP/1.1 200 OK

     ...
     <form method="post" action="login">
       <input type="text" name="username">
       <input type="password" name="password">
       <input type="submit" value="Login">
     </form>
     ...

Client Request:
     POST /login HTTP/1.1
     Host: www.example.com

     username=user123&password=mypassword123

Server Response:
     HTTP/1.1 302 Found
     Location: https://www.example.com/resource.html
     Set-Cookie: SESSIONID=3ab58263c9275fa4274bc927d2694b64;
        Expires=Fri, 13 Apr 2018 21:18:00 GMT; Secure; HttpOnly

Client Request:
     GET /resource.html HTTP/1.1
     Host: www.example.com
     Cookie: SESSIONID=3ab58263c9275fa4274bc927d2694b64

Server Response:
     HTTP/1.1 200 OK

     ...
```

This authentication method is not much secure, as it transmits user's credentials in the plaintext and its security completely depends on the security of the underlying layers. Thus, HTTPS should be always employed when using this method. A *phishing attack* may be used by an attacker to steal user's credentials.

1.4 Authorization

Authorization is a process that verifies whether some user has access to some resource or whether he is permitted to perform some operation (for example, deleting or modifying some resource). Authorization should not be confused with *access control* which is a process of enforcing defined security rules for a particular resource.

There are not many authorization protocols used in the HTTP protocol today. The reason is that the most common process for authorization in the Internet does not use the HTTP protocol directly. Instead, some authentication method is used to verify the user's identity. The identity itself is then used to determine whether the user is authorized to perform an action or request a resource. The user is usually identified by user's credentials contained in the *Authorization* header or by a session identifier. For more information about session identifiers, refer to the Section 1.5.

The OAuth protocol, which is used for third-party access authorization, is described in the following subsections.

1.4.1 OAuth 1.0 Authorization

In OAuth 1.0 protocol, three different roles are used: a client, a server, and a resource owner. This protocol provides a method for a resource owner to authorize clients to access his protected resources on a server using user-agent redirections and without sharing his credentials. The clients are thus able to access resources on behalf of their owners. Although the OAuth protocol is primarily used for authorization, it can be also used for authentication – in addition to verification of the resource owner authorization, the server can verify identity of the clients. The OAuth protocol version 1.0 was defined in the RFC 5849 [24]. The exact name of this scheme used in requests and responses is "*OAuth*".

The whole process is illustrated by the following example. A user of a social network, let's call him Jim, shares some photos on this network. Jim has the role of the resource owner and the photos are the protected resources. The social network site, hosted on `"photos.example.net"`, has the role of the server. Jim would like to use a photo editing web application hosted on `"app.example.com"`, the client, to edit one of the photos he shares on the social network. Jim thus needs the web application to access his photo on the network but without giving it his credentials.

The photo editing web application has the option to load photos from the social network. In order to do that, the application owns its own *client credentials* – `"ky8js72h4kd956ds"` is its identifier and `"83hwgf2u59d745fg"` is its password – and is configured to be able to use interface offered by the social network.

The social network uses `"HMAC-SHA1"` algorithm as its signature method and offers these URI addresses for communication with other applications:

- `"https://photos.example.net/initiate"` – this URI address is used by other applications to ask for *temporary credentials*

- `"https://photos.example.net/authorize"` – this URI address is used to get authorization from the resource owner

- `"https://photos.example.net/token"` – this URI address is used for requesting a set of *token credentials* using the *temporary credentials*

The client, the photo editing application, first needs to obtain a set of *temporary credentials* from the server, the social network, to identify the request for access delegation. To do so, the client sends the following request to the server:

```
GET /initiate HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
   oauth_consumer_key="ky8js72h4kd956ds",
   oauth_signature_method="HMAC-SHA1",
   oauth_timestamp="1523110363",
   oauth_nonce="fTjrTp",
   oauth_callback="http%3A%2F%2Fapp.example.com%2Fready",
   oauth_signature="JgzmHF5%2B8i0fLzAPf%2F8mXkK2Qw8%3D"
```

After the request is validated by the server, a set of *temporary credentials* is returned.

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded

oauth_token=sifd4j6jerhp4j89&oauth_token_secret=
aperj5nfyc32hdio&oauth_callback_confirmed=true
```

Jim's browser is then redirected by the client to a server page that requires an approval from the resource owner, Jim. The URI address of this page is the following:

```
https://photos.example.net/authorize?oauth_token=
   sifd4j6jerhp4j89
```

Jim then has to sign in the social network using his credentials. If the login procedure passes off without any problems, Jim is asked to grant the photo editing web application "`app.example.com`" access to his photos. If he approves that, his browser is redirected to the URI address provided by the client ("`oauth_callback`" parameter).

```
http://app.example.com/ready?oauth_token=
   sifd4j6jerhp4j89&oauth_verifier=ht5eavcno07hfery
```

The server informs the client in this way that the resource owner granted it the permission to access his photos. The client then needs to ask the server for a set of *token credentials* using its *temporary credentials*.

```
GET /token HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
   oauth_consumer_key="ky8js72h4kd956ds",
   oauth_token="sifd4j6jerhp4j89",
   oauth_signature_method="HMAC-SHA1",
   oauth_timestamp="1523110364",
   oauth_nonce="rJ7qn5P",
   oauth_verifier="ht5eavcno07hfery",
   oauth_signature="fk%2BycE1TUcRHdOEv40VkjNU7cr0%3D"
```

After the request is validated, the server returns a set of *token credentials*.

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded

oauth_token=kre73jt065kshq9y&
oauth_token_secret=to6xiqmpv86j4ick
```

With these credentials, the client is now ready to access the Jim's photo.

