Security Assessment of RESTful APIs through Automated Penetration Testing

Bachelor Thesis

submitted by:
Christopher Wiedey

Student ID: 405979
Topic proposed by:
Dr. Ludger Becker

Thesis supervised by:
Dr. Tobias Brix

Münster, August 28, 2020

Contents

1. Introduction

Since the creation of the World Wide Web in the early 1990s, much of its development has been driven by inventive and entrepreneurial spirit rather than conceptional rigor. The growing complexity led to demands in a more standardized architecture of the Web. Roy Fielding's works on frameworks for hypermedia systems proved to be very influential for this. Particularly his concept of *Representational State Transfer* (REST) architecture for hypermedia systems has found widespread recognition [1, 5]. APIs utilizing Fielding's principles - often labeled RESTful APIs - have since become a popular tool for accomplishing functional dependence across distributed systems.
Today, privacy and security have grown to be high-profile issues for users and providers of Web Services alike. Ransoming, identity theft and industrial espionage, for instance, are now commonplace in public perception and have consequences for the personal well-being of many. We believe that the dissemination of knowledge about the undermining of system integrity is useful to create a greater demand for higher security standards and thus improve the level of application security in general.

In this thesis, we collate professional guidelines regarding the security of RESTful APIs and illustrate their raison d'être by presenting concrete scenarios crafted to bypass naive approaches to system integrity. We examine the concept of RESTful APIs - in theory as well as in practice - and related security mechanisms. Drawing from this, we try to identify common patterns that allow a generalization of security-relevant concepts. Furthermore, we enlarge upon the technique of penetration testing as a method of application security assessment and assurance. Relying on characteristics identified beforehand, we look into common vulnerabilities for RESTful APIs by theoretical explanation and practical demonstration. Lastly, we discuss the feasibility of an automation of security assurance and critically evaluate the limits of our previous considerations.

We do not assume that the reader has a deep knowledge of web technologies or security, as we try to explain important concepts in advance. However, a certain familiarity with hypermedia technologies such as HTML and an appreciation of the problem of system integrity is recommended.

2. REST and RESTful APIs

2.1. Basic Terminology

In this section, we will briefly explain our usage of terms derived from the REST architecture style described in 2.2. The terms *REST*, *RESTful* and *RESTful API* have since their conception become colloquial phrases. We nonetheless want to remain as consistent as possible in their usage:

- With the terms *REST architecture*, *REST style* or just simply *REST* we refer to the definitions originally made in [1, 5]. We will explain those in further detail in 2.2.

- Although Fielding makes extensive use of the the term *Application Programming Interface* (API), he does not offer a formal definition [1].
  We understand an API in an abstract sense as a set of system components intentionally exposed to the outside to provide a way of programmatic or manual interaction with the system. As we are only concerned with web technologies, the interface is assumed to be exposed to a network unless stated otherwise.

- We use the term *RESTful* as in [2, pp.36] to refer to an API or Web Service providing system interaction at least partially consistent with the REST style. The exact motivations and limits of this restriction will become clear in 2.2, where we'll examine common usage of this term..

- It is difficult to give a comprehensive definition of a *Web Service*. In order to keep consistency with our different sources, we will use this term in the spirit of the discussion in [2, pp.1] according to its vague colloquial use as "any kind of service delivered using Web technologies"[2, p.2]. Consequently, every (RESTful) API is a Web Service, but not vice versa.

2.2. An Overview of RESTful APIs

The REST architectural style was originally envisioned by Roy Fielding in his dissertation *"Architectural Styles and the Design of Network-based Software Architectures"* in the year 2000 [1]. It was conceived by observing the evolving World Wide Web with a motivation to specify an architecture complementing useful network-based properties such as *portability*, *scalability*, *simplicity*,

performance, *visibility* and *independent evolvability* [1, 2]. REST is defined by consecutively adding constraints to an initially unrestricted system [1, 5.1]. In the following sections, we will further examine these constraints.
While this theoretical groundwork has been fundamental in the more recent development of Web Services, RESTful Web Services today are very heterogeneous and often differ greatly from Fielding's specification. In order to attain a notion of what constitutes RESTful Web Services in practical use, we will explore additional sources.
Neumann et al. (2018) examined public APIs claiming to be RESTful and evaluated the services' adherence to Fielding's REST style [3]. In this chapter, we will use these findings to enrich our understanding of RESTful Web Services by supplementing Fielding's abstract principles with real-life practices. To gain insight into the views of industry professionals, we will also examine concrete guidelines from Microsoft regarding the design of RESTful APIs [4]. Although the guidelines are not a recognized standard beyond that company's sphere, we find it reasonable to consider such a view.

2.2.1. Client-Server

REST architecture imposes the *Client-Server architecture* as a constraint for communication between network components. In [1, 3.4.1], Fielding describes it as the *client* - a process initiating a request - working in conjunction with the *server* - a process reacting to that request.
This design is motivated by the so called *separation of concerns* [1, 5.1]. It permits a clear functional distinction between user interface and data storage. This in turn improves the capabilities of both the server and the client component. The client process only has to be specified in regards to the server interface, leading to greater flexibility and portability, while the server process is reduced in complexity, making it scale better with client demands. Moreover, Client-Server distinction permits independent development components by multiple actors.
As the Client-Server architecture is the dominant network architecture today, it comes to little surprise that each of the surveyed APIs in [3] complied with this architectural constraint.

2.2.2. Statelessness

A much characteristic feature of REST architecture is the constraint of *stateless interaction* between client and server as described in [1, 5.1.3]. This means that client requests must essentially be self-explanatory. The server must be able to interpret a request solely by the data contained in the request, prohibiting any session context for the server to rely on.
The design choice is meant to promote the properties of simplicity, scalability and reliability by relieving the server of the intricacies of session handling.

This, however, comes at the price of a possibly significant overhead in request data and the server's loss of control over the flow of application execution. In his survey, Neumann cannot find conclusive data concerning the overall adoption of this constraint [3, 5].
Microsoft does not explicitly mention statelessness[1]. It becomes clear, however, that use of self-explanatory requests is mandated through HTTP verbs, HTTP header usage and URL patterns [4, Section 7]. More on that in 2.2.4.

2.2.3. Caching

A constraint imposed on server responses is the support for *resource caching* [1, 5.1.4]. The client must be able to decide whether and how long the data contained in a response is safe to cache. It is up to the implementation how this will be realized in detail.
Given that data on the internet is often static, this constraint on resource semantics can potentially reduce traffic by a significant amount, thus further improving scalability and overall efficiency of the network communication.
Neumann finds that only 20.6% of the surveyed APIs support caching [3, 4.1; Fig. 15]. All these cases use the *Cache-Control* response header. Microsoft does not mandate the use a header field in the list of standard response headers [4, 7.6].

2.2.4. Identification, Modification and Representation of Resources

The constraint of an *uniform interface* demands a strict distinction of a component's specification from its actual implementation [1, 5.1.5]. Endpoints - so called *connectors* - facilitate indirect interaction with the component's functional parts while requiring no knowledge of their inner workings whatsoever. Much like the Client-Server constraint, this greatly encourages independent evolvability.
Fielding understands a *resource* as information that is target of a hypertext reference [1, 5.2.1.1]. Any such resource should be identifiable by a *resource identifier* - e.g. *Uniform Resource Identifier* (URI) - and thus be able to be referenced. No further constraints are imposed on the nature of the identifier. The respective naming authority is responsible for an identifier's validity.
Neumann finds that the adoption of resource-oriented URIs has been increasing over time, amounting to 87.8% of the alleged RESTful APIs in 2018 [3, 4.1]. From this we conclude, that the adoption of resource oriented API design itself is becoming similarly popular. Microsoft only requires human readable URLs as resource identifiers [4, 7.1].

[1] We note that Microsoft permits the use of cookies, which can be seen as a contradiction to statelessness. The issue of cookies has also been acknowledged by Fielding [4, 8.2] [1, 6.3.4.2]. See 2.3 for further insight into application state in context of access control.

Method	Description	Is Idempotent
GET	Return the current value of an object	True
PUT	Replace an object, or create a named object, when applicable	True
DELETE	Delete an object	True
POST	Create a new object based on the data provided, or submit a command	False
HEAD	Return metadata of an object for a GET response. Resources that support the GET method MAY support the HEAD method as well	True
PATCH	Apply a partial update to an object	False
OPTIONS	Get information about a request; [...]	True

Table 2.1.: Usage of HTTP verbs as specified by Microsoft. Table taken from Microsoft's API Guidelines [4, Table 1].

Addressability with changing location or name of the resource target is facilitated through canonical identifiers (see [4, 7.3]). Furthermore, explicit versioning is required as part of URL [4, Section 12]. This very common pattern is in conflict with REST style resource identification as noted by Neumann, because version information does not identify a resource and should thus not be part of a resource identifier [3, 4.1].
Resources are accessed through *representations* [1, 5.2.1.2]. A representation can be any sequence of bytes as long as it is attached to metadata in form of key-value pairs. The metadata contains *control data* - e.g. HTTP verbs - which specifies the nature of the resource access such as editing or retrieving data. The representation also contains a directive for its interpretation called *media type*. HTTP verbs are widely used as control data. Though Microsoft does not mandate that specific verbs are to be used, it imposes consistency when using them [4, 7.4]. The the verb definition is shown in Table 2.1.
Representations themselves should be hypermedia. Clients should be able to navigate through an application without knowing any resource identifier beforehand, further strengthening independent evolvability. This concept is named *Hypermedia as the Engine of Application State* (HATEOAS) [1, 5.3.3]. Neumann finds that merely 4.2% of surveyed services confirm to HATEOAS best practices [3, Fig. 17].

2.2.5. Layered System

REST style applications should package their components as hierarchical layers [1, 5.1.6]. Interaction between these components is facilitated through a uniform interface like in 2.2.4. Hierarchy-transparency can be achieved through the use of proxies.

Among benefits such as adding the potential for distributed caching and further decoupling of application components, this constraint leads to easier control over component functionalities through the specification of their interfaces.
Neumann found no conclusive data regarding the adoption of this constraint [3, 5].

2.2.6. Code-On-Demand

The last constraint is explicitly labeled as optional since it should only be included if the application architecture benefits from it [1, 5.1.7]. With *Code-On-Demand*, client functionality may be expanded through executable code sent by the server.
This can simplify client design and decouples server and client. However, it comes at the expense of visibility which in turn may be detrimental to maintainability and security of the service.
Though Neumann does not provide any data on the adoption of Code-On-Demand, modern web browsers have adopted to support four different coding languages: *Hypertext Markup Language* (HTML), *Javascript*, *Cascading Style Sheet* (CSS) and most recently *Web Assembly*[2].

