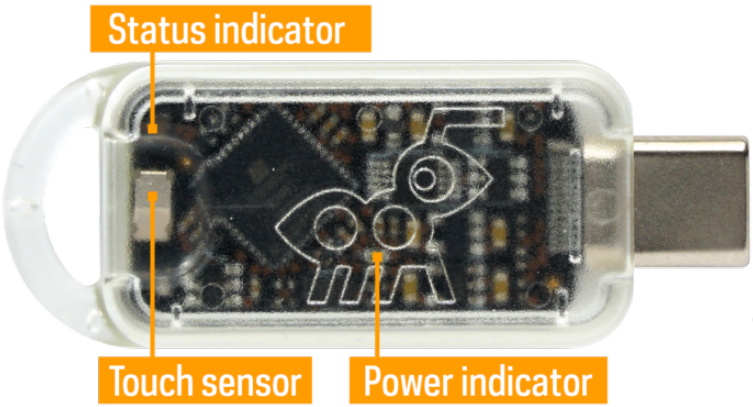
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Experimental Security Evaluation of the Tillitis TKey Authentication Ecosystem

Bachelor's thesis in Computer science and engineering

Josef Eidstedt
William Ekerstedt
Nadia Farias
Jonas Houltz
Tiam Khaniporshokouh
Emelie Skepö

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Bachelor's thesis 2026

Experimental Security Evaluation of the Tillitis TKey Authentication Ecosystem

Josef Eidstedt
William Ekerstedt
Nadia Farias
Jonas Houltz
Tiam Khaniporshokouh
Emelie Skepö

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Experimental Security Evaluation of the Tillitis TKey Authentication Ecosystem
Josef Eidstedt William Ekerstedt Nadia Farias Jonas Houltz Tiam Khaniporshokouh
Emelie Skepö

Supervisor: Rhouma Rhouma, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering
Co-examiner: Irum Inayat, Department of Computer Science and Engineering

Bachelor's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Experimental Security Evaluation of the Tillitis TKey Authentication Ecosystem
Josef Eidstedt, William Ekerstedt, Nadia Farias, Jonas Houltz, Tiam Khaniporshokouh, Emelie Skepö

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this project, a structured security evaluation of TKey-based authentication systems was conducted using an ethical hacking methodology, consisting of reconnaissance, scanning, vulnerability assessment, exploitation, and exploit testing. The results showed that while TKey mitigates several weaknesses of password-based systems, insecure software implementations can still introduce serious vulnerabilities, such as insecure communication, information disclosure, and flaws in authentication logic. Overall, the study demonstrated that security depends not only on the technology itself but also on its design and implementation, highlighting the need for continuous security evaluation across all system layers.

Sammanfattning

I detta projekt genomfördes en strukturerad säkerhetsutvärdering av autentiseringssystem baserade på TKey med hjälp av en metodik för etisk hackning, bestående av faserna rekognoscering, skanning, sårbarhetsanalys, utnyttjande av sårbarheter samt analys efter intrång. Resultaten visade att även om TKey minskar flera av svagheterna hos lösenordsbaserade system, kan brister i mjukvaruimplementationen fortfarande introducera allvarliga sårbarheter, såsom osäker kommunikation, informationsläckage, obehörig åtkomst till interna tjänster och brister i autentiseringslogik. Sammanfattningsvis visar studien att säkerheten inte enbart beror på den använda teknologin, utan även på hur systemet designas och implementeras, vilket understryker behovet av kontinuerlig säkerhetsutvärdering av alla systemlager.

Keywords: Cybersecurity, Ethical hacking, Hardware security token, Passwordless authentication, Two-factor authentication, Vulnerability assessment.

Acknowledgements

During the time spent on this project we have received support from several people. We thank Rhouma Rhouma, senior lecturer at the CSE department at Chalmers University of Technology, our supervisor. He has shown unwavering support throughout this project. We also thank Sasko Simonovski, CEO of Tillitis AB, for the donation of and support using the TKey.

Josef Eidstedt, William Ekerstedt, Nadia Farias, Jonas Houltz, Tiam Khaniporshokouh, Emelie Skepö, Gothenburg, June 2026

Contents

List of Figures

List of Tables

Glossary

.

- **BLAKE2** - Cryptographic hash function
- **BLAKE2s** - Cryptographic hash function optimized for 8- to 32-bit platforms
- **Burp Suite** - Intercepting proxy for web application security testing
- **CDI** - Compound Device Identifier
- **CLI** - Command Line Interface
- **CORS** - Cross-Origin Resource Sharing
- **CSRF** - Cross-Site Request Forgery
- **DOM** - Document Object Model
- **DoS** - Denial-of-Service
- **DDoS** - Distributed Denial-of-Service
- **Ed25519** - A Public-key signature system
- **FPGA** - Field-Programmable Gate Array
- **Hash** - Cryptographic hash function
- **HTTP** - Hypertext Transfer Protocol
- **HTTPS** - Hypertext Transfer Protocol Secure
- **LAN** - Local Area Network
- **MITM** - Man-in-the-Middle attack
- **MongoDB** - Document-oriented NoSQL database
- **Nmap** - Network Mapper security scanner
- **NoSQL** - Not Only SQL database
- **PostgreSQL** - Relational database management system
- **RAM** - Random Access Memory
- **React** - JavaScript library for building user interfaces
- **Redis** - In-memory data structure store used as a database or cache
- **Salt** - Cryptographic data used as additional input to a hash function
- **Session** - A temporary exchange of information between a client and server
- **SQL** - Structured Query Language
- **SSH** - Secure Shell
- **TOTP** - Time-based One-Time Password
- **UDS** - Unique Device Secret
- **USS** - User-Supplied Secret
- **Wireshark** - Network protocol analyzer
- **XSS** - Cross-Site Scripting

1
Introduction

1.1 Background

Password-based authentication is a method commonly used to verify that users have proper credentials. However authentication through the use of passwords has some well-known limitations. Users often tend to reuse passwords, and passwords are also subject to phishing attacks [1]. In order to overcome such limitations, passwordless authentication has emerged, where some methods are based on the use of security tokens [2].

The company Tillitis has developed a hardware authentication token called the TKey, which is used for passwordless authentication [3]. It is a small device resembling a USB thumb drive. When an application requires authentication from a user, the user inserts the device into a USB port on the computer as part of the authentication process. If the user is previously registered by the application, they are then allowed access.

Recent bachelor's theses at Chalmers University of Technology [4], [5], referred to throughout this report as *Project one* and *Project two*, have demonstrated that the Tillitis TKey can be used to build functional passwordless authentication systems. However, these implementations were not subjected to a structured or comprehensive security evaluation.

Passwordless authentication systems, especially those utilizing secure hardware keys such as the TKey, have proven to reduce vulnerabilities to authentication [6]. However, without a structured security assessment, it remains unclear if these systems truly provide stronger security guarantees or if weaknesses in software design and system architecture may undermine the benefits of a passwordless security system. Since security relies not only on the hardware itself but also on its integration with software [7], it is critical to analyze the entire authentication chain, as well as the security of individual components, to identify potential attack vectors. This ensures a comprehensive evaluation of the system's robustness under realistic conditions.

1.2 Project Goals

While previous bachelor projects have developed challenge-based authentication systems for web browsers using TKey technology, they have done so without a thorough security evaluation. This lack of systematic analysis highlights a gap in understanding the actual security of such systems. Therefore, the aim of this project is to perform a structured security assessment of the passwordless authentication systems developed in Projects one and two, as well as the security properties of the underlying TKey technology.