```
GET /photos?file=JimsPhoto.jpg&size=original HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
   oauth_consumer_key="ky8js72h4kd956ds",
   oauth_token="kre73jt065kshq9y",
   oauth_signature_method="HMAC-SHA1",
   oauth_timestamp="1523110365",
   oauth_nonce="fuwpGe",
   oauth_signature="tqFNFaYBgsPHAHg7cGNo%2FZUlUJ4%3D"
```

After the server validates the client request, it returns the photo that was requested. The client is able to reuse these *token credentials*, until Jim revokes the access or signs out of the social network.

In this protocol, three classes of credentials are used:

- *client credentials* – they identify and authenticate the client that makes requests; they are used throughout the whole process
- *temporary credentials* – they are used to identify the access request through the authorization process; they must be revoked after being used once to obtain the *token credentials* and it is recommended to limit their validity period
- *token credentials* – they are issued after the resource owner grants access rights to the client and identify the access grant itself; the server should enable the resource owner to revoke these credentials; they can have a restricted scope or a limited validity period

The whole process based on redirections in the browser of the resource owner is summarized here:

1. The client uses his *client credentials* to request a set of *temporary credentials* that identify the access request in the following steps.
2. The resource owner gives permission to the server to grant access to the client. The client is identified by the *temporary credentials.*
3. After the access is granted to the client, it uses its *temporary credentials* to ask the server for a set of *token credentials.*
4. The client uses the obtained *token credentials* to access resources of the resource owner.

If the value of the *oauth_timestamp* parameter is too old, the server may reject the request. The *token credentials* may be used by another client than the one to which the *token credentials* were issued. The sever can however prohibit such behaviour. The validity period of the *token credentials* may be restricted.

The value of the *nonce* parameter is a unique[16] random string generated by the client. The server uses its value to prevent *replay attacks* by verifying that the request has never been made before. Restriction of the time period after which a request with an old timestamp is rejected decreases the number of nonces that must be remembered by the server.[17]

The *oauth_signature* parameter enables the server to verify that the client knows both the user name and the password and thus to verify the identity of the client. There are three *signature methods* defined in OAuth 1.0 authorization protocol – "*HMAC-SHA1*", "*RSA-SHA1*", and "*PLAINTEXT*". The "*RSA-SHA1*" method utilizes RSA keys instead of the passwords and the "*PLAINTEXT*" method actually does not compute any signature. Servers may implement their own custom methods.

The value of the *oauth_signature* parameter is computed depending on the selected *signature method*. If the "*PLAINTEXT*" method is chosen, then this parameter is set to the value in this form: the encoded client password concatenated with an "&" character concatenated with the encoded token password (that is part of the *temporary* or *token credentials*). As the passwords are sent in the plaintext for this method, the security considerations are analogical to those for Basic authentication. The TLS protocol must be used to protect them. If one of the remaining methods is chosen, the value of the *oauth_signature* parameter is computed in this way:

1. The following HTTP request elements are concatenated:
   - The used HTTP request method (e.g., "*GET*", "*POST*")
   - An "&" character
   - The base URI address (e.g., "*https://photos.example.net/initiate*") – without the query and anchor part, the port number is included only if some non-standard number is used
   - An "&" character
   - The request parameters – they are collected from the request[18], decoded to their original form, sorted, encoded, and finally concatenated into a single string
2. If the "*HMAC-SHA1*" method is used, then hash-based message authentication code algorithm[19] in combination with SHA-1 hash function is

[16]Unique across all requests with the same client credentials, token credentials, and timestamp

[17]In such a case, it is important for the both peers, the client and the server, to have the time synchronized with each other.

[18]The *GET* parameters, *Authorization* header parameters (except for the *realm* parameter), and the *POST* parameters are collected.

[19]Defined in RFC 2104

applied – as an input to the function, the string created in the first step is used, and as a key, concatenated passwords[20] are used

3. If the "*RSA-SHA1*" method is used, then the string created in the first step is signed using the client's private RSA key using RSASSA-PKCS1-v1_5 signature algorithm as defined in the RFC 3447 document

4. The result computed in step 2 or step 3, depending on the method chosen, is encoded using the Base64 algorithm and finally used as the value of the *oauth_signature* parameter

All the authorization parameters used by this protocol can be transmitted either in the *Authorization* request header or in the request body or in the request URI query, as in the case of the Bearer authentication. For security concerns related to putting parameters into the URI query, refer to the Section 1.3.2.

This protocol is designed to partially protect the integrity of requests, especially the parameters sent in messages. However, the header fields and some other message parts are not protected. Furthermore, the confidentiality of requests is not ensured at all. Thus, it is necessary to use this protocol together with the TLS protocol to protect sensitive contents and potentially the client's passwords.

In the case of the "*RSA-SHA1*" method, there are no *temporary* or *token passwords* in use and requests are signed using only the client's private RSA key. It is thus essential to protect this key against its disclosure. In the case of the "*HMAC-SHA1*" and "*PLAINTEXT*" methods, the server need to store clients' passwords in the plaintext in order to compute the signatures. If an attacker gains access to the password database, she will be able to perform any action under an arbitrary client's account. Therefore, the password database must be protected from unauthorized access. If the "*PLAINTEXT*" method is chosen and there is no HTTPS in use, an attacker is able to sniff the passwords in the plaintext. If the "*HMAC-SHA1*" method is chosen and there is no HTTPS in use, then an attacker is able to sniff the signatures and run *offline dictionary* or *brute-force attacks* to recover the passwords from the signature value. To prevent such attacks, servers should generate passwords that are long enough and random enough.

If the client is, for example, some desktop application that is free to download, there is a threat that an attacker will obtain the client's credentials from the executable file using *reverse engineering* methods or from the source code if available.

In contrast with the Mutual authentication, the server identity is not verified in this protocol. An attacker is able to use some counterfeit server to intercept client requests and return some deceptive responses. Additionally,

[20]The same string as in the case of the "*PLAINTEXT*" method

if the "*PLAINTEXT*" method is used by the client, the attacker will obtain the client's password.

As the server needs to track some parameters, such as *nonce* values, *denial-of-service attacks* are possible. The server can be also incited to perform some expensive computations, such as signatures verification, to dissipate its resources.

There is a possibility of *cross-site request forgery attacks* on the URI addresses offered by the server for communication with other applications if cookies are used to identify the request owner. Also, the client's callback URI address is at risk. It is thus important to implement methods of prevention against these attacks on both the server and the client. For example, CAPTCHA test may be used or reentry of resource owner's credentials may be enforced. The resource owner should be always informed to which client he is going to grant the access permission. Another possible attack is the *clickjacking attack*. If the response headers are set appropriately, the attack should be prevented.

Although there are no possible attacks feasible for an ordinary attacker today, the SHA-1 hash function is considered broken or at least insecure and is being replaced by newer and more secure hash functions. As the OAuth 1.0 protocol supports any custom signature methods, it is possible to implement a more secure way of signing requests (for example, using the SHA-2 or SHA-3 hash functions).

Because this authorization scheme suffers from several security weaknesses and is impractical in many ways, it is considered obsolete and should be replaced by the OAuth 2.0 authorization framework defined in the RFC 6749 document [25] and described below.

1.4.2 OAuth 2.0 Authorization

The OAuth 2.0 protocol is not backward compatible with OAuth 1.0, as it is a completely new protocol. However, the overall architecture has been preserved. Its definition can be found in the RFC 6749 [25] and a nice overview was written by Aaron Parecki [26]. The Bearer authentication is a part of the OAuth 2.0 framework.

The OAuth 2.0 framework was designed to remedy the shortcomings of the first version of the protocol. The main differences are listed below:

- OAuth 2.0 offers new scenarios, called *authorization flows.*

  In OAuth 1.0, desktop and mobile applications had to open a browser and redirect a resource owner to a server's web page in order to get his approval. Further, they had to copy the resulting token from the browser back to the application.

In OAuth 2.0, new flows were added which target not only the web applications but also desktop applications and applications for mobile phones and living room devices.

If the resource owner, in OAuth 2.0 called user, trusts the client, he can entrust it his credentials, the user name and the password, which are then used directly by the client. This flow is useful, for example, if the user wants to quickly write a script for his personal use only. He is no longer forced to install a library to handle all the OAuth 1.0 processes.

Another flow is the following: a client may use its own credentials to request an access token to access its own resources – not on behalf of a resource owner.

- In the OAuth 1.0 protocol, servers had to maintain session information between client requests. First, the client asked for *temporary credentials* using its *client credentials*. Second, the client redirected the resource owner in order to obtain his approval. Finally, the client asked the server for *token credentials* using its *temporary credentials*. This caused that the protocol was difficult to scale to large systems.

  In the OAuth 2.0 protocol, the server uses the *client credentials* only if the client obtains authorization from the resource owner. After that, the client uses only the resulting access token.

- The Bearer authentication based on existing cookie architecture was introduced. Its description can be found in the Subsection 1.3.2. Only one secret token is used instead of the *token credentials* that were composed of a public and private part. The bearer tokens also do not require signing of each request – there is no necessity to parse, decode, sort, and encode parameters. There is also no HMAC in use.

- With OAuth 2.0, the role of the authorization server is separated from the server offering API to clients. The authorization server is only responsible for obtaining authorization from resource owners and issuing tokens to clients. The API server uses only the access tokens for authentication. Two different physical servers may be used to implement the two roles.

- In OAuth 2.0, the authorization server usually issues an access token with a short validity period along with a long-lived refresh token. The client may thus obtain new access tokens without requesting a resource owner permission again. Tokens can be also easily revoked.

1.5 Session Management

An HTTP session (also called a web session) is a sequence of HTTP request and response network transactions that are associated to one particular user. Sessions are typically temporary and expire after some time. Usually, their validity is extended with each new request from a client and a fixed maximum overall validity period is set that cannot be exceeded.

Many web applications need to maintain some state information about each user, such as language preference or application settings, for longer duration. Sessions provide the ability to establish variables and to use them for every interaction of the user with the web application during the session validity. Sessions are also used for maintaining information about user's identity which is verified during the authentication process. Based on this identity, user's privileges are checked to ensure that the authorization is handled correctly.

It is a common practice to use a random string, called a *session identifier*, for identification of the session. It should be long enough to be infeasible to guess it[21]. The session identifier is assigned at session creation time, is exchanged between the server and the client and is used instead of using the session information directly. When the server receives a request with the session identifier, it looks up associated session information in its database using the identifier as a key.

HTTP is a stateless protocol – that means that every request and response pair is independent of the other. Therefore, it is necessary to implement session management that links both the authentication and the authorization modules available in the web application. In the HTTP protocol, this is achieved by using cookies. State management mechanism in the HTTP protocol is specified in the RFC 6265 [11].

HTTP cookie can be seen as a pair consisting of a name and a value along with some additional metadata. This metadata is used to determine whether a particular cookie should be used in a specific request and includes flags mentioned in the Subsection 1.1.1. A cookie can be limited to a specific server, URI address, and time period.

As mentioned before, a server inserts the *Set-Cookie* header into its response to establish a new cookie. This header may be sent in any response (with an arbitrary status code). Furthermore, multiple *Set-Cookie* headers can be used in a single response if they set cookies with different names. The format of the header is the following:

```
Set-Cookie: cookie-name=cookie-value(; attribute-name)*
   (; attribute-name=attrribute-value)*
```

[21]OWASP recommends using identifiers with 128 bits or more.

There are several standardized attributes for this header. Some of them contain a value, others are used only as flags. Each attribute should be used at most once. All standardized attributes are listed below:

- *Expires* – this attribute contains a value – a date and time at which the cookie expires

  If set to a date and time in the past, a cookie with identical name, path, and domain should be removed from the browser.

- *Max-Age* – this attribute contains a value – a number of seconds until the cookie expires

- *Domain* – this attribute contains a value – hosts to which the cookie will be sent

  If set to `"example.com"`, the cookie will be sent when making requests to `"example.com"`, `"www.example.com"`, `"www.corp.example.com"`, etc. The specified value must include the origin server. If the attribute is missing, the cookie should be sent only to the origin server (and not, for example, to any subdomain).

- *Path* – this attribute contains a value – a set of paths to which the cookie is limited (including their subdirectories)

  If the attribute is not specified, the browser should use the directory in the URI address of the corresponding request. This attribute should not be used for security decisions.

- *Secure* – this attribute does not contain any value – the cookie should be sent only over secure channels (e.g., HTTPS channels)

  Protects only the confidentiality of the cookie, not its integrity.

- *HttpOnly* – this attribute does not contain any value – the cookie is used only for HTTP requests, not for any “non-HTTP” APIs

If both the *Max-Age* and the *Expires* attributes are set for a single cookie, the *Max-Age* attribute has precedence. If neither the *Max-Age* nor the *Expires* attribute is set, the cookie is considered to be a *session cookie* and deleted after the current session is over (usually when the browser shuts down).

If a new cookie with the same name, domain, and path is set by the server, the browser should replace the original cookie with this new one.

The browser uses the *Cookie* header in a request to inform the server about stored cookies. Only non-expired, applicable cookies are appended to the request. Cookies set by a server response with 100-level status code may be ignored. The format of the header is the following:

```
Cookie: cookie-name=cookie-value(; cookie-name=
   cookie-value)*
```

The header can contain two cookies with the same name, if they were set with different path or domain. A single request must contain the *Cookie* header at most once.

The following example demonstrates how the previously mentioned headers are used.

```
Client Request:
     GET /resource1.html HTTP/1.1
     Host: www.example.com

Server Response:
     HTTP/1.1 200 OK
     Set-Cookie: cookie1=value1; Domain=example.com;
        Expires=Fri, 13 Apr 2018 21:18:00 GMT; Secure; HttpOnly

     ...