2.3. Security Mechanisms for RESTful APIs

Although Fielding addresses the need for authentication mechanisms complementing REST style architectures, he does not go into detail regarding their nature [1, 4.1.4.1] [1, 6.3.4.2]. We understand security mechanisms for REST style architectures as mechanisms ensuring that only intended parties are able to access certain resources. Consequently, there need to be mechanisms in place that prevent access to these resources in some other way. In the following, we will look at both kinds of access control:
First, we will look at *authentication* and *authorization* mechanisms. These are designed to regulate resource access through the proper interfaces. Then, we will explore mechanisms designed to prevent more underhanded attempts to resource access.

2.3.1. Authentication and Authorization Mechanisms

Publicly accessible APIs often serve confidential resources and must therefore be able to check a client's permission for access to them. For this purpose, two distinct concepts of access control - that are often confused with one another - have been established: *Authentication* and *Authorization*.

The *Open Web Application Security Project* (OWASP) defines authentication

[2] `https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en`

as "*the process of verifying that an individual, entity or website is whom it claims to be*" [5]. Permission to access a certain resource is granted because of the client's supposed identity. According to Neumann, 86% of all surveyed APIs require some form of authentication for API calls [3, 4.4].
Authorization, on the other hand, is defined as "*the process where requests to access a particular resource should be granted or denied* " [6]. While a client authorized to access a particular resource will likely be authenticated (which must not always be the case. See 2.3.1.3), an authenticated client may not be allowed to access all resources served by the API.

We will now present different solutions commonly used to control access to RESTful API resources. These, among others, include some security mechanisms surveyed by Neumann [3, 4.4].

2.3.1.1. HTTP Basic

This is a native HTTP authentication method as described by the IETF [7]. The server can challenge the client to authenticate by responding to a request with a `401` (Unauthorized) status code along with the `WWW-Authenticate` header field such as:

```
WWW-Authenticate:  Basic realm="<realm>", charset="<charset>"
```

The client authenticates itself by responding with the `Authorization` header:

```
Authorization:  Basic <credentials>
```

This authentication method is very easy to deploy and can be used in accordance with the REST style *statelessness* principle by including that header in every client request.
The IETF notes several security issues with this kind of authentication: Firstly, the scheme itself does not allow any encryption, which means that secrets are transmitted in plain text. This mode of encryption is also susceptible to *spoofing*: If a malicious server masquerades itself as the actual authentication target, the client might expose the secret by sending it to that server. [7, 4].[3]
Furthermore, the protocol itself does allow for a fine-grained permission management like OAuth 2.3.1.3.

2.3.1.2. Client Certificate

Unlike HTTP Basic authentication, Client Certificate authentication is not facilitated on the application layer, but rather on the transport layer. Analogous to the server identity verification by the client (see 2.3.2), the TLS handshake allows for client authentication by the server [8, 4.4.2].
The great advantage in using client certificates is that secure communication

[3] See 2.3.2 for a solution that solves this problem.

through TLS should already be implemented by a Web Service[4]. Demanding client certificates therefore requires very little additional server-side configuration, as this feature is already fully integrated into the TLS handshake. The security assurance of the authentication process is also hugely simplified, as its correctness depends on the correctness and proper configuration of the TLS implementation. Since authentication takes place on the transport layer, the application itself is unburdened of the need to implement its own authentication procedure.
The improved security and reduction in complexity comes at the cost of increased client-side configuration and maintenance. Authentication and maintenance of username/password credentials can be considered more trivial than authentication through a certificate. When sufficient capabilities of the user-agent or the user cannot be assumed, one might choose to avoid using client certificates in favor of a less complex scheme.

2.3.1.3. OAuth

OAuth refers to two distinct protocols used for authorization: OAuth 1.0 and OAuth 2.0. Both are specified by the IETF and employ *delegated authorization* to control resource access [9][10]. Although Neumann finds that both protocols are actively used by RESTful APIs, IETF deems OAuth 1.0 obsolete [3, Fig. 11] [10]. Due to the the inherent complexity a complete implementation of the OAuth 2.0 flow would entail, we omitted this authorization scheme from our hands-on demonstrations in 4. We will nonetheless briefly explain the OAuth 2.0 protocol.

The IETF gives an overview into the motivations and basic workings of the protocol [10, 1]: OAuth 2.0 is designed to allow resource access for third-party applications without having to store authentication credentials. The resource owner can grant and revoke access to particular resources selectively. Instead of authenticating via username-password credentials, the third-party receives an *access token* through which it can prove authorization to access the resource. Authorization can be granted for a limited time.
If an access token expires or is otherwise deemed invalid, the client can issue a new access token using a *refresh token* [10, 6]. This is illustrated by [10, Figure 2].

2.3.2. HTTP over TLS

Transport Layer Security (TLS) is specified by IETF and designed to facilitate authenticated and encrypted communication on Transport-Layer protocols such as the *Transport Control Protocol* TCP. Although TLS is designed

[4]More on that in 2.3.2.

to supplement arbitrary Transport-Layer protocols, we will limit our explanations to *Transport Control Protocol* (TCP) connections. We will examine TLS 1.3, which is, as of now, the most recent version of the protocol.
The specification allows the protocol to be used in numerous different ways. Discussing all of these options would be too extensive for this thesis and many special cases are not relevant to the subject of conventional Web Services. We will thus only concern ourselves with connections requiring server authentication that are conducting a key exchange. We will also not touch upon the cryptographic details, as they are not relevant to the implementation of these security mechanisms and require knowledge we cannot assume here.
TLS consists primarily of two distinct protocols: The *handshake* protocol and the *record* protocol [8, 4;5]. The handshake is concerned with authentication of the server (and optionally the client) as well as negotiation of cryptographic parameters. During the handshake, client and server conduct some form of *Diffie-Hellman key exchange*, which results in a shared key with which subsequent communication will be encrypted symmetrically. [5]
The record protocol is concerned with protecting higher level traffic such as HTTP communication.
Encryption through TLS does not mask the *hostname* and the *port* from nodes on the routing path, as these are necessary to facilitate the TCP/IP connection. Aside from protecting the contents of the communication between client and server, TLS provides means for the client to ensure the authenticity of the server. According to IETF specification, during the handshake the server must usually[6] offer a certificate to the client whereby the authenticity of the server can be validated [8, 4.4].
The certificate includes a reference to a *Certificate Authority* (CA) and a signature encrypted with a private key only the real server should possess. The CA can verify the authenticity of the server by successfully decrypting that signature with the corresponding public key which the CA has knowledge of. Figure 2.2 illustrates this sequence, at the end of which the client has confidence in the identity of the server.
It should be noted that the authenticity of the CA must usually be verified as well. This leads to a so-called *certificate chain* in which each subsequent CA validates its predecessor up until a *root certificate* which the client's user agent should be aware of.

It is worth briefly discussing why the use of the same TLS context is not at odds with the *statelessness* principle of REST style applications mentioned

[5] See the original publication by Diffie and Hellman for further details on the key exchange [11].

[6] There are some exceptions, namely *Pre-Shared Keys* (PSK), Raw public keys without certificate context or public keys without certificate chain validation [8, 4.4.2; C.5]. In most cases this is not preferable, so we will omit these possibilities from our further explanations.

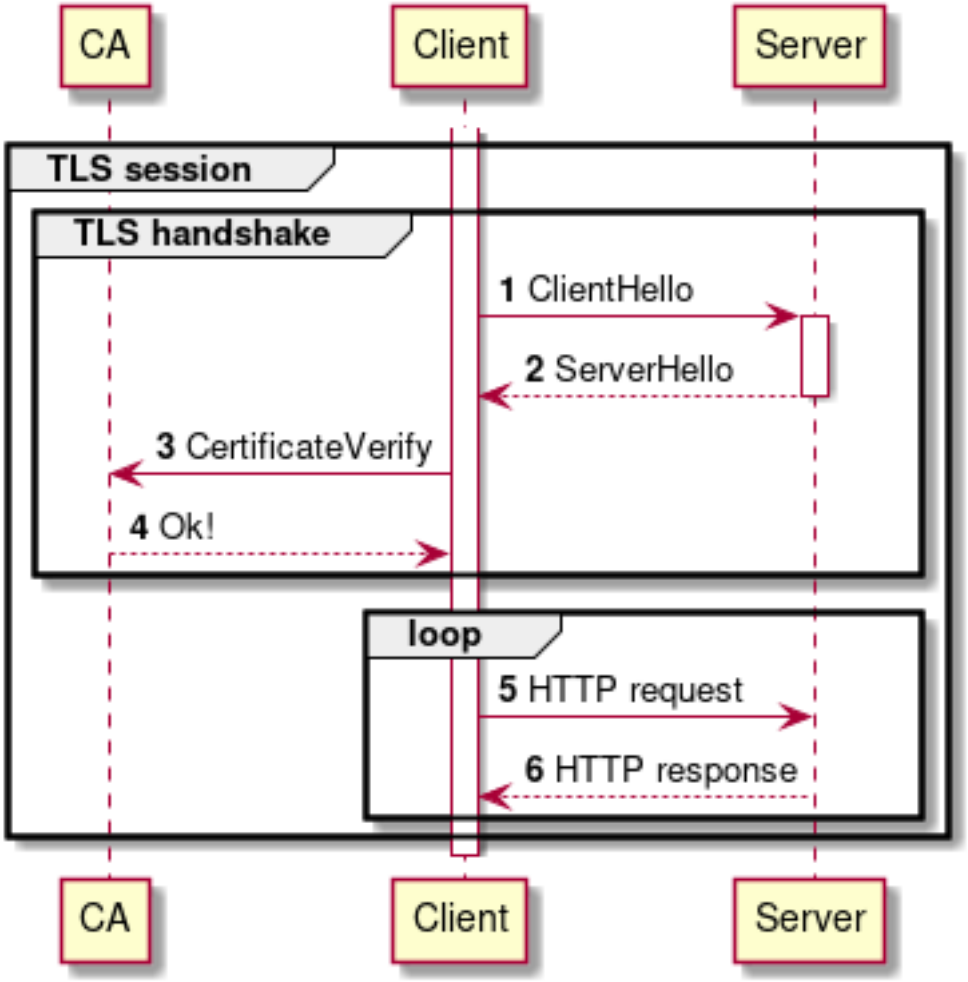

Figure 2.1.: Typical HTTP Server-Client-Communication over TLS.

in 2.2.2. Although both client and server negotiate communication parameters potentially for multiple request-response cycles, this happens on the transport layer. The process is transparent for the application data exchange.