To fulfill this aim, the project seeks to answer the following research questions:

- To what extent have the passwordless authentication systems developed in Projects one and two been correctly implemented according to industry-standard security protocols?
- What vulnerabilities, if any, exist in the passwordless authentication systems, and how do these vulnerabilities affect their overall security compared to traditional password-based authentication systems?
- Does a passwordless authentication system provide security benefits compared to traditional password-based authentication systems despite any identified vulnerabilities?

1.3 Scope

This project does not focus on redesigning or extending the analyzed applications, but rather on analyzing their security properties, identifying potential weaknesses, and assessing their resilience against common and realistic attack scenarios. Countermeasures and fixes are recommended where possible, but are not considered the main objective of the security assessment.

This security evaluation follows the threat model defined by Tillitis for the TKey ecosystem [8]. The purpose of the threat model is to specify the scope of the protection of the TKey, and which components are assumed to be trusted. It also describes previously identified vulnerabilities, which are therefore considered out of scope for this project.

The threat model presumes that the TKey hardware and its associated key-derivation and cryptographic components behave exactly as defined in the design specifications [8]. Attacks requiring physical modification of the device, hardware fault injection, power analysis, electromagnetic side-channel analysis, or other hardware-level attacks are therefore considered outside the scope of this project.

The project will not test the security aspects of cryptographic primitives such as the ed25519 cryptographic protocol or the BLAKE2 hash function. These have already been tested by others, and are proved to be resistant to attacks [9], [10]. Additional tests would provide little new insight and are therefore considered out of scope.

The focus of this work is instead placed on the surrounding authentication ecosystem, including the web applications, backend services, communication channels, authentication logic, challenge-response implementation, session handling, and the interaction between software components and the TKey device.

2

Theory

This chapter introduces key components and concepts within the project that might be unfamiliar to the common reader. The first two sections cover the basics of the TKey hardware and the design of the tested projects, respectively. This is followed by general descriptions of common vulnerability types that are mentioned throughout the security assessment.

2.1 The TKey

The Tillitis TKey is a USB device that resembles a standard thumb drive [3]. It features a USB-C connector that is used to interface with a host computer. The device also includes a touch-sensitive area, which is used to physically assert user presence (figure 2.1). The primary purpose of the TKey is to provide a secure execution environment for small applications. Examples of such applications include Secure Shell (SSH) agents, digital signature tools, and the generation of cryptographically secure random values.

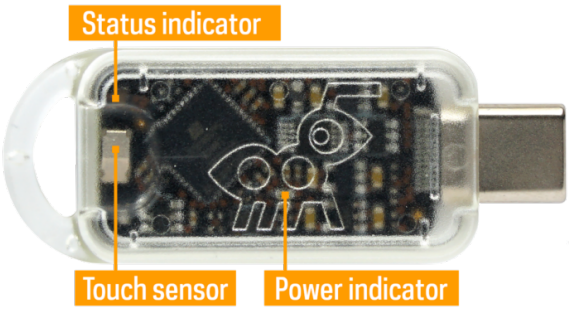


Figure 2.1: TKey interaction points by Tillitis AB is licensed under CC BY-SA 4.0.

Each TKey device contains a unique secret called a Unique Device Secret (UDS) [3]. It is used together with the application on the device and, if needed, an additional secret value from the user in the form of a User-Supplied Secret (USS) to generate cryptographic keys that are unique to each application. It does this by creating a Compound Device Identifier (CDI), by hashing the UDS, the hash of the device application and the USS together (figure 2.2). The hash function used is BLAKE2s. This ensures that the keys can only be generated when the hardware, the application, and the USS are used together.

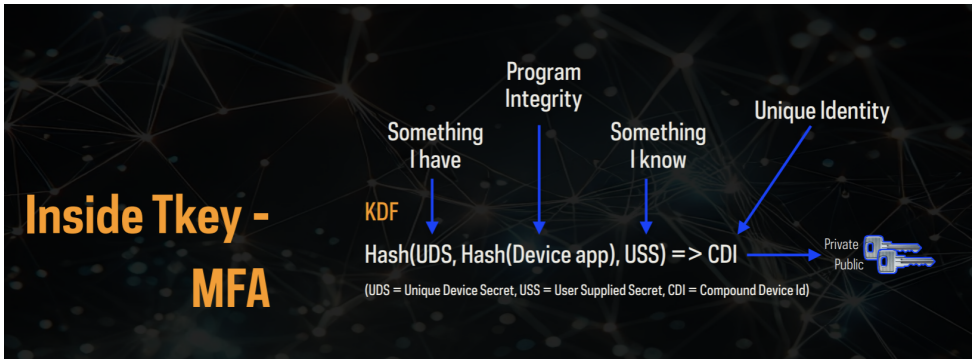


Figure 2.2: CDI generation [11]. Reproduced with permission.

A fundamental feature that enhances the TKey's security is the use of measured boot [3]. The binary of the application loaded onto the device is hashed, and this hash becomes part of the derived CDI. The CDI therefore ensures that if an application loaded onto the TKey is altered in any way, the cryptographic output of the TKey is also fundamentally changed, as the changes will result in a new binary that creates a different hash. Consequently, an attacker cannot manipulate the source code to steal existing keys, as the altered code would generate an entirely different set of keys.

Furthermore, the TKey does not contain any readable or user-modifiable persistent memory [3]. Instead, it relies on a small volatile memory module to store the active application. Each time the TKey is connected to a power source, its memory is completely empty, requiring the host computer to load an application onto it. This architecture ensures that no cryptographic keys, passwords, or sensitive application states are stored persistently on the device.

A key part of TKey is that both the hardware and software are open source [3]. This makes the ecosystem transparent and possible to review from a security perspective. Each of the components can be inspected and tested, which increases security and reduces the risk of hidden vulnerabilities.

2.1.1 Challenge-response mechanism

A common method for authenticating users without relying on traditional passwords is the challenge-response mechanism. The protocol begins when a user initiates an authentication request, to which the application's web server responds by issuing a unique "challenge" [12]. This challenge typically consists of a random string or a nonce, which is often combined with a timestamp to prevent replay attacks.

The challenge is sent to the client, where it is signed using a cryptographic function and the user's private key. The client then transmits this signed challenge back to the web server as a "response". The server verifies the cryptographic signature using the user's corresponding public key, which is retrieved from the web server's database or verified upon receipt. If the signature is valid, the server confirms the legitimacy of the user, establishing that only the holder of the secret private key could have generated the response.

In this project, the TKey operates with a dedicated signer application loaded into its volatile memory. The device utilizes this application alongside internally derived private and public keys to cryptographically sign the challenge provided by the server, as seen in figure 2.3.