Client Request:
     GET /resource2.html HTTP/1.1
     Host: www.example.com
     Cookie: cookie1=value1

Server Response:
     HTTP/1.1 200 OK
     Set-Cookie: cookie1=newvalue1; Domain=example.com;
        Expires=Fri, 13 Apr 2018 21:18:30 GMT
     Set-Cookie: cookie2=value2; Domain=www.example.com

     ...

Client Request:
     GET /resource3.html HTTP/1.1
     Host: www.example.com
     Cookie: cookie1=newvalue1; cookie2=value2

Client Request:
     GET /resource4.html HTTP/1.1
     Host: subdomain.example.com
     Cookie: cookie1=newvalue1
```

There are many security risks associated with cookies [23]. If cookies are used for user authentication, it is possible to perform a *cross-site request forgery attack* to require some actions on a server on behalf of a user. An attacker usually tricks the user to load some malicious page prepared by the attacker. This can be achieved by using a *phishing attack*. After the malicious page is loaded, a specially crafted request to a target host is made (for example, using a JavaScript code, HTTP redirection, or HTML form). The browser automatically attaches cookies intended for the target server to the request. If there is a valid session identifier in a cookie, the target server authenticates the request and performs an action specified by the request – the action is performed on behalf of the user without knowing it. This attack can be used, for example, to change user's password in the target application or to transfer money in internet banking. There are many mitigation techniques – it is recommended to use so-called "*anti-CSRF tokens*" that cannot be guessed by the attacker and if missing, the requested action should not be performed. The server can also require a CAPTCHA test or reentry of user's credentials before any sensitive operation is performed. The experimental *SameSite* flag should prevent the browser from sending cookies within cross-site requests.

To protect confidentiality of cookies, it is necessary to use some secure channel, such as the HTTPS channel. In such a case, it is further important to set the *Secure* flag for every cookie. Unless using a secure channel, the cookies are transferred in the cleartext and can be sniffed by an attacker. The server can also encrypt and sign the content of every cookie in order to protect it.

If the session identifier is stored in a cookie, there is a possibility of the *session fixation attack*. In this attack, an attacker inserts her own value into the cookie containing the identifier[22]. Then, the attacker waits until a victim signs in to the target application on a server using the planted identifier value, as cookies are attached automatically to requests. If the server accepts the identifier received from the browser and connects it with the victim's session, the attacker at this moment knows its exact value and can use it to interact with the server directly. She can, for example, obtain some private information about the victim or perform some actions using his identity. This could be a big security issue especially for computers used by multiple users (for example, in public places). Thus, it is recommended that servers always generate a new session identifier after the user signs in to an application.

Another risk associated with storing session identifiers in cookies are the *session hijacking attacks*[23] where an attacker is able to impersonate a victim in the web application. The attacker can perform a *cross-site scripting (XSS) attack* or use a physical access to a victim's computer in order to obtain a valid session identifier. After that, she is able to spoof the victim's identity. This is dangerous especially when the session cookie has a long validity period which is a common practice for internet stores and other online services today.

Another possible security issue is that cookies cannot be limited to a particular port. Thus, all services running on the same server having the same

[22]All common browsers allow users to modify cookies

[23]Actually, the *session fixation attack* also belongs to this group of attacks.

host name (despite the different port numbers) have access to the same cookies. They obtain them within client requests and they can modify them. The same situation happens also for different schemes, such as the *http* and *ftp* scheme. Some browsers also fail to respect the *Path* attribute in some unusual situations – for example, when JavaScript `document.cookie` function is called.

If there are two different subdomains of one domain, such as `"aaa.example.com"` and `"bbb.example.com"`, some problems may occur. For example, the `"aaa.example.com"` server can set a cookie with the domain set to `"example.com"` and overwrite thus a cookie previously established by the `"bbb.example.com"`. This new cookie will be used in interactions with the `"bbb.example.com"` which can lead to some security implications. Similarly for the *Path* attribute, an application running on the `"www.example.com/app1"` may set the path of a cookie to an arbitrary value – for example, `"/app2"`.

An active network attacker can impersonate a response from an HTTP server (e.g., `"http://www.example.com/"`) and inject the *Set-Cookie* header into the message. The cookie set in this way is then used by the client when sending requests to the legitimate HTTPS server (e.g., `"https://www.example.com/"`). This attack may be mitigated by encrypting and signing the contents of the cookies. However, there is still a possibility to replay a cookie obtained in another way.

Another possible security issue for cookies are *DNS spoofing attacks*. If the DNS server is compromised, the session credentials may be disclosed to a malicious server.

There is also a possibility of *denial-of-service attacks* when sending a big amount of requests to create new sessions to one server.

So-called third-party cookies are used when loading resources from other servers. They are typically used to track a user, for example, for advertising purposes. This is more a privacy problem than a security issue.

Besides the session management defined by the HTTP protocol and described above, there are other ways how to pass on session information and session identifier. The session identifier can also be inserted into a URL address as a POST parameter. However, this approach suffers from many security issues. An unacquainted user can copy the whole URL address (including POST parameters) from a browser and send it to someone else without knowing that he just disclosed his private session identifier. Furthermore, URL addresses are logged in the browser and on the server. If an attacker has access to the browser history or server log file, she is able to gather all such session identifiers.

Session information should always be stored on a server or should have such a format so they cannot be forged by a client or an attacker.

1.6 Summary

Many authentication schemes were specified within the HTTP protocol. All of them rely on security of the underlying layers to provide confidentiality and integrity of HTTP messages. The use of the TLS protocol or another secure channel is especially important for schemes that transfer user credentials in the plaintext. The Basic authentication is such example.

The Digest authentication scheme makes use of hash functions to protect the passwords. However, this scheme is not easy to implement and is not widely used.

The Bearer authentication scheme is usually used within the OAuth 2.0 authorization process. It makes use of access tokens, called bearer tokens, that are generated by a server. These tokens are usually used instead of credentials to access a resource on behalf of its owner.

The Mutual authentication scheme was designed to offer the authentication of a server. Although this scheme is based on passwords, they are not sent directly over the communication channel. This scheme is still experimental.

In general, the password-based methods suffer from several weaknesses, as users tend to chose weak passwords and reuse them. The HOBA authentication scheme aims to prevent such problems by using digital signatures. This scheme is still under examination.

Another experimental authentication scheme is the SCRAM authentication scheme which also allows authentication of a server. Its advantage is that the information stored on a server is not sufficient to impersonate a client.

The VAPID authentication scheme is focused on authentication of an application server to a push service within the Web Push protocol. Thus, it is not used for web applications.

Probably the most common authentication method in use on the Internet today is the form-based authentication. Although this method is not standardized and does not use the HTTP authentication framework, it provides website owners with fine-grained control over the authentication process and with a possibility to customize the design of the login form.

The HTTP protocol offers two authorization protocols – the OAuth 1.0 and OAuth 2.0. These are used for third-party access authorization. A resource owner can grant access to his resources to another parties without revealing his own credentials. The OAuth 2.0 framework was designed to remedy the shortcomings of the OAuth 1.0.

The HTTP cookies are used to provide session management in the HTTP protocol. They can be exploited for many serious attacks. However, if the web application is secured appropriately and the server configured correctly, they should provide secure session management.

Chapter 2

Burp Suite

This chapter describes the Burp Suite tool, some of its features and how this tool addresses authentication, authorization, and session management.

Burp Suite is a graphical tool for security testing of web applications. It records requests and responses between a browser and servers. The recorded requests can be then modified and resent. It offers several tools, each for another purpose. For example, the *Spider* is a tool for automatic crawling of web applications, the *Scanner* is a tool for automatic vulnerability scanning of web applications[24], and the *Intruder* is a tool for fuzzing requests. Its main advantages are the low price, automatic scanning tool capable of detecting up to 100 different vulnerabilities, easy extensibility, and a number of available customization options. These are the possible reasons why this tool has become one of the most popular web application security tools among security professionals worldwide. The Burp Suite documentation can be found online [27].

Burp Suite automatically stores cookies that were issued by web sites in the *cookie jar*. Users have an option to set which tools will use these cookies when issuing requests. However, there is only one *cookie jar* shared between all the tools. Thus, it is possible to maintain only one session at the same time and all threads used for scanning or fuzzing use this single session. This can lead to errors for applications which have an upper limit on the number of concurrent requests per one session. We tried to solve this limitation in the practical part of this thesis.

Burp Suite offers a feature called *Macros*. A macro is a sequence of one or more requests that is defined by a user. They can be used to perform various tasks, such as a login to the application to obtain a new session, obtaining a token used in another request, or fetching a page to check whether the current session is valid. Macros can be launched by *session handling rules*.

[24]The *Scanner* tool finds all possible input points within a request and updates them with a list of potentially dangerous inputs one by one. The tool assesses possible vulnerabilities using the corresponding responses.

A *session handling rule* consists of two parts – a *scope* and a list of actions. The *scope* defines for which requests will be the actions performed. It can be decided based on the tool that is issuing the request, the URL address of the request, or names of parameters within the request. These session handling rules can be used, for example, to check the current session and if not valid, running a macro to log back into the application. It is also possible to use it for obtaining of a valid anti-CSRF token and inserting its value into a request that requires it.

There is a feature called *Match and replace*. It can be used to automatically replace parts of requests and responses – for example, for updating some header values, values of tokens, etc. Unfortunately, this functionality only works for messages going through the *Proxy* tool (messages going from a browser or a desktop application to a server and back). We mitigated this restriction in the practical part of this thesis by implementing a similar functionality that works for requests issued by any tool.

A big advantage of the Burp Suite tool is its easy extensibility. Users can create their own extensions and share them in the official *BApp* store. Extensions published in the official store can be downloaded very easily directly from Burp Suite. They can be written in Java, Python, or Ruby, and Burp Suite offers a plenty of functions that can be used by the extensions, called Burp Suite API. Users can thus customize the default behaviour in numerous ways, such as modifying HTTP requests and responses, customizing the user interface, and adding custom checks to the *Scanner*.

2.1 Authentication, Authorization, and Session Management in Burp Suite

The Burp Suite has support for several types of authentication. In the *User options/Connections* tab, there is a section called *Platform authentication* which lets users configure the following types of authentication: the HTTP Basic, HTTP Digest, and NTLM authentication. The inserted credentials are then used to automatically carry out platform authentication to destination web servers.

For simple login forms, it is possible to specify user credentials in the *Application Login* section in the *Spider/Options* tab. However, this settings may be only used for crawling of a web application (using the *Spider* tool).

In the case of more complex login forms, it is necessary to create a macro that handles user authentication. A session handling rule must be used to launch the macro to authenticate the user automatically. Although the process of setting up the macro and the rule can be complicated, it works quite well in many cases and if configured properly. The macros also allow to copy some parameter values from previous macro responses to subsequent ones. However, only cookies, POST parameters, and GET parameters can be updated in this way. The same restriction also applies to the session handling rules and parameter update using them.