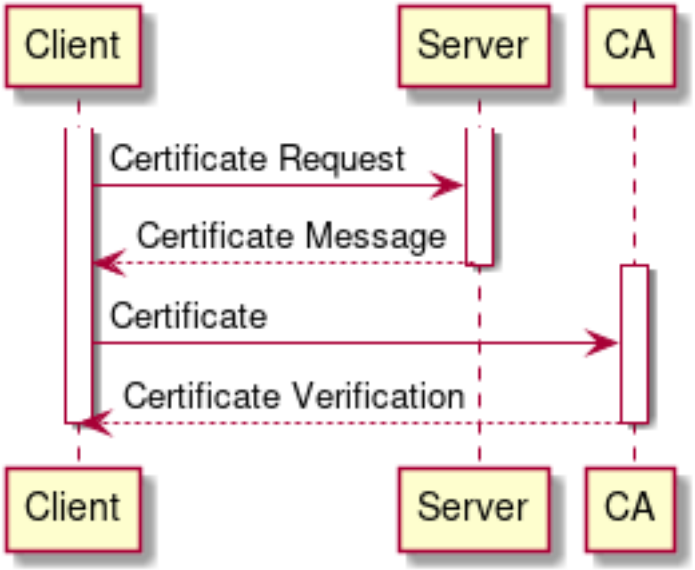

Figure 2.2.: Basic flow of a certificate validation during a TLS handshake.

3. Penetration Testing

There are significant security vulnerabilities to be considered when developing IT-Systems. Given the accessible nature of Web Services, the severity of potential exploitation of these vulnerabilities can not be overstated. Certain techniques have been developed to provide confidence in the security capability of such systems. We will now discuss in detail a technique called *Penetration Testing*.
Since it is difficult to find sources that describe this technique specifically for our object of study, we will examine penetration testing in a general context. This chapter's validity for RESTful APIs will be discussed in 5.2.
As a primary source for definitions and provision of information on this topic, we use a guideline by the German Federal Office for Information Security - *Bundesamt für Sicherheit in der Informationstechnik* (BSI) - aimed at informing enterprises and institutions planning to carry out such tests [12].

3.1. Definition

Penetration testing, or simply *pen testing*, is a technique used to analyze the security of an IT-System. The BSI as an institution is tasked with providing IT-Security advice to organizations and also offers such tests themselves to organizations and institutions[1]. It defines a penetration test as a process to identify the potential threat an adversary may pose to an IT-System and whether current security procedures can ensure a system's safety [12, 1.3].
The terms *Black-Box-Test* and *White-Box-Test* concern the level of knowledge about the system testers are granted beforehand:

- A **Black-Box-Test** is an attempt to simulate an actual attack on a system as it could be conducted by an external actor. Information given to the tester is minimal in order to construct a more realistic intrusion scenario.
- A **White-Box-Test** in turn is an attempt to simulate an attack by an actor that already has some information on the system.

The BSI strongly advises conducting White-Box-Test [12, 1.3]. This stems from a variety of reasons: Vulnerabilities may be overlooked, especially if they

[1] Retrieved 25th August, 2020: `https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/Dienstleistungen/ISPentest_ISWebcheck/ispentest_iswebcheck.html`

can only be found by an internal actor. Black-Box-Test are more prone to cause damage to the system, because the tester is unable to fully judge the consequences of her actions. Lastly, the BSI estimates the effort needed to conduct Black-Box-Tests to be substantially higher than for White-Box-Tests.

3.2. Requirements

The BSI mentions possible motivations for an organization to conduct a penetration test [12, 2.1.1]. These motivations all aim to prevent a successful attack on the organization's IT-Systems, which can have a variety of consequences: *data loss*, *damage to one's reputation* or *loss of corporate secrets*.
Although similar in approach, penetration testing should not be confused with *quality assurance*. The latter has the purpose of testing *functional* aspects of an application, including the implementation of security mechanisms. This quality assurance should be facilitated by repeatable tests and embedded in the life cycle of the system. [12, 2.1.1]. Penetration tests are also not meant to verify adherence to certain security standards. The BSI refers to particular audit procedures designed to achieve this [12, 2.1.1].
A penetration test should aim to find security flaws which are not covered by such security standards. Target components for the penetration test should be chosen based on the potential harm an attack might cause to them. As extensive pentesting is expensive, the expected increase in security should be weighed against the costs of such a test.
The BSI mandates that penetration tests always have to be conducted by *qualified personnel* that is *not involved with the tested system* and has *no dependence relationships* with those responsible for the system [12, 2.1.2]. This is to prevent conflicts of interest which might influence the test in any way. Given these requirements, the BSI strongly advises contracting *external* testers.
There are additional legal and organizational requirements we will not cover in much detail. Among them are agreements regarding *privacy*, *secrecy* and *liability* which should be negotiated before the test is started [12, 2.3].

3.3. Test Parameters

The target system of the test should be chosen carefully. This could be a responsible component if the pentest is a response to a previous security incident, or, in the case of a preventive test, a component that is generally considered critical [12, 3.1.1]. After having chosen the target, the following four aspects must be further specified as noted by the BSI [12, 3.1.2]:

1. **Test-Depth:** Three levels of invasiveness are specified. A *technical security-audit* examines versions and configurations of relevant applications. During a *non-invasive vulnerability scan*, the components are

passively scanned with available tools. In an *invasive vulnerability scan*, attempts are made to actively exploit identified vulnerabilities. This last options should only be utilized when necessary and only using well-known exploits, as it might damage the system in an unforeseen way.

2. **Test-Location:** It must be determined how and where the testers are going to access the test target. Testing in the respective organization is generally advised, as valuable insights might be collected through conversation with the staff. Testing may also take place in a data center to gain insight into the conditions and handling of technical equipment. Pragmatic considerations might also encourage remote testing.

3. **Test-Conditions:** Test equipment and working environments must be considered. It should be evaluated whether the test should be executed in the production environment or in a simulated test environment. Testers should be granted access to relevant systems.

4. **Test-Duration:** It should be specified beforehand how long the pentest should last. That duration should allow for proper familiarization with the respective components.

3.4. Test Procedure

The testers should begin by familiarizing themselves with the system [12, 4.1.1]. For this purpose, a thorough documentation should be provided by the organization, as specified by the BSI [12, 3.1.3]. The BSI structures further progression of the test into four distinct phases [12, 4.1.2]. As most of them are concerned with organizational aspects, we will only examine the third phase, which describes the technical aspects of the test.
This practical phase is again sectioned into distinct modules. Their order is not fixed, some modules might not even be used in certain test settings. We will now briefly summarize these modules as specified by the BSI [12, 4.1.2]:

1. **Conceptional Weaknesses:** Potential shortcomings in the integration or the design of system components may be discovered from examining their documentation. Proof of these weaknesses can be provided through practical tests or further inquiry.

2. **Safety Measures:** The implementation of appropriate safeguards is validated in this module. The objects of investigation are versatile and may include *open ports*, *software versioning* or *access control*.

3. **Identification of Vulnerabilities:** Known vulnerabilities for relevant system components are researched. These can be identified either through

flaws in the safety measures or programmatically using a so-called vulnerability scanner (see 4.4). There are publicly accessible databases providing details on numerous known application vulnerabilities such as the *Open Source Vulnerability Database*[2]

4. **Vulnerability Exploitation:** To prove whether an identified vulnerability is in potentially harmful, it must be exploited. There exist public databases containing scripts that automatically exploit known vulnerabilities, for example Offensive Security's *Exploit Database*[3].
   Special care must be taken when using any kind of exploit. The use of applications in a manner not originally intended can lead to damage to system components, such as data loss or data leakage. Exploits must be thoroughly analyzed before they are used.

The BSI makes the point that tools can be used to support the testing process [12, 2.2.4]. It is noted, that such tools need to be evaluated carefully beforehand, as improper handling might damage system components. Only experts familiar with these specific tools should use them during penetration testing.

3.5. Final Analysis

The test should be concluded with a meeting in which the responsible personnel is informed about the most relevant test results [12, 4.1.2]. The testers might offer their advice if immediate intervention is necessary. As a thorough evaluation might take some time, additional findings are analyzed later and compiled into a final report [12, 4.1.3].

[2] `https://www.whitesourcesoftware.com/vulnerability-database`
[3] `https://www.exploit-db.com/`

4. Security Considerations for RESTful APIs

In this chapter, we raise important points to consider when assessing the security of RESTful APIs. These points relate to the security mechanism presented in 2.3 and components identified to be commonly used in RESTful APIs. Based on our findings in 2.2, we chose *HTTP*, *resource persistence* and *resource representation* as characteristics of conventional RESTful APIs[1].
Just as it is hardly possible to provide perfect security for any Web Service, we cannot put together an exhaustive list of all conceivable risks. We will, however, compile information deemed important by security professionals. For this, we use the security guidelines made specifically for RESTful Web Services in the guidelines maintained by the *Open Web Application Security Project*[2] (OWASP) - a community-driven foundation aiming to improve web security. An evaluation and justification of this source can be found in 5.

In order to illustrate how these vulnerabilities might occur and how they can be identified and solved, we built a simple RESTful API which we will subject to various vulnerabilities explained in this chapter. The API - which we named *Message Service* - serves bindings for a basic text messenger: An authenticated user can send messages to other users and view all messages sent between them. The documentation found in A.1 provides a more in-depth explanation. An accessible interface and a setting in which we present security flaws related to resource representation is provided by the *frontend* to our API. Its documentation is given in A.2.

4.1. Insecurity of HTTP Connections

In the setting of Client-Server communication facilitated partly over a public network, both client and server have little control over the routing of their packages. Figure 4.1 exemplifies this by showing stations a request passes on its way to a destination server. Any of these stations receives packets of the request and is in principle able to modify them and to read their contents.
A situation where an attacker is deliberately relaying the traffic between client

[1] See 5.1 for a thorough discussion on the characteristics of RESTful APIs.
[2] `https://owasp.org/`

```
traceroute to apache.org
1 gw-v3901.uni-muenster.de (10.68.0.1)
2 tu30u30ma2-fwo2.uni-muenster.de (172.27.2.138)
3 tu30u30ma1-fwo2.uni-muenster.de (172.27.2.137)
4 ti23u30gr1.uni-muenster.de (172.27.142.121)
5 cr-han2-be10-3001.x-win.dfn.de (188.1.234.221)
6 cr-fra2-be12.x-win.dfn.de (188.1.144.133)
7 ffm-b5-link.telia.net (213.248.97.40)
8 ffm-bb1-link.telia.net (62.115.114.88)
9 s-bb3-link.telia.net (62.115.138.236)
10 hls-b1-link.telia.net (62.115.123.31)
11 hetzner-svc067711-ic351605.c.telia.net (62.115.183.185)
12 core31.hel1.hetzner.com (213.239.224.38)
13 ex9k1.dc2.hel1.hetzner.com (213.239.224.138)
14 tlp-he-fi.apache.org (95.216.24.32)
```

Figure 4.1.: Different routing points a request from inside the network of the WWU Münster passes on its way to `apache.org`.

and server in order to access or manipulate it is called *Man-in-the-middle attack*[3].