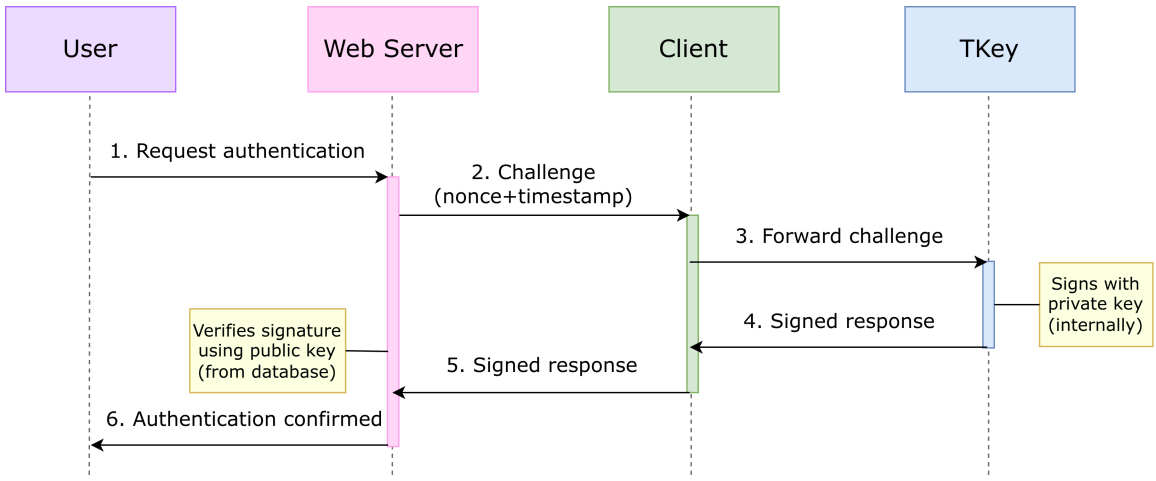


Figure 2.3: Challenge-response authentication flow with TKey.

2.1.2 Tillitis TKey threat model

The Tillitis threat model describes the security assumptions and limitations of the TKey, and clarifies what types of attacks the device is designed to withstand [8]. At a high level, the model assumes that the Field-Programmable Gate Array (FPGA) inside the TKey forms the core security boundary, while external components such as the USB controller and the host computer are outside the security boundary. The most important secrets on the device, such as the UDS and the CDI, are therefore kept strictly inside the FPGA and are never exposed to the outside world.

The threat model distinguishes mainly between software-based attacks, which come from the host computer, and hardware-based attacks, which require physical access to the device [8]. The TKey is designed to resist software attacks by enforcing strict rules on how applications run, how memory is accessed, and how communication with the device is handled. Hardware attacks, such as physically opening the device or manipulating the electronics, are considered out of scope because they require specialized equipment and physical access.

In addition, the threat model outlines which types of attackers the TKey is expected to handle [8]. This spans from unskilled users trying random guesses, to more advanced attackers with significant technical knowledge. Overall, the threat model provides a clear boundary for what the TKey protects against and helps define the assumptions used when evaluating its security.

2.2 System architecture

This section provides an overview of the system architectures of Project one and two. These architectural descriptions highlight how design choices influence both functionality and the resulting attack surface. In addition, the section introduces the infrastructural concepts of proxy server and Docker. Together, these architectural foundations establish the technical background necessary for understanding the vulnerabilities and security considerations discussed later in the report.

2.2.1 System design: Project one

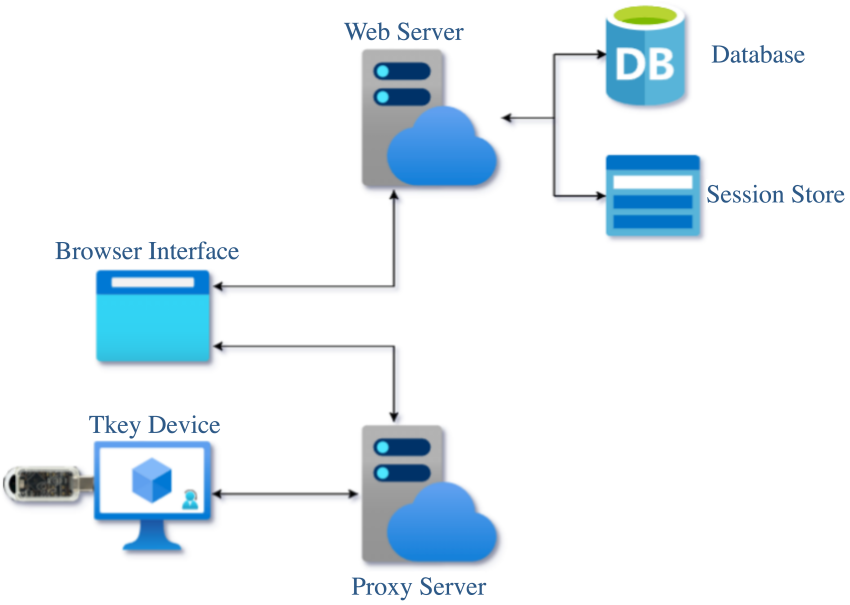


Figure 2.4: An illustration showing the system design of Project one [4].

As seen in figure 2.4, Project one has a system design that consists of the following components:

- **Web-interface:** Implemented using HTML, CSS, and Javascript [4]. It was implemented with a focus on back-end authentication mechanisms and hardware integration. The front-end design was kept minimal to avoid unnecessary dependencies.

- **Proxy-server:** Handles communication between the TKey and the browser interface [4]. It is responsible for detecting and establishing connections to the TKey, loading the authentication application onto the TKey, and managing the transfer of cryptographic challenges and signatures between the TKey and the browser interface. It was implemented with the Go programming language to simplify integration, as the software within the TKey was also written in Go [13].

- **Web-server:** Contains the database and the session store [4].

- **Database:** The database stores user-related information necessary for authentication [4]. It also stores data related to account recovery procedures. It was implemented using PostgreSQL due to its advanced security features and general robustness.

- **Session store:** The session store was used as an alternative to storing session data in browser cookies [4]. This was done using an in-memory storage solution called Redis. Instead of cookies, a reference or token was provided to the user, while the session data was maintained on the server in the session store. It was implemented to improve security regarding session life cycles and revocation of expired sessions. They were implemented using Python with Flask as the web framework due to their ease of use and support for secure handling of web requests.

Furthermore, the web-server, the database, and the session store were containerized using Docker and Docker compose [4].

2.2.2 System design: Project two

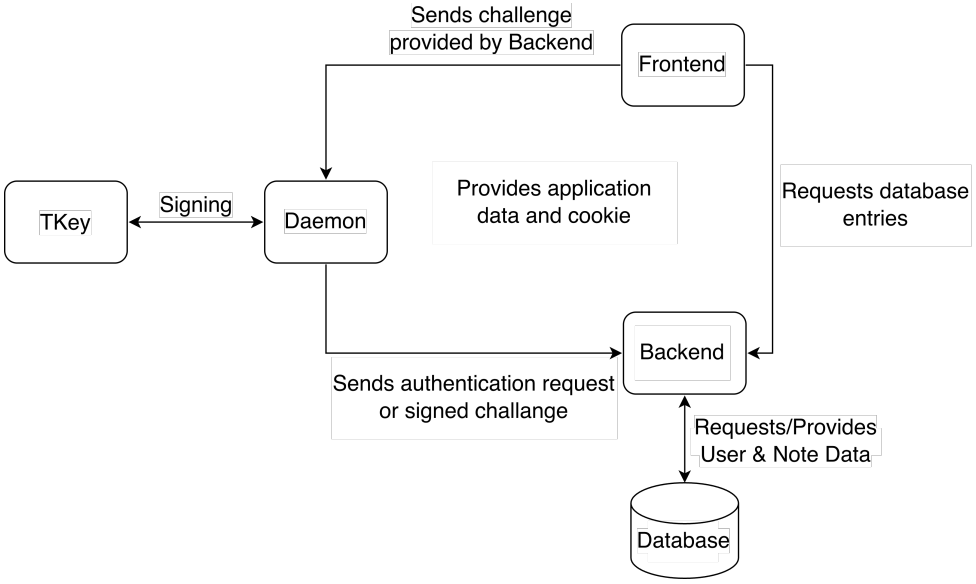


Figure 2.5: An illustration showing the system design of Project two [5].