The Burp Suite currently does not support update of request headers by session handling rules. Users can partially solve this issue by chaining a second instance of Burp after the first one and configuring a match and replace rule in the *Proxy* tool. Another solution is to create an own extension in order to handle tokens within headers. This issue is especially problematic for bearer tokens used in the Bearer authentication scheme for third-party authorization. A similar situation also applies to other HTTP authentication schemes. In the practical part of this thesis, we implemented an extension that mitigates this limitation.

As for the session management, the Burp Suite tool offers many options. It uses the cookie jar to track application cookies. Session handling rules are then used to update the cookie values in requests. It is possible to choose which Burp tools should update the cookie jar and which requests should be updated with the stored cookie values. Is it also possible to use macros and session handling rules for automatic creation of a new session if it expires. A significant limitation is that only one cookie jar exists and only one session can be maintained at the same time. Additional information about session-handling support in the Burp Suite can be found in The Web Application Hacker's Handbook [23] or in the official documentation [27].

CHAPTER 3

Design

A practical part of this master's thesis was to create an extension to the Burp Suite tool. This chapter deals with the design of the extension. First, functional and non-functional requirements are mentioned. Second, some basic concepts used by the extension are explained. Some parts of this and subsequent chapters have been taken from a documentation published on the Internet [28].

3.1 Goals and Requirements

The main goal of the extension is to simplify testing of web applications using Burp Suite. On the official web site, there are hundreds of requests for enhancement of the tool. Many of them are related to session management, authentication, and authorization in some way. Several of the requests ask for a possibility to maintain multiple different sessions at the same time. Another request asks for a feature similar to the *Match and replace* that would work for all the tools, not only the *Proxy* tool. There is also a request to be able to use values extracted from previous messages in the *Match and replace*. Many of the requests are quite old, some dating back to 2015. All of these features have not been added yet. Typically, there is some proposed workaround to solve the problem. However, that is usually quite complicated and impractical. A regular advice to maintain multiple sessions at the same time is to use several instances of Burp Suite or to create your own extension. We tried to create an extension that would implement such features that are missing in Burp Suite and that are requested by Burp Suite users.

The key feature of the extension is to support the most common methods for authentication, authorization, and session management on the Internet today. It includes the cookie-based and URL-based session management, the form-based authentication, the Basic authentication, and the Bearer authentication. The Digest authentication as well as the experimental authentication schemes are not so important to us, as they are not commonly used. The

Authentication Scheme	**Standardized**	**Supported**	**Reason**
Basic Authentication	Yes	Yes	
Bearer Authentication	Yes	Yes	
Digest Authentication	Yes	No	Not widely used
HOBA Authentication	Yes	No	Experimental protocol
Mutual Authentication	Yes	No	Experimental protocol
SCRAM Authentication	Yes	No	Experimental protocol
VAPID Authentication	Yes	No	Used mainly for Web Push
Form-based Authentication	No	Yes	The most common scheme on the Internet

Table 3.1: Table of described authentication schemes and whether they are supported by the extension

VAPID authentication is not used for web applications; therefore, it is not the subject of our interest. An overview of the described authentication schemes may be found in the Table 3.1.

An important criterion is to make the program easy to use. For example, a possibility to export already made settings and import them later is a crucial feature, as the initial setup usually takes a lot of time.

Functional requirements are listed below:

- A possibility to handle tokens used for the most common methods for authentication, authorization, and session management:
  - The form-based authentication
  - The Basic authentication
  - The Bearer authentication
  - The cookie-based session management
  - The URL-based session management
- Maintaining multiple sessions at the same time
- A possibility to define when a session is expired

- A possibility to create new sessions
- A possibility to export user settings to a file and to import such a file

The first functional requirement implies a necessity to parse and update tokens within cookies, URL addresses, headers, GET parameters, and POST parameters.

Non-functional requirements are listed below:

- Files with exported settings are in a human-readable and easily editable format

3.2 Basic Concepts

This subsection describes basic concepts used in the created extension. The aim of this section is to explain the main characteristics of the concepts and to introduce the reader to terms used in the following chapters.

- *Variable* – Variables are unique identifiers that interconnect all structures used in the extension. Sometimes, it is also understood to be a pair composed of the unique identifier and a corresponding value. They are used in Rules, Sessions, Default Parameters, and Session Expiration Rules. A Variable identifies a record in a Session, that is used for extracting tokens from responses and updating them in requests. Variables are also used for specification of default values and rules for Session expiration.

- *Session* – A Session is a database or a set of pairs consisting of a Variable and a corresponding value. It keeps all the tokens extracted from responses that can be later used in requests for authentication, session management, etc. Actually, it is possible to use the Session to store almost anything that can be extracted from responses – a status code of the last response within the Session, values of response headers, values of arbitrary cookies, URL address of the corresponding request, etc. Which parts of responses should be extracted into the Session and where should be the values used within requests is defined by Rules.

  There may be multiple Sessions used by this extension and the number of Sessions is automatically increased according to the current need. If a new request comes and there is no free Session, a new one is created – a Burp macro defined by the user is launched and tokens are extracted from the macro responses. The Sessions are independent of each other.

- *Rule* – There are two types of Rules – Response Rules and Request Rules.

Response Rules define which parts of responses should be extracted and in which Variables within the Session should the extracted values be stored. If the Variable is not present in the Session, it is created.

Request Rules define which parts of requests should be updated and which Variables (more precisely, values of these Variables) should be used for the update. If there is no such Variable, a Default Parameter is used. And if there is no such Default Parameter, the Request Rule is omitted.

- *Default Parameter* – A Default Parameter is a pair composed of a Variable name and some fixed value (a string). Default Parameters are used if there is some Request Rule that should update a request with value of some Variable but this Variable is not present in the Session (was not extracted yet or there is no corresponding Response Rule that extracts its value). In such a case, the Default Parameter is used if it exists. If there is no Default Parameter with such Variable name, the Request Rule is omitted. The value of a Default Parameter can be an empty string.
- *Session Expiration Rule* – Session Expiration Rules are rules that define when a Session is not valid anymore – validity of a token expired, user was logged out, etc. Session Expiration Rules have this form: "*If a value of some Variable equals to some string/contains some string/matches some regular expression, then the Session should be invalidated.*" If at least one Session Expiration Rule applies, the Session is considered as expired and is thrown away. There can be more Session Expiration Rules for one particular Variable.

There may be multiple Response and Request Rules associated with one Variable. It is possible to create several Response Rules that extract some value from a response and store it in the same Variable. For example, it is possible to extract a value either from a cookie or from a message body – the last one found will be stored in the corresponding Session. Further, it is possible to use a value of one Variable in multiple places in a request. For example, it is possible to update either a cookie or a POST parameter (or both at once) with the value of the Variable.

Chapter 4

Realisation and Usage

Burp Suite enables several ways how the extension can cooperate with the tool itself. Our extension implements an interface called "*IHttpListener*". This interface causes that the extension is notified about outgoing requests and incoming responses made by any Burp tool. The intercepted responses are used for extraction of tokens, and the requests are updated with the extracted values.

4.1 Implementation

It was necessary to make several implementation decisions. The first decision concerned the choice of the programming language. Burp Suite offers three options – Java, Python, and Ruby. As Burp Suite itself is written in Java and Java extensions do not require any additional environment[25], it was decided to use Java. Furthermore, there may be some memory issues when using extensions written in Python or Ruby.

Besides the ability to parse and update tokens within cookies, URL addresses, headers, GET parameters, and POST parameters, we decided to support updating and parsing of JSON message bodies, XML message bodies, and we also offer a general solution for parsing and updating almost anything in requests and responses using regular expressions (so-called General Request and Response Rules). Thanks to this, the extension offers many additional options that were not specified in the assignment but we believe will be useful. For example, it enables to handle anti-CSRF tokens.

We decided not to support a dynamic computation of values in the first version of the program, as it is not used by the most common methods of authentication, authorization, and session management. Still, we believe it would be a nice feature to add in the future.

[25]In order to use a Python extension, you need to download a Python interpreter implemented in Java, called Jython. For Ruby, you need to download JRuby.

We looked for some solution that would allow us to create new Sessions whenever it is necessary. Burp Suite macros appeared to be a good candidate for login operations. Although there were some problems we had to solve, we finally decided to use them, as Burp Suite users are already familiar with them.

XML was chosen as a format of exported files. It is both human-readable and machine-readable. It is also easily editable and is supported by many text editors. In most browsers, XML files are rendered in an organized and highly readable way – elements can be collapsed, etc[26]. People familiar with HTML language usually do not have problems with understanding XML files. We also believe that the XML format is much more readable even for non-technical people than the JSON format. Furthermore, XML format was a choice for several popular Java serialization libraries. Its main disadvantage is its verbosity. However, the length is not a critical parameter in the case of settings files that are not primarily intended for sending over a network.

During the export, all Rules, Default Parameters, Session Expiration Rules, and additional settings created by a user are saved. The Sessions are not exported. The main reason is that they would be anyway after some time with high probability expired. Also, the macro used for Session creation is not exported. However, it is usually stored in the Burp Suite itself.

For parsing JSON message bodies, a Google library called *gson* [29] was chosen. It is developed by a large, international company, it has an open source code, and it is still an active project. Its main advantage is that it is easy to use.

For handling XML message bodies and XML files, we decided to use the *DOMParser* library (the *javax.xml* package) that is integrated in Java by default. Thus, there is no need for including an external library to the extension.

As there may be several requests and responses at the same time, it commonly happens that the input function to the extension is called several times simultaneously. It was thus necessary to perform all the sensitive operations in a thread-safe manner. We chose to use Java `synchronized` keyword to create *synchronized statements.*

To handle several sessions at the same time, we created a new class called *SessionManager*. It maintains two structures – a pool of free, available Sessions and a hash table of reserved Sessions. A more detailed description of the class can be found in the Subsection 4.1.2. Again, it was critical to perform all operations involving the two structures in a thread-safe manner.

It was necessary to create several graphical elements to allow users to enter their Rules, Default Parameters, and more. The extension has its own tab within the Burp Suite window. Some additional tabs were created within

[26]This was one of the reasons why JSON format was not chosen. Further, JSON is mostly used with JavaScript, and JSON files usually contain more lines of code than the XML ones when formatted. However, JSON is usually shorter than XML in number of characters or bytes.

the main tab to handle different types of Rules and other settings. Also, several separate windows are used (e.g., for creation of Rules). The interface of the extension was designed in order to make it as uncluttered and organized as possible. Great emphasis was placed on ease of use.

4.1.1 Encountered Problems

During the design and implementation of the extension, we encountered several problems. We hit the limits of the available Burp Suite API several times.

First, a macro cannot be launched by an extension directly. So, we specified the following URL address which is requested by the extension and which indicates that the macro for Session creation defined by the user should be created. This requires this URL address to be in scope of the macro – users have to set it up.