4.1.1. Mallory's Middleman Machine

The network layout for this scenario is shown in Figure 4.2 and the sequence of the event is depicted in Figure 4.3. All communication is facilitated trough HTTP requests without TLS. It is possible to carry out this scenario in a local setting by following the explanations in A.3.
To give insight into the issue, we present a realistic scenario that touches upon essential aspects of Man-In-The-Middle attacks:

Alice wants to send a message to Bob through the *Message Service* API which is served by the API endpoint on the domain `api-endpoint`. Alice enters the message text and her credentials into a User-Agent on her PC, which in turn sends an HTTP Request to the API endpoint. The API validates the authentication for Alice's user account, ensures that the account is authorized for this action and persists the message.
Some time later, Bob wants to check for new messages from Alice. He instructs his User-Agent to prompt the API endpoint for all messages sent by Alice. Just like Alice, he passes his credentials to the User-Agent which embeds them into the HTTP-header of the request. Unbeknownst to him, his requests to the API at `api-endpoint` are routed through Mallory's Man-in-the-middle machine, the middleman server. There are many reasons something like this can happen:

[3] `https://owasp.org/www-community/attacks/Man-in-the-middle_attack`

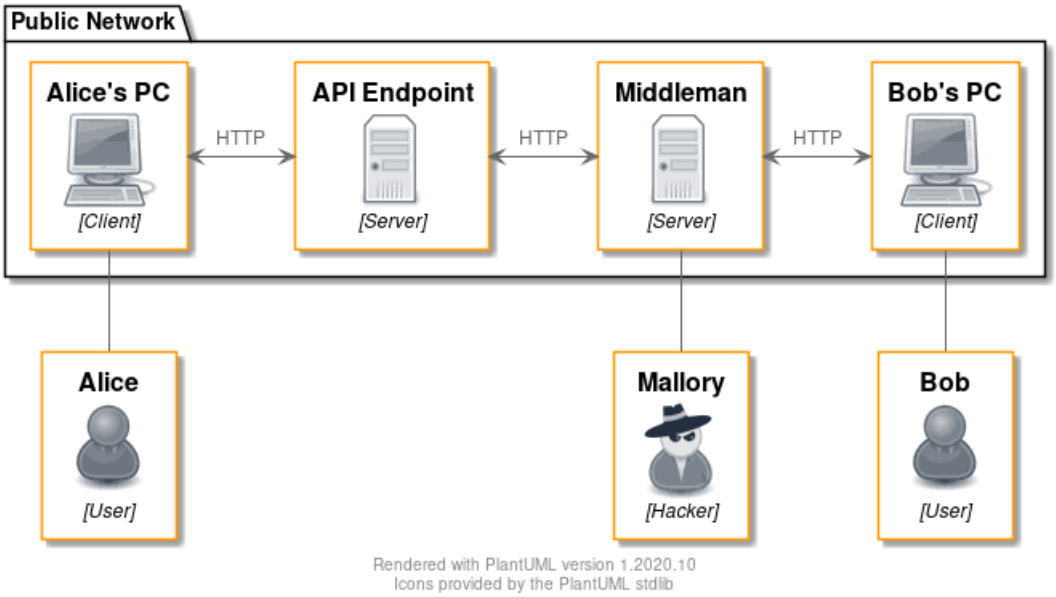

Figure 4.2.: Network layout for the *Man-in-the-middle* scenario in 4.1.1 and A.3.

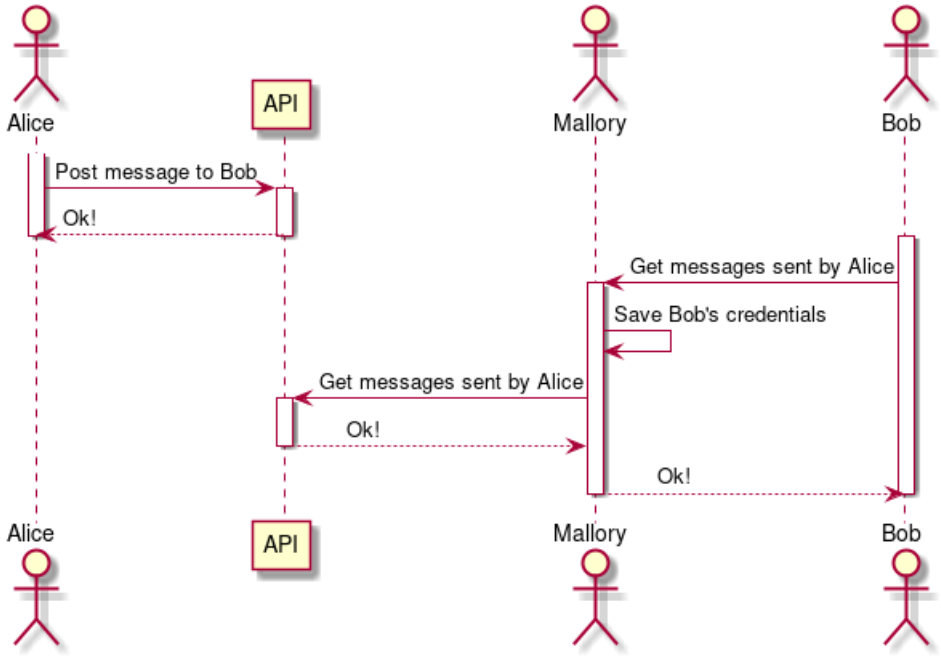

Figure 4.3.: Sequence of the Man-in-the-middle scenario described in 4.1.1 and A.3.

- The middleman might be a gateway Bob's network packages necessarily have to pass through, like a *router* on his local network, a *VPN-Server* or an *Internet Exchange Point* (IXP).
- Bob's Domain Name System (DNS) resolution might resolve the API's domain name to the middleman's IP-Address instead that of the API's IP-Address. Upon receiving Bob's request, the middleman can relay the request to his actual destination i.e the API.
- Much like the incorrect DNS resolution, Bob might directly connect to the middleman server by providing his User-Agent with the wrong address for the API. This can easily happen through subtle errors such as a differing *port*, *subdomain*, *Top-level domain* or simply wrong spelling.
- The security of Bob's User-Agent, machine or network might be compromised in other ways to allow for such an attack.

Bob's request reaches the middleman server. There, instead of being directly relayed to the API endpoint, it is thoroughly inspected. Mallory is now able to obtain Bob's credentials by reading the HTTP header.[4].
By now, Mallory herself can request the API for Bob's resources. She copies Bob's original request for his messages, sends it to the API endpoint herself and receives a response from the API containing the message Alice previously sent to Bob.
In order not to draw Bob's suspicion, Mallory must respond to the request she originally received from Bob. She can, however, choose to alter the contents of Alice's message or completely omit it from the response at all. When Bob eventually receives the response for his initial request, he doesn't know that it was actually sent by Mallory. From his point of view, he has received the requested resources.
Mallory, on the other hand, now has control over Bob's user account through his credentials. She can query for other resources and also post new messages. As long as Bob's traffic is routed through her middleman server, she can also continue to inspect any of Bob's subsequent requests to the API and modify the responses.

4.1.2. Identifying Man-in-the-middle Vulnerabilities

The vulnerability - as discussed here - primarily results from using unencrypted HTTP instead of HTTP over TLS. The problem can be identified on the Transport Layer. In most cases, HTTP endpoints are served through the port `80`, HTTPS through the port `443`. Thus, these endpoints can be monitored and policed through firewalls and the configuration of the HTTP server.
Improper use of HTTPS might be harder to identify. Serious shortcomings

[4]see 2.3.1.1 for an explanation how exactly the credentials are handled.

like unsuccessful certificate validation are often identified by user-agents. More subtle issues can be discovered automatically by analysis tools such as *HTTP Observatory* by Mozilla[5].

4.1.3. Preventing Man-in-the-middle Attacks with HTTPS

As mentioned in 2.3.2, HTTPS is specifically designed to prevent a scenario as shown in 4.1.1. OWASP lists the following benefits of using TLS [13]: *Confidentiality*, *Integrity*, *Replay prevention* and *Authentication*.

We will now revisit our middleman example from 4.1.1 to showcase how it and similar scenarios can be prevented with the use of HTTPS. In order not to unnecessarily widen the scope of these examples, we assume that negotiation of communication parameters between client and server is minimal[6]:

Message Confidentiality Protection of HTTP application data is facilitated through encryption. We apply TLS encryption to our example from 4.1.1 to see how it mitigates the Man-in-the-middle vulnerability. See the IETF's specification for a more technical explanation [8, 5.2]:

Once again, Bob wants to read the messages Alice sent to him via the *Message Service API*. Only now, his User-Agent tries to establish a HTTPS connection to the API endpoint. This means that a TLS handshake is conducted before any application data is sent. The User-Agent generates a private-key-public-key pair and sends the *ClientHello* containing his public key towards the API endpoint. Again, the traffic is intercepted by Mallory's Man-in-the-middle server.
Mallory is able to read the public key contained in the message and keeps track of it. She relays the traffic to the API endpoint which responds with its certificate containing the API's public key. Mallory is now in possession of both public keys and forwards the API's response back to Bob's machine.
With their own private key and the other party's public key, both Bob and the API can calculate an identical key - the master-secret - used to encrypt and decrypt further communication e.g. the HTTP request containing Bob's credentials. Even with having knowledge of both public keys, Mallory is not able to calculate the master-secret and is thus unable to decrypt the communication.

Server Authentication Trusted certificates allow the authentication of the server. We will now apply this scheme to the scenario from 4.1.1. See the the IETF's specification for a more technical explanation [8, 4.4]:

[5] `https://github.com/mozilla/http-observatory`
[6] TLS permits negotiation of cryptographic parameters [8, 4.1.1]

As passively eavesdropping on the communication proves to be unsuccessful, Mallory now tries to impersonate the API endpoint. When she intercepts Bob's public key, she responds with her own certificate, waiting for Bob to make a new request containing his credentials.
Before continuing the conversation with the fake API endpoint, Bob's User-Agent checks whether the certificate is valid. He sends it to the CA that originally issued the certificate to verify if it actually belongs to `api-endpoint`. The CA tells Bob that Mallory's certificate does not match the one of the API endpoint.
Now the User-Agent knows it was not talking to the actual API endpoint and aborts the connection to Mallory before having sent any application data.