Project two was built with a layered approach that enforced separation of concerns, focused on modularity, and simplified parallel development [5].

Illustrated in figure 2.5, the system architecture of Project two consists of the following key components:

- **Front-end:** Was implemented using React as it enabled to structure the user interface into reusable modules [5].

- **Daemon:** Enabled users to authenticate using the TKey device [5]. The daemon communicates with the application through the requests *login*, *register*, and *add public key*.

- **Back-end:** Was implemented using Go and was built using a four-layer architecture to simplify the development process [5]. The presentation layer handles request processing and security validation. The application layer handles business logic for authentication and cryptographic operations. The domain layer defines domain entities that the application layer interacts with. Database interactions are kept independent of other components in the data access layer.

- **Database:** The database was implemented using MongoDB, a document-oriented NoSQL database [5]. The database contains two collections: user notes and users. It stores every username and their public keys (label and key) in the collection belonging to that user. In the user notes collection, the username of the user that created the note, the name of the note, and the note itself are stored.

2.2.3 Proxy server

A proxy server handles client requests by forwarding them to other servers and relaying the response back to the client [14], as illustrated in figure 2.6. It is commonly used for routing, load balancing, or isolating internal services from direct external access. When properly configured, the proxy acts as a security layer by filtering and controlling traffic. Incorrect configuration of the proxy server can instead expose internal functionality directly to attackers.

Figure 2.6: Request and response flow through a proxy server.

Examples of such incorrect configurations are open proxies that lack access control, improper routing that exposes internal services, and insecure communication between the proxy and backend servers [15]. Furthermore, incorrect handling of request headers can compromise access control decisions, potentially leading to unauthorized access [16].

2.2.4 Docker

Docker is a containerization platform used to run different components of a system in isolated environments. Docker provides the ability to package and run an application in a loosely isolated environment called a container [17]. The isolation and security let you run many containers simultaneously on a given host. However, if not properly configured, it can increase the system's attack surface instead of improving security, and may expose sensitive resources to unnecessary access.

Docker uses the Linux file permission system. File permissions in Linux are of the form -*rwxrwxrwx*. The first *"-"* states the file type, the first set of *rwx* describes what permissions apply for the owner of the file, the second set describes what permissions apply for the user group that owns the file, and the last set describes what permissions apply for everyone else. The letter *r* stands for *read-permission*, and grants a user the right to access the contents of a file, and also allows the creation of copies of files. The letter *w* stands for *write-permission*, which grants users the right to modify the contents of a file. The *x* stands for *execute-permission* and allows users to execute the contents of a file.

2.3 Common vulnerabilities

This section provides an overview of common vulnerabilities that are frequently encountered in modern systems and applications. Each subsection introduces a specific vulnerability category, explains the underlying principles, and illustrates how such weaknesses typically manifest in practice.

2.3.1 Broken logic

Broken logic vulnerabilities are present if it is possible to manipulate the intended functionality of an application [18]. If an application behaves like a shopping site, it could suffer from insufficient restriction of client-side controls. In this specific example, the application suffers from a broken business logic vulnerability that enables users to change the price they pay for the products in their shopping cart. This can be demonstrated by selecting an expensive jacket that costs *1337$*, and adding it to the shopping cart, as seen in figure 2.7. Once the jacket is added, the attacker starts checking the network traffic to the site. When clicking on the icon of the shopping cart, the resulting post-request reveals the price for each selected item. By copying that request and changing the item price, the attacker can successfully buy items much cheaper than the site originally intended and thereby break the business logic of the site, as shown in figure 2.8 [19].

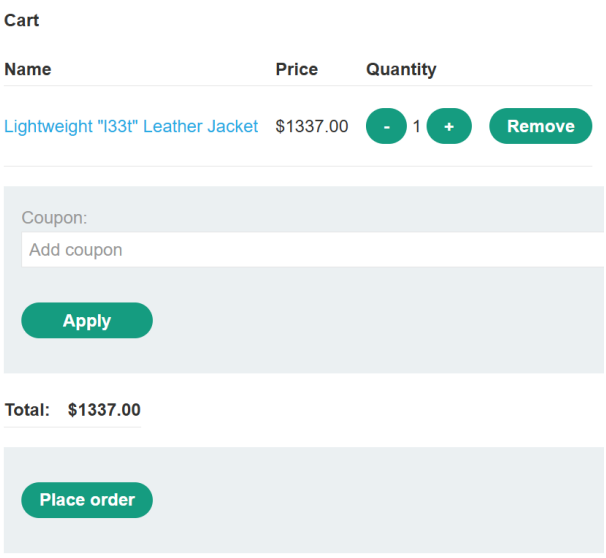


Figure 2.7: The attacker adds an expensive item, in this case a jacket for *1337$* to the cart.

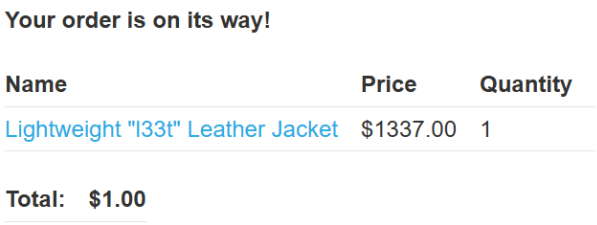


Figure 2.8: An attacker can successfully change the total price paid. After changing the price they continue to checkout like a normal customer. Making an expensive item cost only a fraction of the intended price.

2.3.2 Information disclosure

Information disclosure is when the system gives away sensitive information to any user [20]. Information disclosure comes in many forms, such as the extreme case where an application gives away the password of a user, or more subtle cases, such as verbose error messages. For example, a database error message that reveals the underlying Structured Query Language (SQL) server version and internal file paths provides an attacker with valuable reconnaissance information.

In the following example, the tested application suffers from information disclosure vulnerabilities in its error messages [21]. The application behaves like a shopping site, where each item sold is identified by an id number, which can easily be seen by observing the url after clicking on the *View details button* belonging to a specific item. For each item, the url ends with *product?productId=X*, where *X* is the id number of that item. If instead the number is replaced with a letter *a*, the application crashes and throws the error message displayed in figure 2.9. The error message displays a full stack trace in its response, revealing that the application runs *Apache Struts 2 2.3.31*, which can then be searched for known vulnerabilities on the internet in an attempt to compromise the application.

2.3.3 Denial of Service

Denial of Service (DOS) vulnerabilities imply that it is possible to make something unavailable to benevolent users [22]. This can be done by exhausting resources or by causing it to become unresponsive. DOS attacks come in many forms, such as crashing an application, dumping all the data an application has stored, or by drowning a server in fake requests to make it unavailable to genuine requests. Another common term related to DOS is Distributed Denial Of Service (DDOS). DDOS is a variant of DOS attack, where the only difference between a DDOS and a DOS attack is that the DDOS is distributed; that the attack comes from multiple source hosts, rather than a single host, as illustrated in figure 2.10.

Figure 2.9: Providing invalid input instead of an id number results in a crash that responds with a full stack trace. The important part is that it reveals that the application runs version *2 2.3.31* of Apache Struts, which can the be checked for vulnerabilities by attackers.