```
http://localhost:80/launch-burp-macro-122333
```

The overall process for creation of a new Session begins if a new request comes to the extension and there is no free Session available. In such a situation, the extension creates a request that is sent to the URL address mentioned above. If the user macro is set properly, it should be launched. The extension extracts and updates values within the macro messages. The new Session is made up of the extracted values after the macro is over.

Another complication with macros is that the extension is not able to recognize the requests and responses issued by the macro. This is a serious inconvenience, as we need to use the extracted values from the macro responses to create the new Session. Thus, it is necessary to insert the following header into all macro requests – again, the user must arrange this:

```
AuthenticationMaster: launch-burp-macro-122333
```

If this header is present in an incoming request, the extension recognizes that this is a macro request and uses it (and the corresponding response) for the creation of the new Session. The header is removed during request processing and is not sent to any server – it should not be exposed anywhere.

The last serious issue was how to recognize the corresponding requests and responses. When a request comes, it is assigned some free Session and its Variables are used for update of the request. After the request is processed, the assigned Session is freed by the request and reserved for the corresponding response. After the response comes, it gets the same Session as was assigned to the corresponding request and some values are possibly extracted. After the response is processed, the Session is freed and can be reused. To perform all the described actions, we need to recognize to which request a response belongs. The Burp Suite API offers a function that returns an identifier of the pair – the request and the response. Unfortunately, this function is available only for messages going through the *Proxy* tool – requests coming from a browser and corresponding responses. Thus, it is not useful for our extension. Furthermore, the object holding the message pair also alters and cannot be used for this purpose. The only unchanging and unique object is the whole request string (under certain circumstances). Thus, we decided to use the Java `hashCode` called on the request string, as it is quite fast and sufficient for our purposes. To use these hashes reliably, our extension must be the last extension that modifies requests. To ensure that, the user needs to put the extension to the last position in the list in *Extender/Extensions* tab in the Burp Suite.

Although these workarounds work without any problems, they can be annoying for users. Unfortunately, we were not able to find better solutions to these problems, even after discussions with Burp Suite employees. However, these workarounds do not affect servers, as the custom header is removed by our extension during the request processing. If the extension is turned on, the header should never leave the tool.

4.1.2 Classes

This subsection provides an overview of all implemented classes along with their brief descriptions. The classes are organized into several packages.

- *ActiveSessionsSnapshot* – This class is used to copy the state of all Sessions at a certain time. The information is then visible in the graphical interface.

- *BurpExtender* – This class is used to interconnect the Burp Suite with the created extension and implements several interfaces defined by the Burp API. The *IBurpExtender* interface must be implemented by all extensions. The *IHttpListener* interface is used to notify the extension of all requests and responses. When the *IExtensionStateListener* interface is implemented, the extension will be notified when the extension is unloaded – this can be used for cleanup actions, saving state of the extension, etc. This class contains an input point to the extension. If a new message comes to this class and the extension is enabled, the message is passed to the *MessageProcessor* class.

- *DefaultParameters* – This class stores default values of Variables. These values are used if there is some Request Rule performed for some request and there is no relevant Variable in the assigned Session.

- *MessageProcessor* – This class performs the main processing of requests and responses. First, a new Session is demanded from the *SessionManager*. Second, all relevant Rules are collected. Then, the message is parsed and the Rules are applied. Finally, the Session is returned to the *SessionManager* and the message is completed and returned to the Burp Suite.

- *RulesDictionary* – This class maintains all Rules created by a user. Depending on the URL address of the message, the list of relevant Request or Response Rules is returned.

- *SerializationHandler* – This class is used to export user settings to a file and import user settings from a file.

- *SessionDB* – This class implements the Session, as described above. It stores a set of Variables and the extracted values.

- *SessionExpirationRules* – This class stores Session Expiration Rules that were created by a user. It is used to verify that a Session is still valid and did not expire (according to the Session Expiration Rules).

- *SessionManager* – This class manages Sessions. It maintains two main data structures – a pool of free Sessions and a list of reserved Sessions that were already used by a request and are waiting for the corresponding response. If a new request comes, the *SessionManager* assigns it some free Session from the pool. If there is no free Session, a new one is created. After the request is processed, the Session is returned to the *SessionManager*, placed into the list of reserved Sessions and waits for the corresponding response. When the response comes, it obtains the Session that was reserved for it. After the response is processed, the Session is returned back to the *SessionManager* and is released for use by a new request (it is placed into the pool of free Sessions).

- *Classes for Rules* – The base class, called *Rule*, implements all features common to all Rules. It also declares some abstract methods. There are two other abstract classes that inherit from the *Rule* class – the *ReqRule* and *ResRule* classes that correspond to the Request and Response Rules respectively. From these abstract classes are inherited non-abstract, final classes that can be instantiated. The final classes implement specific types of Rules – for example, the *ReqRuleCookie* class implements Request Rules for handling cookies and the *ResRuleHeader* class implements Response Rules for handling headers. The final classes offer methods for extracting and updating tokens within messages. It is possible to restrict a Rule to certain URL addresses.

  In the case of Request Rules, it is possible to set some prefix and suffix that will be appended to the value of a Variable when applying the Rule (updating a request).

  In the case of Response Rules, it is possible to set some regular expression that will be used for further extraction of the value. For example, you can create a Rule that extracts content of some header and then to set the regular expression that extracts only the first word of the content.

- *Classes for graphical interface* – There are many classes for creation of new tabs, windows, forms, etc. The Java *Swing* library was used.

4.1.3 Overall Process

This subsection briefly describes steps of processing requests and responses. The aim of this description is to present the main process performed by the extension.

The following list describes how a request is processed.

1. A new request comes to the *BurpExtender*.
2. If the extension is enabled and the request comes from a Burp tool that is allowed, it is sent to the *MessageProcessor*; otherwise, the request is not processed.
3. If only items in scope are set to be processed and the URL address of the request is from the scope set in Burp Suite, it is processed; otherwise, the request is not processed.
4. A new Session is assigned to the request. If there is no free Session, a new one is created.
5. Then, all the Request Rules that are relevant to the particular URL address are collected.
6. The collected Rules are applied one by one – the request is updated with values in the Session (alternatively Default Parameters). First, cookies, URL address, GET parameters, and POST parameters are updated. Second, General Rules are applied. Then, Headers are updated. Finally, XML and JSON parameters are updated.
7. The assigned Session is reserved for the corresponding response – it is waiting until the corresponding response comes. To recognize that the response belongs to the request, a hash of the request body is used – Burp Suite does not offer any unique identifier for the pairs of requests and responses.
8. The message is completed and returned.

The following list describes how a response is processed.

1. A new response comes to the *BurpExtender*.

2. If the extension is enabled and the response comes from a Burp tool that is allowed, it is sent to the *MessageProcessor*; otherwise, the response is not processed.

3. If only items in scope are set to be processed and the URL address of the original request is from the scope set in Burp Suite, it is processed; otherwise, the response is not processed.

4. The Session that was reserved by the request is given to the response. If there is no such Session, the response is not processed. Possible reasons for a missing reserved Session are the following: the extension has just been enabled and the corresponding request was not processed, another extension changed the request (and also its hash), settings of the extension were changed, etc.

5. Then, all the Response Rules that are relevant to the particular URL address are collected.

6. The collected Rules are applied one by one – values are extracted from the response and inserted into the Session. Order is not fixed.

7. The assigned Session is freed and can be used by another request.

4.2 Usage

To add the extension into the Burp Suite, it is necessary to open the *Extender/Extensions* tab in the Burp Suite and click on the *Add* button. A new window appears. It is required to insert a path to a jar file with the extension and click on the *Next* button. Output of the extension printed out during the load appears, as well as possible load errors. If there is no error on the output, the extension should be loaded successfully and a new tab with the extension should appear. As mentioned before, it is necessary to place the extension in the list of extensions so that it is the last one that modifies requests. Now, it is possible to set up the extension or load some configuration file in the new tab.

4.2.1 Use Cases

The created extension can be used, for example, for handling different tokens within requests and responses. However, there are many other possible uses. Some examples are described below.