Message Integrity Preventing the modification of messages by a third party is primarily ensured by encryption. However, even the encrypted message could still be altered, an issue that is solved using a so-called *Per-Record Nonce* [8, 5.3]. Again, we show how such an attack is prevented by using our example from 4.1.1:

In this scenario, Bob has successfully established a TLS session with the real API endpoint. The traffic is still routed through Mallory's middleman machine and she decides to alter the response Bob receives from the API.
Anticipating that Bob attempts to retrieve all messages by Alice, Mallory modifies the API's response, deleting all application data[7].
Mallory is unable to add the expected Per-Record Nonce to the new application data. Even if she can guess what the correct value is, she is unable to encrypt it with the proper master-secret. When Bob's User-Agent receives the modified response, it notices the missing nonce. Bob is alerted that the response been modified.

Replay Prevention The last aspect concerns *session replay*, a technique with which requests of a TLS session are saved and sent again. This is prevented by different means: By using the Per-Record-Nonce from the previous scenario and also by using different random values to create the shared master-secret during the TLS handshake [8, 5.3] [8, 4.1.3]. We once again explain how such an attack is prevented by using our scenario from 4.1.1:

We assume Bob has successfully received all his messages from the API. Mallory intercepted this response and knows - through unrelated means - that it contains the response to Bobs message query.
During the same TLS session, Bob - looking for new messages from Alice -

[7] This would, of course, also void all structural HTTP data, an issue Bob would hardly accept as a valid response. However, Bob deeming the response as unacceptable is not an acceptable metric for verifying message integrity.

issues the same request again. Mallory intercepts his request and correctly guesses what Bob is requesting from the API. Not wanting him to get any new messages, she sends him the same response Bob has already received from the API.
Bob's User-Agent receives the fake response, decrypts the application data and finds that the *Per-Record Nonce* is derived from an earlier request in the TLS session. Just like before, Mallory is unable to modify the encrypted nonce. Bob therefore knows that the response is not legit.
Note that the Per-Record-Nonce offers the same protection for traffic received by the *server*. Both communication endpoints keep tack of the amount of request / responses they sent so far and derive the nonce from this value.

In a slightly different scenario, Mallory recorded Bob's communication from an entire TLS session. After some time, she wants to find out if Bob got any new messages. To that end, Mallory sends the recorded requests - including the TLS handshake - to the API once more. Although she cannot decrypt the traffic, she wants to compare the new responses to the ones she recorded earlier to find out whether the requested resources have changed.This fails, however, as the API endpoint responds with a different *ServerHello* when starting the TLS-Handshake. The ServerHello contains a new random value used to calculate the master-secret. Because the master-secret has changed for the new session, Mallory is unable to use the old session data.

4.2. SQL-Injection

According to OWASP's chart of *Top 10 Web Application Security Risks*[8], vulnerabilities to *injection* are the most common security flaws in web services. The term refers to the insertion of code into user input which - through inadequate design of the application - is executed on the server. Depending on the component executing this code, this flaw can lead to *exposure* and *modification* of arbitrary data or even direct access to the server's OS interfaces.
In the following, we will focus our explanations on injections targeting SQL-Servers - so-called *SQL-Injections*. SQL databases are widely popular resource persistence solutions for web services. Most of the points raised here can similarly be applied to other persistence solutions. We mainly draw information on this topic from OWASP's SQL Injection Prevention Cheat Sheet, which offers comprehensive explanations on how to identify and mitigate such injection vulnerabilities [14].

SQL-Injection attacks require dynamically built SQL-Statements containing untrusted user input. The dangers of such implementations may not always be obvious to a developer. We now provide a realistic scenario on how such a

[8] `https://owasp.org/www-project-top-ten/`

query might occur in an application. The layout of this scenario is shown in Figure 4.4. A guide on how to reenact this example is included in A.4.

As part of the *Message Service* RESTful API (see A.1), the `user` resource is exposed in a RESTful manner through the following URI:

`message-service/users/<email>`

Upon receiving such a request, the API itself queries its SQL database for the `user_id` corresponding to the given email.

For example: As a result of the GET-request for

`message-service/users_injectable/alice@wonder.land`,

the API sends the following Query to its SQL-Database:

```
SELECT id, name, email
FROM users
WHERE email = 'alice@wonder.land';
```

One might falsely assume that RESTful APIs which provide access to resources in such a way are immune to injection given the more limited set of possible URL characters. This is wrong, as arbitrary ASCII characters can be encoded in an URL as we will show now. The API passes the `<email>` part of the URI to the SQL-Statement. Consequently, a GET-request for

`message-service/users_injectable/%27%20OR%201=1--`

results in the following statement being sent to the database:

```
SELECT id, name, email
FROM users
WHERE email = '' OR 1=1;
```

OWASP differentiates between three kinds of data retrieval in injection attacks [14]:

1. **Inband:** The most basic form of SQL-Injection. The response of the modified SQL-Statement already contains the data the attacker intends to acquire. The injections given above are a an example of this. For our scenarios in A.4, we concern ourselves solely with these kind of data retrieval due to the sophistication required by the following types.

2. **Out-of-band:** The attacker gains access to the data through interfaces different from the one the injection took place through. For example: The `user_id` of the message recipient are replaced with a `user_id` for a user whose messages the attacker can read. That way, the attacker could simply access the messages through the application's HTML interface.

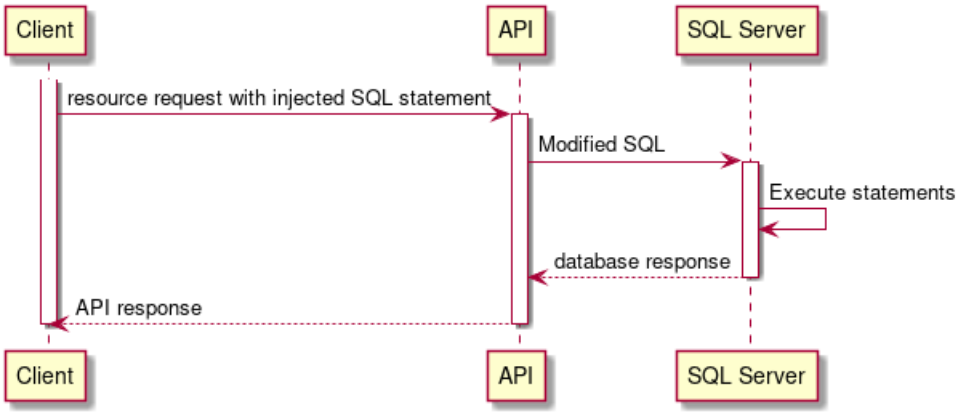

Figure 4.4.: Request processing in the SQL-Injection scenario from 4.2.

3. **Inferential or Blind:** Similar to *Out-of-band*, the attacker does not acquire the data directly. Different behavior to injected queries is measured, such as response time and error messages. The data is then retrieved by cleverly inferring it from a multitude of injection statements. One could, for example, inject the following statement valid for MySQL databases.[9] The statement is taken from OWASP's article on Blind SQL Injection[10] and has been slightly modified for clarity.

```
1 UNION
SELECT
 IF (SUBSTRING(user_password,1,1) = CHAR(50),
      -- then
       BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),
         -- else
          null)
FROM users
WHERE user_id = 1;
```

With this SQL-Statement, an attacker aims to find out whether the password of the user with `id` 1 begins with the character "2"[11]. If that is the case, the database will execute the `ENCODE` function multiple times before completing the query. Because the query is artificially delayed only when the if-condition is true - i.e the character has been correctly guessed - the attacker now knows whether the password begins with the a "2" and can continue to find out the password by brute force.

[9] See the Readme in A.4 for an explanation to how `UNION` statements can be injected.
[10] `https://owasp.org/www-community/attacks/Blind_SQL_Injection`
[11] "2" has an ASCII value of "50", hence the `CHAR(50)`.

4.2.1. Preventing SQL-Injection Attacks

Given the potentially disastrous consequences of an SQL-Injection vulnerability, software components handling user-input should tackle the problem from various angles and implement multiple safety mechanisms. OWASP provides the following solutions for such an *defense-in-depth approach* to security assurance [15, 7]. We built exemplary SQL-Injection scenarios into the `message-service` API in section A.4 which the following approaches.

1. **Sanitizing user input** through HTML-Entity encoding. HTML-Entities are representations of a certain set of reserved characters such as quote chars or brackets. Translating these characters to their HTML-Entity encoded representation can prevent an SQL-Parser from misinterpreting such user- input as code. Many libraries offer functionality for this encoding, such as PHP's `htmlentities` or Spring's `HtmlUtils`. An implementation with python's `html` module is available through the `users_sanitized` route in A.4.

2. **Static Code Analysis** during the implementation of the software. Tools are able discover flawed implementations in the source code itself. As the various flavors of this approach are intricate and manifold, we omit further reaching explanations and usage scenarios in favor of brevity and refer to OWASP's introduction into this topic [12].

3. **Parametrization of SQL-Statements** through the database engine. Such constructs aim to transparently separate code (the SQL-Statement) from user-input (the parameter). The engine constructs a statement with specific placeholders for the parameters and parses it. The parameters are then bound to the parsed statement. As the statement logic has already been parsed and the parameters arrive separate from the statement, the parser will not confuse the code with the parameters.
   Common web frameworks provide constructs for prepared statements, which act as a wrapper around the respective database engine's API. *PHP Data Object* [13] (PDO) and *Java Database Connectivity* (JDBC) are examples of such frameworks.
   Through the `users_prepared` route in A.4, we initiate a prepared statement before querying it supplemented with the (unsanitized) user-input.

4. Using **Stored Procedures**. This technique is similar to prepared statements in that the actual statement is parameterized and parsed before

[12] An overview of techniques, a discussion on uses and limits of Static Code Analysis as well as a listing of concrete tools is given here: `https://owasp.org/www-community/controls/Static_Code_Analysis`

[13] Note that PDO will emulate a prepared statement if the associated database driver does not support actual prepared statements. Those are just usual SQL-Statements with escaped parameters. See `https://www.php.net/manual/en/pdo.prepare.php`.

the user-input is inserted. Stored procedures are defined on the database level by a database user and provide an additional layer of transparency, as applications accessing the database do not have to use any SQL-Statements aside from the procedure call.
It is important to note that stored procedure calls must also be properly handled when containing user-input. User-input must at least be sanitized or a Prepared Statement should be used.
The `users_procedure` route in A.4 implements an SQL call to the predefined procedure `get_user_by_email` with unsanitized user-input.

5. Provide developer training to raise awareness and thus decrease the likelihood of injection vulnerabilities. This is hugely important as injection-related security flaws can easily be prevented and arise as a result of flawed implementations.

The route `users_multiple` in A.4 combines the methods 1,3 and 4, providing the most robust approach.

4.3. Cross-Site-Scripting

A potential security fault regarding *resource representation* resulting from improper handling of user-input is *Cross-Site-Scripting* (XSS). Being included in OWASP's Top 10 security risks in Web Applications[14], XSS vulnerabilities should definitely be considered when assessing the security of RESTful APIs.