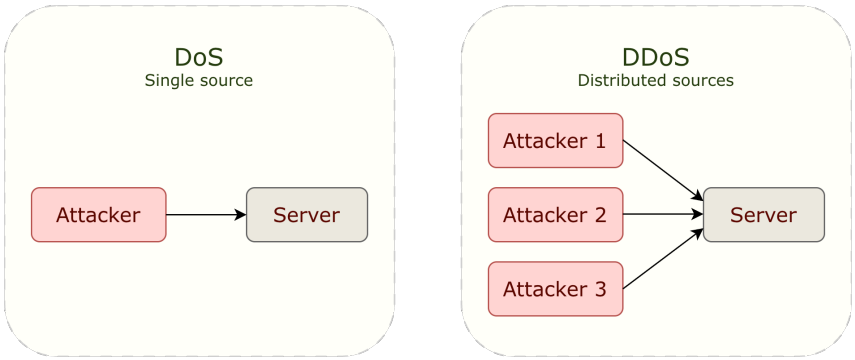


Figure 2.10: The difference between DoS and DDoS attacks, showing single versus multiple attack sources.

2.3.4 Brute Force vulnerability

A Brute Force attack consists of an attacker guessing sensitive information such as login credentials or encryption keys [23]. The attacker tries all different possible combinations until a correct value is found, usually with the help of computers to try a large number of combinations. An application is therefore vulnerable to a brute force attack if it is possible to repeatedly try new attempts until a correct combination is found.

If a webpage has insufficient protection against brute force attacks, an attacker can make a seemingly benign attempt to login, catch the resulting request when it is sent, and then use that request to make repeated attempts with different input. Furthermore, an

attacker could create lists of likely usernames and passwords to completely automate the attack, which can then be used to try all likely entries and thereby perform a brute force attack.

2.3.5 Cross-Site Scripting

Cross-Site Scripting (XSS) attacks are a type of injection, where malicious scripts are injected into otherwise benign and trusted websites [24]. A XSS attack occur when an attacker uses a benign web application to inject malicious code, generally in the form of a browser html script, that will be read by the browsers of other users without their consent.

Reflected XSS occurs when an application receives data in requests and includes that data in the resulting response in an unsafe way [24]. If the application has an input field like the one in figure 2.11, one can attempt to provide input that could be interpreted as HTML-code. If the code is executed, rather than treated as text, the application is vulnerable to reflected XSS. One way to test for this vulnerability is to use an html-script along with the *alert* command. Such a script will result in a popup-window if it was executed as code. In figure 2.12 the vulnerable example application was given the input

```
{<script>alert(document.cookie)</script>}
```

which resulted in a popup window that contained the session-cookie.

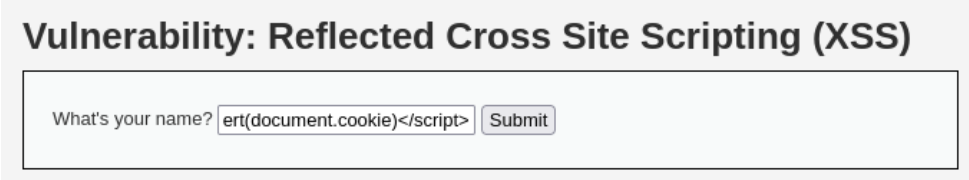


Figure 2.11: An example of an application that is vulnerable to Reflected XSS.

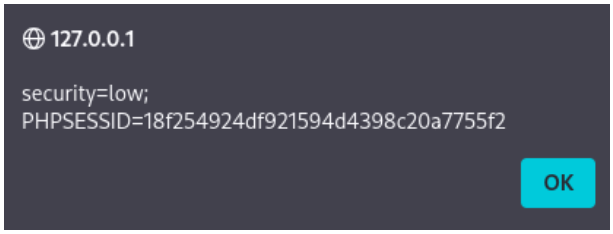


Figure 2.12: Reflected XSS alert in the form of a popup message with the session cookie.

Stored XSS works similarly to reflected XSS by exploiting unsanitized input, but instead posts its payload to a database, where it persists as an entry [24]. If the application uses a guestbook like in figure 2.13 where the resulting entries are shown on the page, it might be vulnerable to stored XSS.

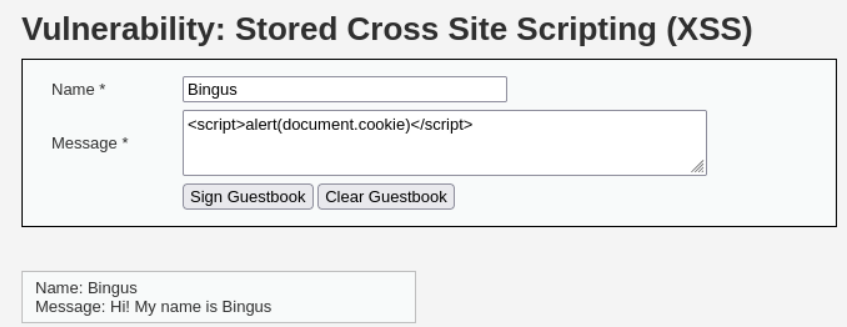


Figure 2.13: An example of an application with a guestbook feature that is vulnerable to stored XSS. A payload containing a script with the *alert* command is provided as input in the writing field of the guestbook. New entries are displayed on the bottom of the page.

Entries are displayed on the guestbook page after being written. Consequently, entries containing HTML-scripts will also be displayed. If the input is insufficiently sanitized, this will result in the code being executed by the browser of any user who accesses the guestbook, without that user ever having consented to what the script does. In figure 2.13 the same payload was provided as for the reflected XSS scenario, resulting in the same alert message containing the session cookie.

2.3.6 Man-in-the-middle Attacks

Software is vulnerable to Man-in-the-middle (MITM) attacks when communication between different program entities is vulnerable to eavesdropping or unauthorized tampering [22]. Such attacks can be passive (only observe) or active (observe and manipulate), as illustrated in figure 2.14. The goal of such an attack could be to gather sensitive information such as user credentials, cookies, or other data that can be used to perform other types of attacks.

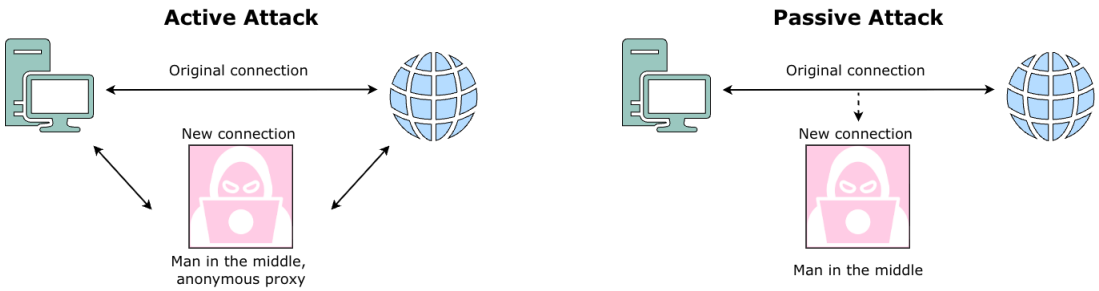


Figure 2.14: Conceptual overview of active and passive MITM attacks, illustrating the interception of an original connection.