Use Case 1 – The Basic authentication scheme (described in the Subsection 1.3.1)
The extension can be, for example, used to add the *Authorization* header with the Basic scheme and some fixed credentials to every request going to

some particular server. First, it is necessary to create a new Request Rule that will insert the *Authorization* header. Then, it is necessary to set the authentication scheme and the credentials. To do that, we can set the prefix to `"Basic "`. Let us assume that the Variable used by this Rules is called `"credentials"`. Finally, we need to set the value of the Variable. We can create a new Default Parameter `credentials="dXNlcjpwYXNzd29yZA=="`. The `"dXNlcjpwYXNzd29yZA=="` string is a Base64-encoded string `"user:password"`.

Use Case 2 – The Bearer authentication scheme (described in the Subsection 1.3.2)
To handle Bearer tokens, it is necessary to set up a Response Rule for token extraction depending on how the token is sent to us – we can use the JSON Response Rule, General Response Rule, etc. Let us call the Variable for extraction `"BearerToken"`. Then, we can create a Header Request Rule that will add the *Authorization* header to particular requests. We can restrict the URL addresses for which the Rule should be applied. We have to set up the prefix to `"Bearer "` and to use the extracted value of the `"BearerToken"` Variable for update.

Use Case 3 – The form-based authentication scheme (described in the Subsection 1.3.9
The extension can be used to handle form-based authentication. It is necessary to create Default Parameters for the user name and password and update their values in a login request using an appropriate Request Rules. Further, we have to create a Cookie Response Rule to extract the session identifier from a cookie to some Variable. We have to use a Cookie Request Rule to update the cookie in requests using the extracted value.

Use Case 4 – The anti-CSRF tokens handling
The extension can help manage anti-CSRF tokens. A General Response Rule can be used for extraction of the token from an arbitrary part of a response. A Request Rule for POST parameters can be used to update the value of the token within the request body. Both Rules have to use the same Variable name.

Use Case 5 – The URL address modification
The extension can also be applied if a URL address of some application changes. It is possible to change the protocol, host name, port number, path, or any other part of the URL address using regular expressions. If the host name changed, for example, from `"www.example.com"` to `"www.new.example.com"`, it is possible to redirect all requests to the new host name. It is also possible to change some directory – URL addresses beginning with `"www.example.com/old/..."` can be changed to `"www.example.com/new/..."`. Furthermore, the protocol and the port number can also be changed. Thus, it is

possible to redirect all HTTP requests to use the HTTPS protocol if available on the server.

This list of use cases is not exhaustive – the extension can be used for many other purposes that are not mentioned in this subsection. We also plan to implement some additional features to extend its capabilities. For example, it would be beneficial to add a possibility to compute Variable values dynamically. That would allow users to handle tokens used for the Digest authentication and some other experimental authentication schemes. Further, a possibility to parse HTML responses would be useful. That would allow users to use selectors for token extraction (similarly, as in the case of XML responses and the XPath language).

New versions of the extension can be found on the official page [28], as well as updated information about the extension.

Chapter 5

Testing and Evaluation

This chapter describes how the extension was tested, what are its main features, and what are differences between our and other similar extensions.

5.1 Testing

The extension was developed and tested mainly on the computer Lenovo Thinkpad W530 244744G with Intel Core i7-3720QM, 16 GB of RAM memory, and a hard disk drive, running 64-bit operating system Windows 10 Enterprise. However, some other colleagues also participated in the testing with their own devices. Because Java is a cross-platform in the sense that compiled Java programs run on all platforms with a Java virtual machine, there should not be any problems with using the extension on many different devices.

During testing of the extension, two main criteria were set – the proper functioning of the implemented functionalities and the speed. The functionalities were tested continuously on several different applications. After a new functionality had been added, it was manually tested to work as expected. As for the Request Rules, it was necessary to use some extension for recording requests and responses[27] to check whether the requests were modified by the Rules as expected. As for the Response Rules, the correctness of the extracted values was checked using the extension text output. Later, a new window was used which shows current Variable values of all valid Sessions. All the implemented features were tested and work as expected.

Several colleagues participated in the testing and review of the extension and provided a valuable feedback. Some comments were already reflected and some new features were added based on the provided feedback. For example, a new feature for manual deletion of a Session was implemented and built into the extension.

[27]The *Logger++* and *Flow* extensions can be used for this purpose.

Number of session	**Average scan time [s]**	**Request rate [requests/s]**
1	2,072	1.39
5	343	8.42
10	157	18.34
20	65	44.18

Table 5.1: Measured times and request rates

Regarding the speed of the extension, handling of tokens requires some additional overhead. Further, it is necessary to consider the time it takes to create new Sessions at the beginning of every active scan or fuzzing. However, these additional time requirements are negligible[28] and more than compensated by the ability of using multiple sessions at the same time.

The Burp Suite *Scanner* tool offers an option to set a number of concurrent requests for the scanning. For many applications (especially the ones with anti-CSRF and similar tokens), it is not possible to use more than one concurrent request if the tester wants to process the tokens correctly and achieve the most accurate possible results, as Burp Suite can maintain only one session at the moment. Therefore, testers cannot utilize this option in many cases. Our extension allows them to use multiple sessions at the same time and thus to fully utilize this option. For each of the concurrent requests, another Session is used. In the following paragraphs, we will interchange the terms "number of concurrent requests" and "number of sessions". During the testing of the extension, we came to the following finding: if using multiple concurrent requests during an active scan, the speed-up is super-linear. This trend is reflected in our measurements.

For a scan of one particular request we needed 2,072 seconds using a single session[29]. Average number of requests per second was 1.39. For a scan of the same request using 5 threads, we needed 343 seconds on average and the average number of requests per second was 8.42. For a scan using 10 threads, we needed 157 seconds on average and the average number of requests per second was 18.34. When we used 20 threads, we needed 65 seconds on average and the average number of requests per second was 44.18. Summary of the measured times and request rates can be found in the table 5.1.

As we can see from the values that are plotted in the graph 5.1, managing of multiple sessions greatly speeds up scanning of requests and the whole application in general. The super-linear speed-up was not disrupted by our

[28] Scanning and fuzzing time depends primarily on the speed of the connection. Processing on the local computer has quite a little impact on the overall time.

[29] Scanning of requests takes quite a long time, especially if the requests contain many parameters that should be tested. The request scan we performed took more than 34 minutes using a single session. If we would need to scan the whole application that usually contains tens or hundreds of such requests using a single session, it would take days.

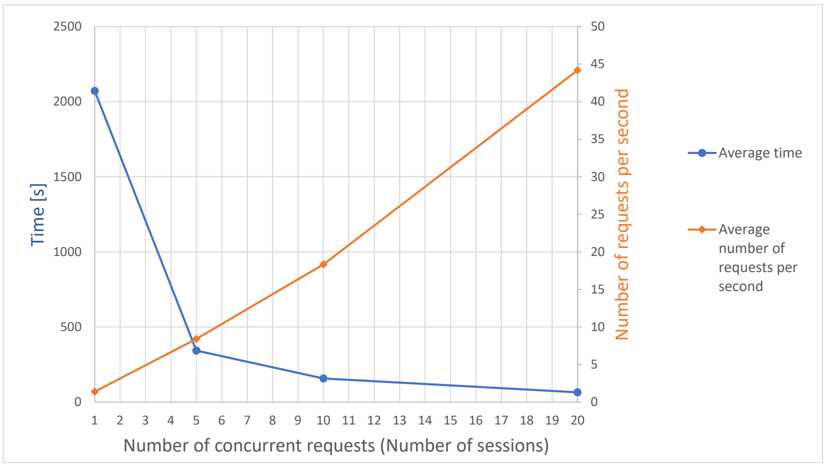

Figure 5.1: Graph of measured times and request rates

extension and even synchronization of multiple threads and greater network congestion for 20 concurrent requests did not reverse the trend. We believe that this extension will be helpful to many testers and will save their time.

5.2 Features

Our extension meets all the requirements specified in the assignment and it is also suitable for other purposes. Its main features are listed below:

- Handling of the following parts of requests and responses:
  - Cookies – extraction and update of their content
  - Headers – extraction and update of their content
  - URL address – extraction and update of the URL address
  - GET and POST parameters – update of their content
  - JSON message body – extraction and update of the JSON structure
  - XML message body – extraction and update of the XML structure using the XPath language
  - Other parts – extraction and update of other parts of messages using a regular expression

This can be used to manage authentication, authorization, and session tokens, as well as anti-CSRF tokens. Further, it is possible to automatically resend requests with a URL address different from the original one.

- A possibility to use fixed values of tokens that cannot be extracted from a response (Default Parameters) – this feature can be used for example for storing testing credentials
- Maintaining multiple sessions at the same time
- A possibility to define when a session is expired based on the extracted values from responses
- A possibility to create new sessions using Burp Suite macros
- A possibility to export and import user settings to an XML file
- An organized graphical interface that is user-friendly and easy to use

Our extension makes the token handling significantly easer. It also offers additional features that are not available in the Burp Suite tool. Using our extension, the time necessary for web application testing may get shorten several times.

5.3 Comparison with Other Extensions

There are not many extensions that have similar features as the ours. The Burp Suite itself does not support handling of multiple sessions at the same time and token management is far more complicated to set up.

There is an extension called *TokenJar* that handles anti-CSRF tokens, session identifiers, and other tokens. However, it does not provide so many options as our extension, does not allow to maintain multiple sessions at the same time and does not offer automatic creation of a new session. This extension has one big advantage – it allows to compute parameters dynamically using JavaScript; thus, it is possible to use this extension for generation of random tokens, etc. It also preserves its state automatically.