Similar to attacks based on SQL-Injection as discussed in 4.2, XSS attacks inject specific instructions into user supplied-input. In contrast to SQL-Injections, where injected SQL-Statements are executed through the database engine, code injected through Cross-Site-Scripting is executed on the client's User-Agent. Vulnerabilities to XSS thus concern the *Code-On-Demand* aspect of REST style APIs as described in 2.2.6.
The possible effects of XSS attacks greatly depend on the particular environment the injected code is executed in. Code-On-Demand solutions are manifold and the *Client-Server architecture* (see 2.2.1) permits the use of arbitrary User-Agents. The degree of access an attacker might obtain to the client's machine is therefore hardly estimable. Even in the context of Internet Browsers and manipulation of the *Document Object Model* (DOM) through Javascript, possible consequences of XSS are manifold and conditional on a multitude of factors. In order not to digress from this thesis' topic, we will refrain from delving into that complex topic.

[14] `https://owasp.org/www-project-top-ten/`

4.3.1. Types of XSS

OWASP categorizes different kinds of XSS which depend on the origin of the injected data [16]: *Server XSS* and *Client XSS*. A further distinction, *Stored XSS* and *Reflected XSS*, depends on whether the injected data is persisted.
We will now explain these different types of XSS in detail. To demonstrate their causes and effects, we provide a hands-on example in A.5 which can be tried out in one's own Internet Browser. See also Figure 4.5 for a visual explanation of the attack sequences.

Stored Server XSS In this case, the client receives the malicious code from the server itself. The attacker must have somehow succeeded in persisting the code as part of a resource in the server's database. When the client requests and processes that resource, with its User-Agent, the malicious code is executed.
This form of XSS is especially severe because it can affect any client accessing the resource in question over an arbitrary amount of time. In our hands-on scenario (see A.5), the malicious code is part of a text message sent between two users. Both parties are potentially affected by the attack.

Stored Client XSS The attacker manages to somehow store the malicious code with the client. When using an Internet Browser, this might be facilitated through the *Browser Cache* or a *Cookie* which was used in our hand-on demonstration in A.5.
When the content of the stored element is processed as intended by the application, the malicious code is executed by the User-Agent. Just like *Stored Server XSS*, this type of XSS is rather severe because the attack can be carried for as long as the code is stored by the client. A single stored script, however, can only affect one individual client.

Reflected Server XSS This type of XSS relies on the server *reflecting* the malicious script. After somehow making the client send the payload script to the server, the client receives a response with the malicious script from the server. This type of XSS takes advantage of the client presuming the server's responses to be trustworthy. This kind of attack does not use persistence mechanisms to facilitate the attack. An attack is therefore only conducted once per sent payload script.
In our example in A.5, the server includes parts of user supplied login credentials into an error message. Appended HTML/Javascript code is then automatically embedded into the Webpage.

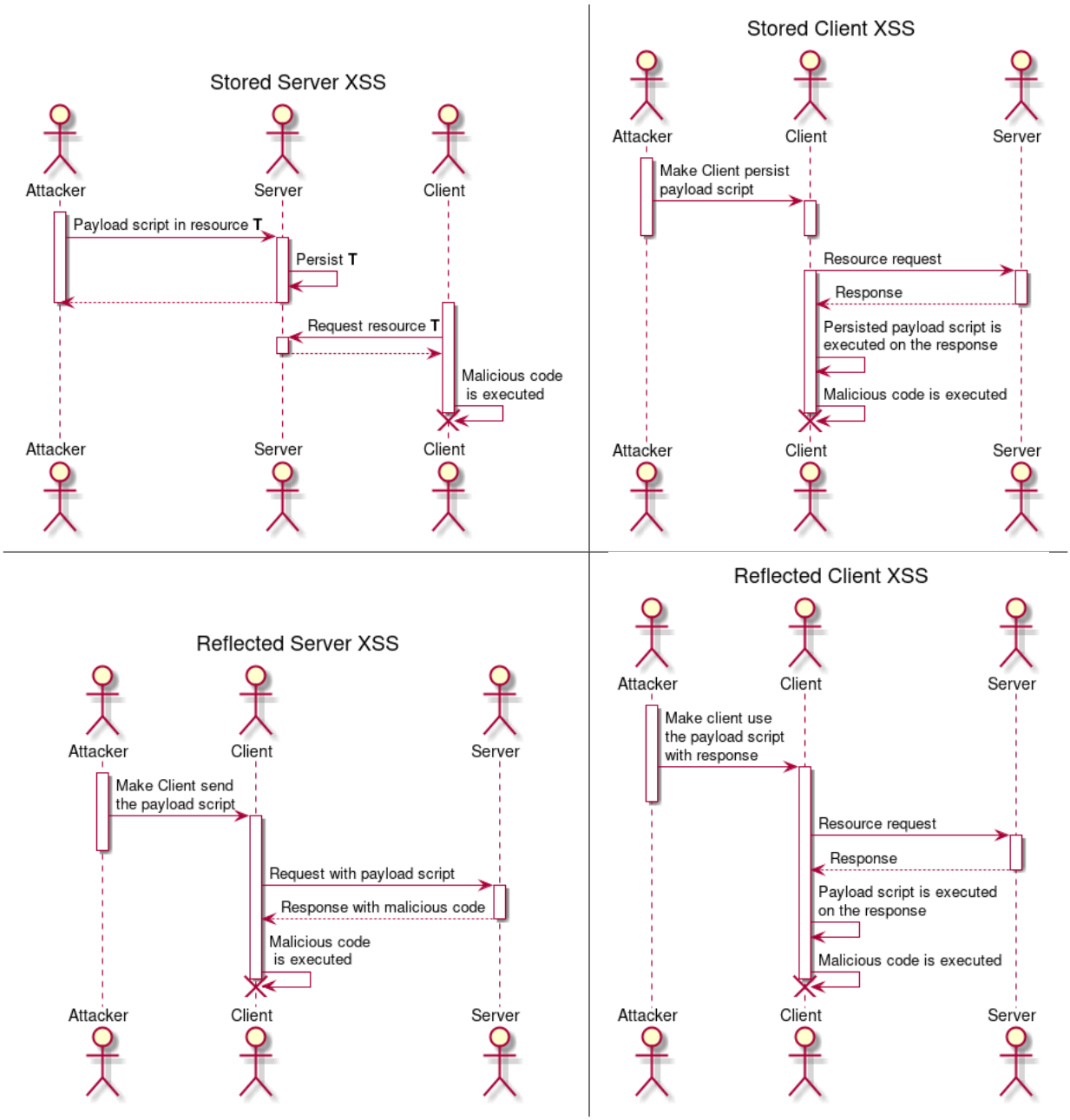

Figure 4.5.: Different types of Cross-Site-Scripting attacks.

Reflected Client XSS Just like *Reflected Server XSS*, this type of XSS does require any persistence mechanisms. The malicious script is injected using client-side data. That data can originate from any volatile source. In our exemplary scenario in A.5, we used GET-Parameters supplied in the URL. Setups like these allow an attack to take place just by following a link.

4.3.2. XSS Prevention

OWASP gives advice on how to prevent XSS vulnerabilities in a web application as part of their *Cheat Sheet Series* [17][18]. We will examine points relevant to our scenarios in A.5 and implement them in order to mitigate XSS vulnerabilities present.
The article encourages whitelisting sections of a HTML page which are allowed to contain untrusted input. Code execution is prohibited for these sections. It is vital that untrusted input is solely allowed in these specific sections and that this data cannot leak into sections that allow code execution. The prevention of XSS therefore narrows down to ensuring the correct implementation of this whitelisting. This includes preventing new whitelisted sections from being created through untrusted user input and ensuring the integrity of existing sections.
Rules pertaining to specific classes of sections for untrusted input are provided by OWASP, which give a lucid overview on different types of whitelisted section and how to properly implement them [17]. We will now present the rules which are fundamental for any HTML/Javascript context. These rules are expanded upon through additional ones for web pages utilizing DOM-Manipulation through Javascript like the frontend of our running example A.2 [18]. We will mention the rules for this more special context if relevant to the example.
The referenced encoding techniques are integral for the application of these application. OWASP advises against implementing these parsers by oneself, as subtle errors can easily lead to critical vulnerabilities [17].

As a 0th rule, OWASP excludes certain HTML environments from acting as whitelist sections as a matter of principle [17]: `<script>` and `<style>` entities, HTML comments, tag names and attribute names. Directly running Javascript code from untrusted sources is likewise strongly discouraged.

1. Untrusted input inserted into HTML content - meaning between the opening and closing tags - must always be HTML-Entity encoded. This prevents structural changes to the web page, including the insertion of `<script>` areas and the modification of sections whitelisted for user input.
   In the case of DOM modification through Javascript, the HTML subcontext - that means the HTML content dynamically inserted into the DOM - must similarly be HTML-Entity encoded. Additionally, the input

must be Javascript-Encoded afterwards to prevent the untrusted input interacting with the insertion function itself.

2. Before inserting untrusted input into HTML attributes, it must be escaped in a way to prevent breaking out of the attribute context. For this, OWASP advises encoding all non-alphanumeric characters whose ASCII value is below 256 into the `&#xHH;`[15] format.
   Attribute values should generally be quoted, although the substitution above accounts even for unquoted values. One should avoid inserting untrusted input into more complex attributes such as `href`, `src`, `script`, `style` and event handlers. If this has to be done, follow the rules for the attribute's respective environment.

3. When inserting untrusted input into javascript functions parameters or assignment operands, encode all non-alphanumeric characters whose ASCII value is below 256 into the `\xHH` format. It should be noted that some Javascript functions are always vulnerable to XSS when untrusted data is inserted as a parameter, regardless of how the data is escaped. The function `exec`, which executes a given string as Javascript code, is a notable example.
   When loading JSON-Strings directly into the DOM, be sure ot flag the response's content-type as `application/json` instead of `text/html`.The latter leads to the response being interpreted as HTML, executing code from `<script>` entities contained in the JSON-String. For additional guidance, we refer to the source's section on JSON values in HTML context [17, Rule #3.1].

4. The use of untrusted input in CSS likewise leads to XSS vulnerabilities. Arbitrary Javascript can be executed using the `evaluate` function or the `url` property. It is therefore vital to only include untrusted input into property values and never for properties that are complex, such as `url`. In order not to break out of a property's key-value context, untrusted input should always be encoded. All non-alphanumeric character with ASCII value below 256 should be encoded in the `\xHH` format.