2.3.7 SQL-injection

SQL injection attacks are a type of injection attack where SQL commands are injected through input data from the client into an application with the intention of affecting the execution of predefined SQL commands [25]. A successful SQL injection exploit can read

sensitive data from the database, modify data within the database, or execute administrative operations on the database itself.

An example of what SQL-injection might look like can be seen in figure 2.15. In this example there is an application that contains a form field called *User ID*. The form field uses a query that looks like:

{SELECT first_name, last_name FROM users WHERE user_id = '\$input'}

Providing proper input into the form field results in a response containing the name of the user with that corresponding id, as seen in figure 2.15.

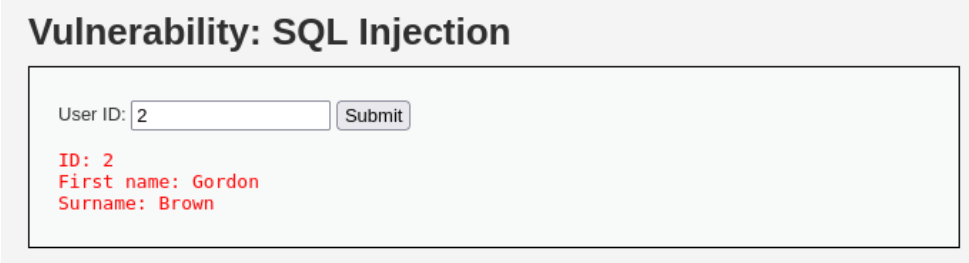


Figure 2.15: An example of an application that is vulnerable to SQL-injection. Providing *2* as input returns the name "Gordon Brown".

If the form field has insufficient filtering of the provided input, it might be possible to exploit it by providing input that results in an "all-true" scenario. A payload that looks like *1' OR '1'='1* would close the original condition and add an *OR* statement that is always true. This results in the database returning the information of all users, as shown in figure 2.16.

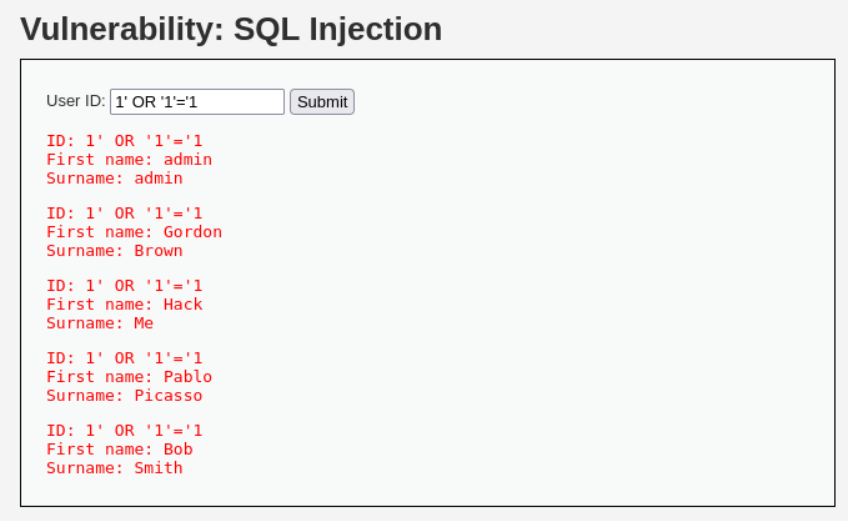


Figure 2.16: The provided input *1' OR '1'='1* is always true, which results in the database returning all entries.

2.3.8 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated [26]. If the victim is a normal user, a successful CSRF attack could force the user to perform state changing requests like transferring funds or changing their email address. If the victim is an administrative account, CSRF could compromise the entire web application.

This can be shown in an example where an application has a form to change the admin password. The form contains fields for *New Password* and *Confirm Password*, as seen in figure 2.17.

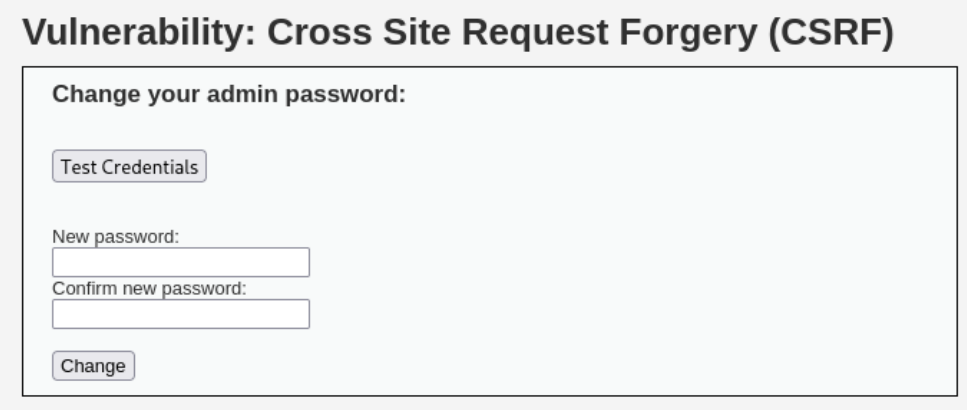


Figure 2.17: An application with a form to change the admin password.

If the application lacks proper protection against CSRF, the form could be exploited to force the user to perform unintended actions without their consent [26]. In figure 2.18 an attacker has intercepted the resulting request from a genuine attempt to change a password to *123*, taken that request, and transformed it into HTML-code.

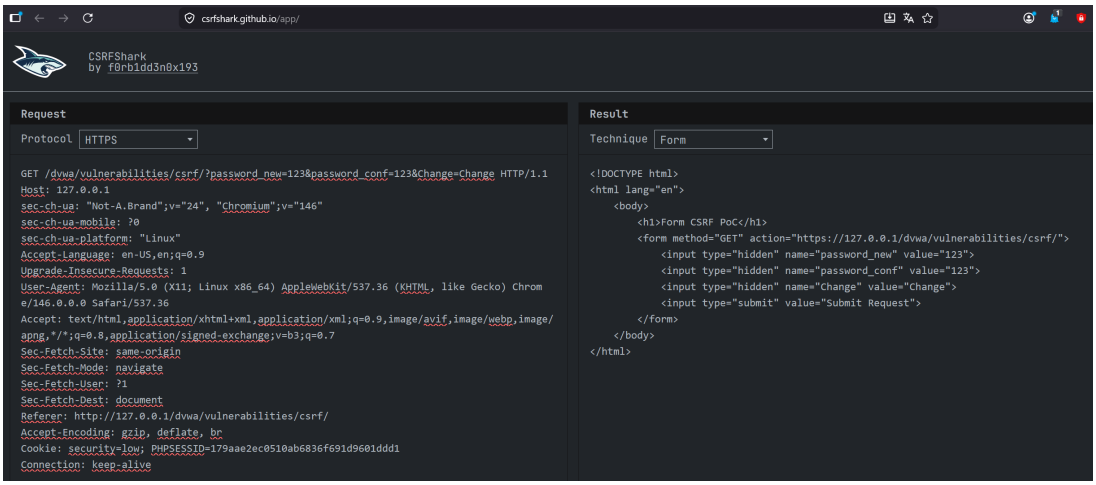


Figure 2.18: A request (left) is intercepted and transformed into HTML code (right) in order to generate a webpage that can be used to send requests.

The attacker could then paste that code into a text editor to reshape it based on the actions they want to execute on behalf of the victims when the exploit is triggered. In this example, the attacker uses vim to create a file called *index.html* (figure 2.19), where the password is changed to *YouRHacked* instead of *123*, as shown in figure 2.20.