There are several extensions that handle anti-CSRF tokens, such as the *CSRF Token Tracker*, *CSurfer*, and *Match/Replace Session Action.* There are also some specialized extensions for handling bearer tokens[30]. However, features of these extensions are usually very limited.

At the time of writing this thesis, we are not aware of any other similar extension.

[30]For example, the *UpdateToken* and *BearerAuthToken* are both available on the GitHub page. These two extensions are not available in the official BApp store and must be downloaded and loaded to the Burp Suite manually.

Conclusion

The aim of this thesis was to find out which authentication, authorization, and session management methods are used in the HTTP protocol and to analyse their security aspects and weaknesses. There are many available methods, especially for authentication. Some of them enable authentication of the server, while others do not. In some cases, credentials are transferred in the plaintext which brings additional security threats. None of the methods is completely secure – each has its own weaknesses that can be used for an attack, and these weaknesses were described in this master's thesis. All the studied methods also rely on security of the underlying layers to provide confidentiality and integrity of HTTP messages and in many cases to protect the credentials themselves. Thus, these methods should always be used over a secure channel (e.g., using the HTTPS protocol).

Within the practical part of this thesis, an extension to the Burp Suite tool has been created. It meets all the specified requirements and provides many additional functionalities. Its main feature is easy management of authentication, authorization, and other tokens contained in HTTP messages. Many options are offered to simplify the set-up of the extension. Further, the extension allows testers to maintain multiple sessions at the same time and thus to speed up web application testing several times in some cases. A large number of Burp Suite users ask for adding similar feature into the Burp Suite itself, yet without success. According to information obtained from an employee of the company which develops the tool, adding such feature is not planned in the near future. As far as we know, there is also no comparable extension available at the moment. Thus, we hope that this extension will be useful for many people. At least, until a similar functionality is released as a part of the Burp Suite itself. All the predefined requirements have been fulfilled and the extension was successfully tested.

In terms of future plans, we would like to publish the source code of the extension on the GitHub page. Other users will be thus able to further improve the extension and customize it. We also plan to include the extension in the official Burp Suite store. This store contains dozens of such extensions and users can install them directly from the Burp Suite tool. It is possible that the extension will be officially introduced in September at the OWASP meeting held in Prague.

In the meantime, there are many useful functionalities that could be added into the extension. For example, a possibility to calculate values of tokens dynamically using code snippets. This would enable users to handle tokens used within Digest authentication scheme and multiple experimental schemes. It would also allow users to use the current time for the token computation, to concatenate multiple values into one token, etc. Further, we believe it would be useful to add a possibility to parse HTML responses. Finally, automatic backup of the settings would be beneficial in case of any program failures. We plan to implement all the mentioned enhancements to make the extension even more useful and handy, and we would like to continue developing and supporting the extension in the future.

Bibliography

[1] FIELDING, R. et al. *RFC 2068 – Hypertext Transfer Protocol – HTTP/1.1* [online]. 1997 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc2068`.

[2] FIELDING, R. et al. *RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1* [online]. 1999 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc2616`.

[3] FIELDING, R.; RESCHKE, J. *RFC 7230 – Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing* [online]. 2014 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc7230`.

[4] FIELDING, R.; RESCHKE, J. *RFC 7231 – Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [online]. 2014 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc7231`.

[5] FIELDING, R.; RESCHKE, J. *RFC 7232 – Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests* [online]. 2014 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc7232`.

[6] FIELDING, R.; LAFON, Y. and RESCHKE, J. *RFC 7233 – Hypertext Transfer Protocol (HTTP/1.1): Range Requests* [online]. 2014 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc7233`.

[7] FIELDING, R.; NOTTINGHAM, M. and RESCHKE, J. *RFC 7234 – Hypertext Transfer Protocol (HTTP/1.1): Caching* [online]. 2014 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc7234`.

[8] FIELDING, R.; RESCHKE, J. *RFC 7235 – Hypertext Transfer Protocol (HTTP/1.1): Authentication* [online]. 2014 [viewed 23 February 2018]. Available from: `https://tools.ietf.org/html/rfc7235`.

[9] *HTTP* [online]. 2017 [viewed 23 February 2018]. Available from: `https://developer.mozilla.org/en-US/docs/Web/HTTP`.

[10] *Test HTTP Methods (OTG-CONFIG-006)* [online]. 2015 [viewed 24 February 2018]. Available from: `https://www.owasp.org/index.php/Test_HTTP_Methods_(OTG-CONFIG-006)`.

[11] BARTH, A.; BERKELEY, U.C. *RFC 6265 – HTTP State Management Mechanism* [online]. 2011 [viewed 10 March 2018]. Available from: `https://tools.ietf.org/html/rfc6265`.

[12] DIERKS, T.; RESCORLA, E. *RFC 5246 – The Transport Layer Security (TLS) Protocol Version 1.2* [online]. 2008 [viewed 10 March 2018]. Available from: `https://tools.ietf.org/html/rfc5246`.

[13] SHEFFER, Y.; HOLZ, R. and SAINT-ANDRE, P. *RFC 7457 – Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)* [online]. 2015 [viewed 10 March 2018]. Available from: `https://tools.ietf.org/html/rfc7457`.

[14] GREEN, M. *A Few Thoughts on Cryptographic Engineering* [online]. 2018 [viewed 16 March 2018]. Available from: `https://blog.cryptographyengineering.com/`.

[15] *Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry* [online]. 2017 [viewed 18 March 2018]. Available from: `https://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml`.

[16] RESCHKE, J. *RFC 7617 – The 'Basic' HTTP Authentication Scheme* [online]. 2015 [viewed 1 April 2018]. Available from: `https://tools.ietf.org/html/rfc7617`.

[17] JONES, M. and HARDT, D. *RFC 6750 – The OAuth 2.0 Authorization Framework: Bearer Token Usage* [online]. 2012 [viewed 1 April 2018]. Available from: `https://tools.ietf.org/html/rfc6750`.

[18] SHEKH-YUSEF, R.; AHRENS, D. and BREMER, S. *RFC 7616 – HTTP Digest Access Authentication* [online]. 2015 [viewed 1 April 2018]. Available from: `https://tools.ietf.org/html/rfc7616`.

[19] FARRELL, S.; HOFFMAN, P. and THOMAS, M. *RFC 7486 – HTTP Origin-Bound Authentication (HOBA)* [online]. 2015 [viewed 5 April 2018]. Available from: `https://tools.ietf.org/html/rfc7486`.

[20] OIWA, Y. et al. *RFC 8120 – Mutual Authentication Protocol for HTTP* [online]. 2017 [viewed 6 April 2018]. Available from: `https://tools.ietf.org/html/rfc8120`.

[21] MELNIKOV, A. *RFC 7804 – Salted Challenge Response HTTP Authentication Mechanism* [online]. 2016 [viewed 9 April 2018]. Available from: `https://tools.ietf.org/html/rfc7804`.

[22] THOMSON, M. and BEVERLOO, P. *RFC 8292 – Voluntary Application Server Identification (VAPID) for Web Push* [online]. 2017 [viewed 9 April 2018]. Available from: `https://tools.ietf.org/html/rfc8292`.

[23] STUTTARD, D. and PINTO, M. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. 2nd ed. Indianapolis: Wiley, 2011. 912 p. ISBN 978-1-118-02647-2.

[24] HAMMER-LAHAV, E. *RFC 5849 – The OAuth 1.0 Protocol* [online]. 2010 [viewed 7 April 2018]. Available from: `https://tools.ietf.org/html/rfc5849`.

[25] HARDT, D. *RFC 6749 – The OAuth 2.0 Authorization Framework* [online]. 2012 [viewed 13 April 2018]. Available from: `https://tools.ietf.org/html/rfc6749`.

[26] PARECKI, A. *OAuth.com – OAuth 2.0 Servers* [online]. 2016 [viewed 13 April 2018]. Available from: `https://www.oauth.com/`.

[27] *Burp Suite Documentation* [online]. 2018 [viewed 15 April 2018]. Available from: `https://portswigger.net/burp/help/`.

[28] DRHOVA, K. *Authentication master – Burp Suite Extension* [online]. 2018 [viewed 15 April 2018]. Available from: `http://klara.drhova.cz/AuthenticationMaster/index.html`.

[29] *google-gson* [online]. 2018 [viewed 16 April 2018]. Available from: `https://github.com/google/gson`.

[30] *The DROWN Attack* [online]. 2016 [viewed 16 March 2018]. Available from: `https://drownattack.com/`.

[31] *Triple Handshakes Considered Harmful: Breaking and Fixing Authentication over TLS* [online]. 2014 [viewed 16 March 2018]. Available from: `https://www.mitls.org/pages/attacks/3SHAKE`.

Appendix A

Acronyms

3DES Triple DES, Triple Data Encryption Algorithm

3G Third Generation

AES Advanced Encryption Standard

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart

CBC Cipher Block Chaining

CORS Cross-origin Resource Sharing

CSP Content Security Policy

CSRF Cross-site Request Forgery

DNS Domain Name System

GB Gigabyte

GCM Galois/Counter Mode

HMAC Keyed-hash Message Authentication Code

HOBA HTTP Origin-Bound Authentication

HSTS HTTP Strict Transport Security

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IP	Internet Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
LAN	Local Area Network
LTE	Long-Term Evolution
MAC	Message Authentication Code
MIME	Multipurpose Internet Mail Extensions
NTLM	NT LAN Manager
OWASP	Open Web Application Security Project
RFC	Request for Comments
RSA	Rivest–Shamir–Adleman
SCRAM	Salted Challenge Response Authentication Mechanism
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VAPID	Voluntary Application Server Identification Authentication
XML	Extensible Markup Language
XSS	Cross-site Scripting

Appendix B

Attacks against SSL and TLS

There are many types of attacks against the SSL and TLS protocols, some of them having serious consequences. Certain versions of these protocols suffer from implementation and design flaws. Some attacks take advantage of configuration issues. A brief description of chosen attacks can be found below.

- *BEAST attack* (*Browser Exploit Against SSL/TLS attack*) – This attack was published in 2011 and affects SSL 3.0 and TLS 1.0. The attacker takes advantage of a vulnerable implementation of the CBC mode of operation in the vulnerable protocols. In this implementation, the initialization vector is predictable and the attacker is thus able to guess and decrypt parts of a packet. Especially, decrypting HTTP cookies can be a serious issue. This attack can be prevented by using TLS 1.1 or TLS 1.2.