5. When inserting untrusted input into GET-Parameters of an URL which is part of an HTML attribute value, encode all non-alphanumeric characters with ASCII value below 256 into the `%HH` URL encoding scheme. Moreover, attribute values should always be quoted.
   Data inserted into URL paths should not be handled that way, but according to the context they appear in. In the case of URLs as an HTML attribute value, the URL paths should be attribute encoded.
   The protocol part of an URL should never be built using untrusted input, as Javascript could potentially be loaded and executed or an insecure

[15] The letters "`HH`" stand for a 2-digit hexadecimal number.

connection vulnerable to Man-In-The-Middle attacks (see 4.1) could be initiated.

6. When expecting untrusted data containing HTML markup, encoding is not trivial, because the content cannot simply be HTML-Entity encoded, as this would destroy the markup. OWASP advises using external libraries that filter HTML components deemed unsafe. The *OWASP Java HTML Sanitizer*[16] is an example of this.

As shown in the hands-on scenario A.5, these rules can be applied to the API's frontend where required. This prevents all XSS attacks demonstrated in that scenario.

4.4. Test Automation

Given the rather procedural definition of a penetration test in 3, one might be tempted to try to automatize parts of it. We will briefly explore aspects of this idea now and further discuss it in 5.5. As mentioned in 3.4, penetration testers may rely on tools designed to aid the testing provided they are familiar with it.
We will now examine one such tool, the *OWASP Zed Attack Proxy* (ZAP Proxy), which claims to be "the world's most widely used web app scanner"[17] and is licensed under the free Apache License, Version 2.0. It can be run locally and accessed either directly through a REST API, a Web UI or SDKs for several programming languages. An extensive collection of add-ons is available through its marketplace[18].
The tool scans an application through a so-called *spider*. Spiders aim to explore an application through its hypermedia references. During the scan, the spider acts as a proxy between target server and User-Agent. Several User-Agents such as *Google Chrome* and *Mozilla Firefox* can be chosen to send the requests. The proxy then analyzes the traffic sent between User-Agent and target server and tries to find vulnerabilities .

We scanned the frontend (A.2) and the backend (A.1) of our example applications. The frontend had XSS vulnerabilities from A.5, the backend had SQL-Injection vulnerabilities from A.4. We used the *Ajax Spider* through a *Headless Chrome* user-agent. We deliberately didn't configure to the ZAP Proxy beyond providing user credentials and the server address. This was done to figure out how much information the proxy would provide to a layperson that lacks specialist knowledge to configure the proxy thoroughly.

[16] `https://github.com/OWASP/java-html-sanitizer`
[17] Statement from its homepage, retrieved 27th of August 2020: `https://www.zaproxy.org/`
[18] `https://www.zaproxy.org/addons/`

These are the results of both scans:

- Scanning the frontend yielded several alerts and notices, most notably the capture of authentication credentials by the proxy. This is a result of Man-In-The-Middle vulnerabilities as explained in 2.3.2 and 4.1. Apart from issues regarding the usage of plain HTTP, the ZAP Proxy found several HTTP header misconfigurations. The *X-Frame-Options* and *CSP* Headers were not found to be set. Their absence can lead to and aggravate Cross-Site-Scripting vulnerabilities.
  Each security issue discovered was supplemented with further information on the issue and possible solutions. However, the scan could not find any of the four Cross-Site-Scripting vulnerabilities from 4.3.

- Scanning the backend didn't produce any results. ZAP was unable to authenticate with the HTTP-Basic scheme, although credentials were provided. Because authentication is necessary for all API routes, no further results can be presented here.

While our test settings and results might beg many questions, we leave further examination to our final discussion of pentest automation in 5.5.

5. Discussion

To begin with, we want to address our choice and handling of sources regarding security advice. We chose OWASP over other authorities for the following reasons:
OWASP gathers and publishes information transparently through its community-driven processes, while also being endorsed and supported by well-known corporations such as *Oracle* or *Symantec*[1]. This makes it particularly well-suited as an unbiased but also competent authority on Web Security. Furthermore, OWASP offers freely accessible security guidelines as part of their *Cheat-Sheet-Series*. These guidelines are comprehensible to layman. We find this approach sensible when aiming to impart these practices to developers and engineers not particularly skilled in this area.
We understand that our approach to Web Security as done in this thesis has no aspirations to generality or professionalism. While we have confidence that our solutions to security vulnerabilities have been correctly collated from the aforementioned sources, we want to emphasize that our security advice in 4 should be taken with a grain of salt. We lack the specialist knowledge to evaluate the accuracy of our sources and therefore trust them on the grounds of their authority. When requiring professional security guidance, you should consult genuine experts in that field.

With that in mind and drawing from the examinations of preceding chapters, we want to discuss several open questions yet unanswered.

5.1. Characteristics of Conventional RESTful APIs

Before beginning to discuss this topic, we must clarify what we mean by *conventional* RESTful Web Service. In 2.2, we presented formal and informal characteristics of *Representational State Transfer* and examined adherence to them in public RESTful APIs as well as enterprise guidelines. We studied Roy Fielding's abstract definition of the elements constituting what he originally envisioned as REST [1, 5]. Drawing from a survey by Neumann et al., we found that apart from *Client-Server-Communication*, none of these characteristics is consistently used among a representative share of the top 4000 public

[1] See `https://owasp.org/supporters/` for a list of other supporters.

RESTful APIs [3, Fig. 15]. Moreover, we found that competent enterprises can not be expected to follow Fielding's definitions rigorously, which is exemplified by *Microsoft's REST API Guidelines* [4]. We conclude that the original cornerstones of Fielding's REST style are, apart from the *Client-Server-Model*, not appropriate for establishing common features of RESTful APIs.
One can justifiably challenge the ability of our sources to permit general statements about RESTful APIs. Neumann only examined services contained in an index of popular APIs, while Microsoft's guidelines can only be assumed to be applied for that company's respective services. We acknowledge that characteristics identified through these means have no claim to universal validity and we encourage more rigorous attempts in finding common trends and characteristics of existing RESTful APIs. However, our sources give evidence of the heterogeneity of RESTful APIs in general by providing counterexamples to supposed patterns.
Although not formally part of the REST style, we identified *the usage of access control* to be a widely used component of RESTful APIs. While there exists no common standard of authentication or authorization mechanisms, Neumann's data shows that apart from proprietary schemes, the mechanisms *OAuth 1.0/2.0* and *HTTP Basic* are prevalent [3, Fig. 11]. We thus count *the employment of access control*, and especially those variants, as characteristic features of conventional RESTful APIs.
Taking into account the prevalent, but also inconsistently implemented *resource-oriented design* of RESTful APIs, we include this design philosophy as a common element. We conclude *resource persistence* and *resource representation* as additional characteristics, as they are a direct consequence of embracing that design philosophy.
It should come to little surprise that it is usually implied for RESTful APIs to be served via *HTTP*, with or without TLS (compare [2, pp.1], [4]). We identify this as a common characteristic of Web Services and especially RESTful APIs.

To summarize: One can generally say that the notion of a RESTful API is in practice not well-defined. Apart from the aforementioned commonalities, we could not find notable structural or functional components consistently implemented in public APIs. However, the characteristic components we did find, fit existing RESTful APIs rather well.
During our research, we came across the vendor-neutral, thorough specification of RESTful APIs called *OpenAPI*[2] by the *OpenAPI Initiative*. We consider an approach similar to this thesis, but limited to OpenAPI compliant services, to be more fruitful.

[2] `https://www.openapis.org`

5.2. Classification of Penetration Testing

In 3, we examined penetration testing as a method for security assurance of Web Services and technical systems in general. The specification of this process by the BSI clarified this notion, which we find to be frequently subject to misconceptions [12]. Penetration testing is a process that demands extensive preparation and the thorough evaluation of intermediate and final results. Skipping or neglecting these phases can have undesirable, in some instances severe consequences for the organization and system being tested:
Thorough test preparation, as outlined in 3.2, should not be taking lightly. A system should implement and verify appropriate IT-Security-Guidelines such as the BSI's *IT-Grundschutz*[3] rather than blindly implementing arbitrary security practices. Strictly speaking, a pentest does not aim to verify an application's functionality, even if they concern security mechanisms. A penetration test is also not concerned with checking the implementation of IT-Security-Guidelines. Rather, it aims to find blindspots in and beyond existing security policies.
Failing to adequately outline proper test objectives can lead to inconclusive or misguiding assessment of security, resulting in unnecessary personnel and financial expenses. Commencing without properly evaluating legal consequences could prove disastrous for an organization.
While conducting the pentest, analysis of discovered vulnerabilities might demand an adjustment of test parameters. The extend of system intrusion and the exploitation of vulnerabilities must be carefully reconciled with the constraints specified during the preparation. Hacking the system irresponsibly might entail damage to vital system components or disruption of the productive system. Possible consequences need to be evaluated, a step which requires expert knowledge. Lastly, the eventual test interpretation and comprehensive advice concerning the improvement of security as described in 3.5, is likewise task for specialists.
This is all to emphasize that penetration testing is not equivalent to *vulnerability discovery* and *exploitation*. While these techniques can be a part of a penetration test, they do not make up the whole test. Careful planning and consideration of possible test settings must be carried out beforehand. Extensive analysis and presentation of the results is necessary afterwards.

Although we examined penetration testing in a general context so far, our explanations also apply to RESTful APIs as a special case of IT-Systems. The required IT-Security-Guideline should be specifically tailored to these kinds of applications. We unfortunately could not find guidelines similarly formal as the BSI's IT-Grundschutz. This is likely due to the previously discussed fact,

[3] `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2020.pdf?__blob=publicationFile&v=6`

that RESTful APIs is not a well-defined classification for Web Services. Being aware of the various components of one's RESTful API, one might choose appropriate documents from OWASP's Cheat Sheet Series[4] to act as a security guideline. As far as we looked into them, we found these documents well-written and, as demonstrated in 4, they give viable solutions to security issues.

5.3. Security Considerations for RESTful APIs

In chapter 4, we examined security vulnerabilities relevant for RESTful APIs. We presented a scenario for a Man-In-The-Middle-Attack on our example application in 4.1.1 and demonstrated, how trivial it can be for an attacker to sniff traffic content such as user credentials.
We perceive Man-In-The-Middle vulnerabilities to be extremely severe when it comes to APIs employing access restrictions for its resources, as exploitation of these vulnerabilities render all forms of access control pointless. Policing traffic routes is hardly an option for preventing this, as such a degree of control over routing points is both impractical and unrealistic. Encrypting HTTP traffic with TLS as explained in 2.3.2 and 4.1.3 is the better solution to prevent Man-In-The-Middle-Attacks, because HTTP data is secured regardless of how much trust is placed into in-between actors. TLS also offers authentication via certificates, which prevents impersonation of a Web Service by malicious actors independent from DNS services. Employing TLS as mentioned before, entails proper configuration of the HTTP-Server, an aspect we did not touch upon thoroughly. We want to refer to *badssl*[5] by the *The Chromium Project* for a demonstration on flawed TLS usage.