Figure 2.19: The attacker can further change the intercepted code based on what changes they want to make on behalf of the victim.

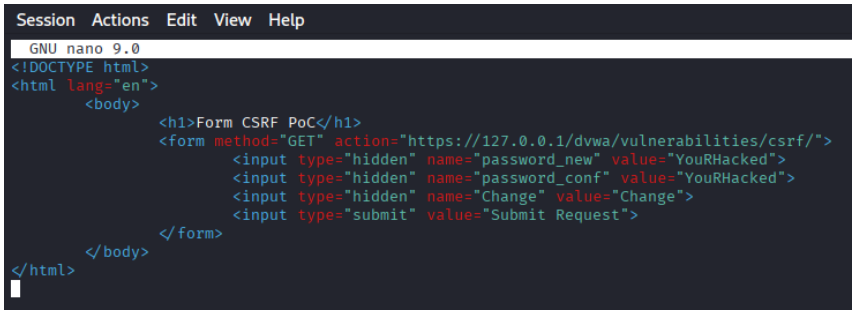


Figure 2.20: The code is changed to send requests containing a different password.

The attacker can then use the file to generate an alternative site to send requests from (figure 2.21). In this case the attacker creates a webpage called index.html, as shown in figure 2.22.

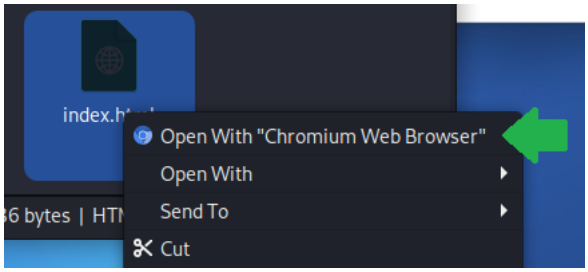


Figure 2.21: The resulting file is used along with a web browser to send requests to the application.

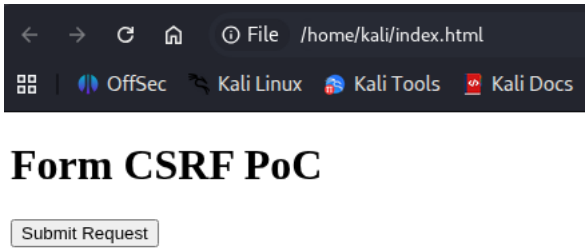


Figure 2.22: The file creates a webpage with a submit button.

The resulting site contains a blank page with a submit button. Pressing the button will execute the code in the created file. Any request sent from it to the application will be treated as genuine, as the application has no means to protect itself against CSRF.

3
Methodology

The methods used in this security assessment followed the process used in ethical hacking. The process consists of five steps: reconnaissance, scanning, vulnerability assessment, exploitation, and post exploitation [27], as illustrated in figure 3.1.

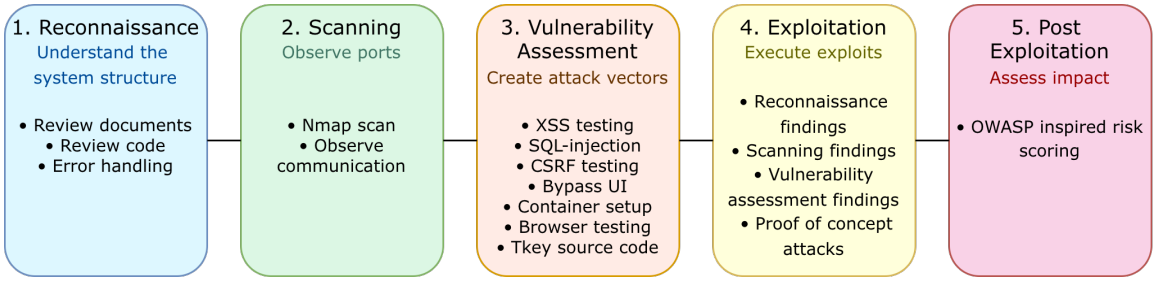


Figure 3.1: The five steps of the ethical hacking methodology used in this security assessment.

The reconnaissance step investigated the structure of the targets and established what the projects consisted of. In this case, this included the expected function of the TKey and the authentication applications and their internal structure. The scanning step performed scans to give a good picture of the layout of the ecosystem of each project. The vulnerability assessment step then investigated if there were any vulnerabilities present or other misconfigurations that might be used to develop exploits. The exploits step attempted to use the information gathered during the reconnaissance-, scanning-, and vulnerability assessment-steps to produce attack vectors, along with a proof of concept of such attacks. The post exploitation step finally analyzed the result of the attacks and described the impact of the found exploits in terms of risk, along with relevant countermeasures to prevent an attacker from using the identified weaknesses.

The developers behind the analyzed projects received the same task description when making their applications, but the groups had chosen quite different approaches when solving it. This meant that the applications had different prerequisites in order to run, and would not necessarily be compatible with all or even the same browsers and operative systems.

3.1 Reconnaissance

After installing all prerequisites and successfully running the applications, the reconnaissance step began by reviewing the developer reports for the two projects. This was done

to identify any previously discovered vulnerabilities and to determine whether mitigation attempts had been made. This information was later used to verify the existence of these vulnerabilities and assess the effectiveness of the proposed mitigations.

Documentation related to the TKey and its threat model were also reviewed during this step, as it provided an understanding of the protection mechanisms and their limitations. In addition, a walkthrough of each project's code was conducted together with one of the original developers to better understand the system architecture and intended behavior. Since the code was open-source and publicly available on GitHub, it was necessary to examine it in detail, as attackers would have the same access. This included analyzing the code, checking comments for useful information, and reading through README files.

The behavior of the applications was then analyzed to ensure that they functioned correctly and that any findings were not caused by faulty installation or execution. This was followed by an evaluation of their error handling capabilities, as improper handling or crashes could lead to information leakage. Tests at this stage involved unexpected but non-malicious inputs, such as empty usernames, login attempts without a TKey, or with an incorrect TKey, or two TKeys, in order to observe whether the system handled these cases without failure.

Finally, approaches that an attacker might take without using the intended user interface were investigated. This involved the use of specialized tools such as Nmap to identify open ports and exposed services that belonged to the projects, as well as to verify the accuracy of the provided documentation. These tests were conducted locally to eliminate the influence of external firewalls and to ensure that the results were not affected by external factors.

3.2 Scanning

The scanning step was initiated after the reconnaissance step. The scanning step utilized the tool *Nmap* to scan a host computer running one of the projects. This was done to observe what ports different components were running on, and observe what other information could be obtained through such scans. Scans were also performed using the tool *Wireshark* to observe communication between different components when different requests were made. This was done to see if the communication was properly protected, if communication followed the intended behavior of the program, and what protocols were used by the different communication requests.

3.3 Vulnerability Assessment

The vulnerability assessment process was initiated once the scanning step had been completed. The goal of the vulnerability assessment part was to gather information about the different components of the applications, with the intention of using that information to construct exploits to use in attacks.

3.3.1 Testing for XSS

Testing was conducted using Burp Suite to capture and modify HTTP requests, while Chrome DevTools was used to inspect the application and execute JavaScript in the browser console.