- *BREACH attack* (*Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext*) – This compression attack is a variant of the *CRIME attack* described below. The inbuilt HTTP-level data compression is exploited in this case which is much more prevalent than the TLS-level compression used in the *CRIME attack*.

- *CRIME attack* (*Compression Ratio Info-leak Made Easy attack*) – This attack uses a vulnerability found in TLS compression and can be used to predict sensitive information (such as the HTTP cookies) if TLS-level compression is used. This is done by guessing the secret character by character and observing the compressed size of the request for these different input values (for example, by sniffing the network traffic). If the compressed size is smaller, then the inserted character is equal to the one contained in the secret. To conduct this attack, the attacker needs to have reasonable control over the victim's browser or to make the victim click on a link with some malicious code. This issue can be mitigated by turning off the TLS compression on the server or client side.

- *DROWN attack* (*Decrypting RSA with Obsolete and Weakened eNcryption*) [30] – This cross-protocol attack [31] exploits a vulnerability in the deprecated SSL 2.0 protocol together with the configuration of a server that shares the same RSA keys between the two protocols. The SSL 2.0 protocol supports vulnerable export cipher suites [32] and thus can be used to obtain a symmetric session key for a captured TLS connection. Also, some weaknesses in older OpenSSL implementations can be exploited to reduce the effort required to break the encryption. The attacker can gain unencrypted communication between clients and the server and can impersonate a website on the vulnerable server (change the content that the clients see). This issue can be mitigated by ensuring that the server's private keys are not used with any software that allows SSL 2.0 connections.

- *FREAK attack* (*Factoring RSA Export Keys*) – This attack is a *downgrade attack* against export cipher suits – especially, the RSA cryptosystem. An attacker can trick a vulnerable client and a vulnerable server (using the *man-in-the-middle attack*) to use a weak RSA export keys. If the export RSA modulus with 512 bits or less is used, then the attacker can factor it quite easily using the number field sieve algorithm and can impersonate the server and fake a web site [33]. This issue can be solved by disabling support for all export cipher suites on the server.

- *Heartbleed attack* – This is an implementation attack that exploits a coding mistake. The vulnerability was found in 2014 in the extension of the cryptography library OpenSSL which is widely used by TLS protocol. In "*heartbeat*" message, the client sends a payload that contains data and information about its size. The server should response with the same message (the same data and size of the data). Due to the coding error, the server did not check whether the size corresponded to

[31]Cross-protocol attacks exploit bugs in one protocol implementation (in this case SSL 2.0) to attack the security of connections made under a different protocol (in this case TLS).

[32]Export cipher suites were designed in 1990s to be sufficiently weak that they could be broken easily by the National Security Agency (NSA) and the U.S. government but not by other organizations with lesser computing resources. However, with unceasing increases in computing power, they can be broken today by anyone with access to cloud computing services. For example, RSA using modulus with 512 or less bits can be broken in several hours. The *FREAK attack* exploits export-grade RSA, the *Logjam attack* exploits export-grade Diffie-Hellman, and the *DROWN attack* exploits export-grade symmetric ciphers – all three kinds of deliberately weakened cryptographic primitives have been exploited.

[33]Generating RSA keys is computationally expensive, and many servers do not generate them for every single connection. In fact, some of them generate a single RSA key at the startup and use it until they are switched off.

the data received. If an attacker sent bigger size than the real data size was, the server responded with the data from the attacker's request and some random data that was placed after it in the server's memory to meet the specified length.

- *Logjam attack* – This attack is a *downgrade attack* against Diffie–Hellman key exchange. We can gain the secret key established using Diffie–Hellman key exchange if we are able to solve the discrete logarithm problem which can be solved using the index calculus algorithm. In this algorithm, an attacker is able to compute first three steps of the index calculus algorithm in advance. The last step can be computed in relatively short time (for 512-bit prime in order of minutes). In this attack, the attacker uses a *man-in-the-middle attack* to downgrade a TLS connection to use 512-bit export-grade Diffie–Hellman prime and then computes the secret key that is shared between the client and the server. Using this shared key, she is able to read the exchanged data, modify them, or inject data into the connection. This attack can be mitigated by allowing only Diffie–Hellman key exchange with 2048 or more bits or by switching to Elliptic-curve Diffie–Hellman key exchange (ECDH) that cannot be solved directly by using the index calculus algorithm.

- *Lucky Thirteen attack* – This attack is a novel variant of the *padding oracle attack*. It is a *timing side channel attack* [34] against TLS implementation of the CBC mode of operation that allows the attacker to decrypt arbitrary ciphertext. It is rather a theoretical attack, as the timing differential caused by invalid padding is very small and can be exploited only from a close distance (e.g., over a LAN) and with repeating the process many times to eliminate the noise (that could change in the future with faster networks, etc.). The *Lucky Thirteen attack* can be mitigated by using authenticated encryption, such as AES-GCM, or encrypt-then-MAC instead of the TLS default of MAC-then-encrypt.

- *POODLE attack* (*The Padding Oracle On Downgraded Legacy Encryption attack*) – This attack is a variant of the *padding oracle attack* on the CBC mode of operation and affects SSL 3.0 (there is also its newer variant against TLS). It takes advantage of the support of SSL 3.0 on many servers and clients (due to compatibility reasons) and a vulnerability in SSL 3.0 protocol which is related to block padding. The attacker forces the client to downgrade the connection to SSL 3.0 protocol using a *man-in-the-middle attack* and then deciphers the value of an encrypted block by modifying the padding bytes. The attacker can thus retrieve

[34] *Side channel attacks* are attacks that exploit information leaked into surrounding which is not part of the normal function. For example, power consumption, electromagnetic radiation, timing information, sound, or returned errors can be used to provide additional information about the system.

the plaintext character by character. The easiest mitigation technique is to completely disable SSL 3.0 on the server.

- *Renegotiation attack* – This vulnerability is a major attack on the TLS renegotiation mechanism and applies to all current versions of the protocol and also to SSL 3.0. It was discovered in 2009. This attack allows an attacker to inject plaintext into the victim's requests. For example, if the attacker is able to hijack an HTTPS connection, she is then able to insert her own requests to the conversation the client has with the web server. However, the attacker cannot decrypt the communication between the client and the server. Using the *renegotiation attack*, an attacker can also downgrade a HTTPS connection to a HTTP connection, inject custom responses, perform denial of service, etc. There is an extension that is a permanent fix for this vulnerability.

- *Sweet32 attack* – This attack breaks all 64-bit block ciphers (e.g., 3DES) used in CBC mode by exploiting a *birthday attack* and some way to capture enough traffic to launch the *birthday attack* (either a *man-in-the-middle attack* or injection of a malicious JavaScript into a web page). An attacker is able to recover a plaintext without knowing the encryption key. Firstly, she needs to collect a big amount of blocks enciphered with the same key; the amount depends on the length of the block and is equal to roughly 32 GB of ciphertext for 64-bit blocks to have a good chance of success. Secondly, she needs to find a collision of two blocks – if we assume that half of the plaintext blocks are known to the attacker, she would need to increase the amount of ciphertext to about 64 GB. Finally, she is able to find the plaintext block corresponding to the colliding ciphertext block using the known plaintext block and ciphertext blocks preceding to the colliding ones (they play the role of initialization vectors).

- *TIME attack* (*Timing Info-leak Made Easy attack*) – In this compression attack, the attacker needs to redirect a victim to a malicious website that will run some code to get the secret data. The attacker does not observe the compressed size but the time it takes to send these messages across the network. If the message is longer (less compressed) and overflows into an additional TCP packet, then the time necessary for the sending will be longer. In contrast with the *CRIME attack* and the *BREACH attack*, the attacker does not need to sniff the network. This attack can be mitigated by disabling TLS compression.

- *Triple Handshake attack* [31] – In this attack, the attacker is able to establish two connections which has the same encryption keys. If a client connects to a malicious server, the client presents user's credentials. The server can then misuse these credentials and impersonate the client at any other server that accepts the same credentials. The malicious server performs a kind of *man-in-the-middle attack*. This attack does not rely on implementation errors and can be prevented, for example, by using the *Content-Security-Policy* response header.

Appendix C

Contents of enclosed DVD

```
readme.txt ....................... the file with DVD content description
extension ........................... the directory with the extension
    AuthenticationMaster.jar ........... the extension in JAR format
    doc ............................... the directory of documentation
    src ................................ the directory of source codes
text ....................................... the thesis text directory
    DP_Drhova_Klara_2018.pdf ........... the thesis text in PDF format
    thesis ............. the directory of LaTeX source codes of the thesis
```