We identified injection attacks as a vulnerability for resource persistence in RESTful APIs as explained in 4.2. Injection vulnerabilities are a result of flawed handling of user input. Their exploitation by an attacker can lead to arbitrary access to a database.
Using SQL-Databases as an example persistence solution, we examined concrete attack scenarios and demonstrated through our example application how these vulnerabilities come to be (see A.4). We presented different solutions to these vulnerabilities in 4.2.1, namely *Stored Procedures*, *Prepared Statements* and *String Sanitizing*. Each of these solutions is able to prevent SQL-Injection on its own, though using more than one is recommended. Availability of these solutions is also a factor to consider. While String Sanitization should be available out of the box in most circumstances, the other two solutions are not supported by all database engines. The initial creation of Stored Procedures also requires privileged access to the database.

[4] `https://cheatsheetseries.owasp.org/`
[5] `https://github.com/chromium/badssl.com`

As mentioned above, injection vulnerabilities stem from mistakes in the implementation of the database communication. It is desirable and in our view also feasible to identify such vulnerabilities prior to deployment of the application through *Static Code Analysis*. We did not look into this subject in this thesis, but deem further research into that topic valuable, especially given its relation to the automation of security assessment.
While our examinations concerned communication with SQL-Databases, we suspect that similar injection vulnerabilities and solutions to them also exist for NoSQL-Databases. Future research should provide more insight into that topic.

In 4.3, we examined Cross-Site-Scripting vulnerabilities by examining their occurrence in the use of Client-Side web technologies such as HTML, Javascript and CSS. The idea that these technologies are characteristic of RESTful APIs is contestable. However, they facilitate resource representation which is, as argued in 5.1, a characteristic of RESTful APIs.
We found that XSS vulnerabilities occur Client-Side through injection of code into the user interface. Attacks can be classified as described in 4.3.1, *Stored Server XSS* is arguably the most severe type, as it affects multiple users for an arbitrary amount of time. XSS on an Internet Browser allows external Javascript to be executed, potentially altering the DOM, Cookies or the browser storage. As demonstrated in our hands-on example of different XSS types in A.5, *Social Engineering* can be facilitated by manipulating the website in a way that seems somewhat genuine or convincing. We omitted more subtle and realistic exploits of XSS vulnerabilities, because they require deeper examination of more advanced topics.
We found improper handling of untrusted user input to be the general source of XSS vulnerabilities. Correctly escaping these inputs depends on the context the data is inserted into. OWASP provides a set of rules ensuring proper handling of such inputs [17][18]. They proved helpful in eliminating the XSS vulnerabilities in our example application. We would like to remark, that we found the presentation of these rules somewhat convoluted. This is rather an effect of the intricacies of code nesting in HTML/Javascript environments than a shortcoming on OWASP's part. We see potential for a more lucid exemplification of XSS prevention, possibly in the form of a *decision map*.

We acknowledge that many important aspects of REST Security were not touched upon by us. Topics like *Update Policy*, *Firewalls* and *Server Configuration* are relevant aspects of System Administration warranting a separate examination.
Cross-Origin Resource Sharing (CORS) and *Cross-Site-Request-Forgery* (CSRF) are techniques hugely relevant to the security of Web Services and related to XSS. However, meaningful examination of these topics demands background knowledge partly provided by our chapter on XSS (see 4.3). Subsequent work

may use that chapter as groundwork for these more advanced topics.
We don't want to conceal the fact that we did not examine notable access control mechanisms more in-depth, particularly *OAuth 1.0* and *OAuth 2.0* (see 2.3.1.3). The reason for this is twofold: Underlying principles for these mechanisms are quite sophisticated, thus requiring lengthy clarifications and examples unsuited for our approach of broad-based coverage. Furthermore, implementing these mechanisms into our example application in a way that allows the demonstration of relevant vulnerabilities is beyond its design constraints and the capabilities of the developer.

5.4. Purpose of the Example Application

We decided to develop the applications in A.1 and A.2 to accentuate the theoretical considerations of our sources and our examinations in 2.2 with a practical implementation. This has already been proven to be fruitful, as it allowed us to better internalize and present this theoretical work.
The application is meant to be a *Deliberately Insecure Web Application*, a term used by OWASP for a collection of such applications[6]. Our application does not correctly correspond to Fielding's REST style, nor is it consistently secured. It is bespoke for the vulnerabilities examined in 4.
We planned on showing that the vulnerabilities from 4.1.1, 4.2 and 4.3 are exploitable and could be fixed with the given advice. This succeeded, as vulnerability exploitation could be successfully simulated *before* including appropriate fixes, but *not afterwards*.

5.5. Automation of Penetration Testing

We conclude this thesis by discussing whether automated penetration testing is a possibility for evaluating the security of RESTful APIs. We firmly answer this question with 'No'.

As argued in 5.2, penetration testing entails steps aside from scanning for vulnerabilities. Preparation and follow-up are vital parts of the process. Interviews with the staff are to be expected throughout the test execution. Even the actual examination of a system doesn't generally follow a checklist or security guideline, but depends on the testers' individual abilities. These requirements, we estimate, are impossible to automate for the foreseeable future.

Even if we restrict ourselves to the automation of vulnerability scanning, we face some problems. In 4.4, we attempted to scan our example application

[6] `https://owasp.org/www-project-web-security-testing-guide/latest/6-Appendix/B-Suggested_Reading#deliberately-insecure-web-applications`

using the *ZAP Proxy* vulnerability scanner with minimal configuration. We found that while the tool found some vulnerabilities related to TLS and HTTP-Header configuration in application's frontend, existing XSS vulnerabilities were not discovered. Scanning the backend failed completely. While we concede that this is most likely due to insufficient configuration, it exemplifies the blatant fact that reasonable operation of such a tool requires both knowledge of surveyed system as well as security technology in general. We are confident that this observation can be extended to comparable security scanners.
Vulnerability scanners can absolutely be used to support other security assessment procedures, provided they are correctly handled and understood (see last paragraph of 3.4). However, actual security assurance - with or without tools - requires knowledge of appropriate security procedures as well as insight into the composition of the system. As RESTful APIs withstand formalization of their structure, this process is further complicated.

Having said this, assurance of the correct implementation of an application's functional specification, inter alia it's security mechanisms, can be gained through extensive *unit testing*. To achieve automation, this process can be integrated into build and deployment pipelines. A quality assurance like this is also recommended by the BSI to validate a system's functional aspects [12, 2.1.2].

6. Conclusion

In this thesis, we aimed to research RESTful APIs and whether their security can be ensured through automated penetration testing.
Comparing aspects of those APIs both in theory and practice, we found that apart from the usage of HTTP and a rough notion of *resource orientation*, general similarities in these Web Services can hardly be identified. We provided background-knowledge on security mechanisms for Web Services. We examined a definition of *penetration testing* with the question of potential automation in mind. Covering common elements of RESTful APIs established beforehand, we constructed an example application which we used to demonstrate relevant security vulnerabilities and their mitigation through hands-on scenarios.

We critically discussed our methodology and came to the conclusion, that the idea of automated penetration testing is a futile endeavor, regardless of the type of application. While there are tools for automated vulnerability scanning, we pointed out that their appropriate use requires expert knowledge. In the case of RESTful APIs, special knowledge of the application's structure is also required, as these services can hardly be generalized.

In summary, the notion of a RESTful API is not very helpful to classify Web Services for security purposes. The client-specific, personal support of a penetration test cannot be replaced by automatic mechanisms. Automation can support a penetration test, but that still requires security experts. Rather than penetration testing, we deem automated *quality assurance* to be a more realistic way to improve the security of a Web Service.

A. Appendix

Various applications and scenarios used in the thesis are referenced here. Please read the readme file in the root directory of the appendix first. A digital copy of this thesis is located at `appendix/thesis.pdf`.
There is also an online version located at `https://zivgitlab.uni-muenster.de/c_wied05/bachelorarbeit-beispielprogramme` that is identical to the offline version.

Before using any of these applications or scenarios, please look at the readme file at the respective location.

A.1. Backend for the Example Application

`./bachelor-applications-backend`

A.2. Frontend for the Example Application

`./bachelor-applications-frontend`

A.3. Man-In-The-Middle Scenario

`./tls_playground`

A.4. SQL-Injection Scenario

`./injection_pharmacy`

A.5. XSS Scenario

`./xss_intersection`

Bibliography

[1] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[2] Erik Wilde Cesare Pautasso. *REST: From Research to Practice*. 08 2011.

[3] A. Neumann, N. Laranjeiro, and J. Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 2018.

[4] Microsoft. *Microsoft REST API Guidelines*. `https://github.com/microsoft/api-guidelines/tree/7e63c970b5763f8ffdf4ac276f598ee4d13a667d`.

[5] OWASP. *Authentication Cheat Sheet*. `https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html`.

[6] OWASP. *Access Control Cheat Sheet*. `https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html`.

[7] IETF. *The 'Basic' HTTP Authentication Scheme*. `https://tools.ietf.org/pdf/rfc7617.pdf`, 2015-09 edition.

[8] IETF. *The Transport Layer Security (TLS) Protocol Version 1.3*. `https://tools.ietf.org/html/rfc8446`.

[9] IETF. *The OAuth 1.0 Protocol*. `https://tools.ietf.org/html/rfc5849`.

[10] IETF. *The OAuth 2.0 Authorization Framework*. `https://tools.ietf.org/html/rfc6749`.

[11] Whitfield Diffie and Martin E. Hellman. New directions in cryptography, 1976.

[12] Bundesamt für Sicherheit in der Informationstechnik (BSI). Durchführungskonzept für penetrationstests. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Sicherheitsberatung/Pentest_Webcheck/Leitfaden_Penetrationstest.pdf?__blob=publicationFile&v=10`, 2016.

[13] OWASP. *Transport Layer Protection - OWASP Cheat Sheet Series*. https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html.

[14] OWASP. *Injection Prevention - OWASP Cheat Sheet Series*. https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html.

[15] OWASP, https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf. *Code Review Guide*, 2017.

[16] OWASP. *Types of XSS — OWASP*. https://owasp.org/www-community/Types_of_Cross-Site_Scripting.

[17] OWASP. *Cross Site Scripting Prevention - OWASP Cheat Sheet Series*. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[18] OWASP. *DOM based XSS Prevention - OWASP Cheat Sheet Series*. https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html.

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit über *„Security Assessment of RESTful APIs through Automated Penetration Testing“* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Vorname Nachname, Münster, August 28, 2020

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Vorname Nachname, Münster, August 28, 2020