Attempts to find reflected XSS were made by observing where the applications might display user provided input when rendering the user interface in the browser. If such places had vulnerabilities against reflected XSS, they might treat input as executable code when the user interface is rendered. This was tested by adding a payload

```
<script>alert(1)</script>
```

to a query parameter, for example:

```
http://localhost:3000/settings?test=<script>alert(1)</script>
```

If the query was vulnerable to XSS, the result would be the webpage showing an alert-popup, and nothing if there was no XSS vulnerability present.

Tests were also conducted to investigate whether the projects were vulnerable to stored XSS. This was done by providing the same payload as for reflected XSS, but in this case the database was supposed to first store the payload and later trigger the XSS when requested to retrieve the payload in another query.

3.3.2 Testing for SQL-injection

Since the code used in the projects was open source, attempts to find SQL-injection vulnerabilities started with analyzing all the code that handled user input. If the code properly removed all possibilities to write input that could be treated as being part of a query, rather than a parameter, then SQL-injection could not be performed. Tests like providing the common input *1' OR '1'='1* were made in an attempt to break out of parameters and cause injected code execution. Similar attempts were also made where the input was crafted based on the observations of the code. In Project one, attempts were also made to automate the process of establishing a reverse shell through the use of the tool *sqlmap*. This was done by first capturing the resulting requests from functions that contained input parameter fields within the application. The parameter whose input field was to be tested for sql-injection was then marked with an asterisk *(*)* by using a text editor, and the name of the parameter was saved for later use. The asterisk was placed to indicate to sqlmap where the parameter it was supposed to test resided in the request. The altered request was then saved as a .txt file to be used by sqlmap. This file was then referred to in the final command, where it was supposed to attempt sql-injection. The resulting commands were of the form:

```
sqlmap -r request.txt -p PARAMETER\_NAME -{}-batch --os-shell
```

3.3.3 Testing for CSRF

Testing for CSRF was performed using Burp Suite by intercepting and replaying POST requests to state-changing endpoints. Requests were sent both with and without a CSRF token, as well as with modified token values, by using Burp Suite Repeater. The CSRF token was removed or replaced in the request before being resent to the server. The

requests were executed against relevant application endpoints to observe their handling of CSRF-protected requests. All tests were conducted on authenticated sessions using captured HTTP traffic.

3.3.4 Testing attack vectors that avoided the user interface

Attempts were also made to interact with the applications without going through the user interface. This was done by attempting to send requests directly to other parts of the applications than the user interface. HTTP requests sent to the database by the applications were observed in order to attempt to make imitations using third party tools.

3.3.5 Testing container setup

Project one used Docker as a means to run the project in an isolated environment. As a result, the security assessment included an analysis of the configuration of the files inside the Docker setup. The analysis was performed manually through the Docker app and checked for overly permissive file configurations.

3.3.6 Testing through the browser

Attempts were also made to gather information about the two projects through the use of web-developer tools and through the use of the url. The web-developer tools provided in the Mozilla Firefox browser were used in an attempt to gather sensitive information from the browser pages of the application. There were also tests where the url was changed in an attempt to gain access to pages that the application requires authentication for.

3.3.7 Testing the TKey

The TKey source code was also open source and would therefore also be available to any malicious user. Therefore, a thorough examination of the TKey source code was also conducted. This consisted of the examination of four distinct components: the TKey firmware, the signer application loaded onto the device, the TKey client for device communication, and the signer client used to interact with the signer application via the TKey client.

The testing was focused on sections of the code that were identified as of particular interest. Examples include segments that appeared to contain logical flaws or those suspected to be susceptible to unexpected system events.

Additional testing focused on edge cases and user interactions. This included attempting to log in to the same account using different TKeys, unplugging the device during active signing processes, logging in or registering with more than one TKey plugged in, and ignoring user-presence requests (touching the sensor). In essence, the TKey was subjected to exploratory testing to simulate both intentional and unintentional misuse by a real-world user.

3.4 Exploitation

The exploitation step attempted to use the information gathered during the reconnaissance, scanning, and vulnerability assessment steps to create exploits that could be used by an attacker. Some vulnerabilities could potentially allow attackers to perform exploits on their own, while other vulnerabilities might be used together to perform more powerful exploits. Any identified exploits were tested and performed with a proof-of-concept demonstration of those exploits. One common goal for hackers is the retrieval of administrator credentials on the attacked host. Other malicious attacks may be stealing data, corrupting applications or even the operating system on the attacked machine. Such vulnerabilities are explored during this phase.

3.5 Post Exploitation

Identified vulnerabilities were evaluated during this phase. The evaluation was performed based on the severity of the vulnerabilities and was graded using a project-specific risk evaluation model. This risk evaluation scoring model was inspired by the OWASP Risk Rating methodology, but was simplified and adapted to better fit the scope and nature of the two analyzed projects [28]. The model was made to provide a defensive evaluation of the vulnerabilities, where the severity of each vulnerability was described with an understandable review, with the additional goal of avoiding making it too complicated.

3.5.1 Risk Evaluation Criteria

In this model, each vulnerability was evaluated using two main criteria:

1. **Likelihood:** Likelihood represents how easily an attacker could exploit a vulnerability. It was evaluated based on the following two factors:
   - **Attack Complexity (AC):** The technical difficulty required to exploit the vulnerability. This included factors such as the need for specialized tools, specific conditions, or the overall complexity of carrying out the attack.
   - **Required Access / Privileges (RA):** The level of access required by the attacker to perform the exploit (e.g., no authentication, local network access, or physical access).

   To provide a balanced evaluation, the Likelihood score was calculated as:

$$\text{Likelihood} = \frac{AC + RA}{2}$$

2. **Impact:** Impact represents the potential consequences of a vulnerability on the system. It was evaluated based on the three core principles of information security (CIA):
   - **Confidentiality (C):** The extent to which sensitive information may be exposed.
   - **Integrity (I):** The extent to which an attacker may modify or manipulate data.

- **Availability (A):** The extent to which the system's functionality or availability may be disrupted.

Since a severe impact in any one of these dimensions could significantly compromise the system, the Impact score was defined as:

$$\text{Impact} = \max [\text{C, I, A}]$$

3.5.2 Final Risk Score

The overall severity of a vulnerability was determined by combining Likelihood and Impact. Following the general principles of risk assessment, the final Risk score was calculated as:

$$\text{Risk} = \text{Likelihood} \times \text{Impact}$$

3.5.3 Scoring method

In this model, each evaluation criterion was assigned a score in the range of 0 to 3. A higher score indicates a higher level of risk and a more severe security condition, while a score of 0 indicates that the vulnerability was either not present or not exploitable in the evaluated system.

1. **Likelihood Criteria:**
   (a) **Attack Complexity (AC):** This criterion represents the technical difficulty required to exploit a vulnerability:
      - **3 (Low complexity):** The attack can be performed easily without requiring special conditions or advanced tools.
      - **2 (Medium complexity):** The attack requires some specific conditions or moderate technical knowledge.
      - **1 (High complexity):** The attack is difficult to perform and requires advanced expertise or specific conditions.
      - **0 (Not exploitable):** Exploitation of this vulnerability is not possible in the system.
   (b) **Required Access / Privileges (RA):** This criterion represents the level of access required by an attacker to perform the exploit:
      - **3 (No privileges required):** The attack can be performed without authentication or special access.
      - **2 (Limited access):** The attack requires user-level access or local network access.
      - **1 (High privileges required):** The attack requires administrative or physical access.
      - **0 (Not applicable):** The attack path is not applicable or cannot be used in the system.

2. **Impact Criteria:**
   (a) **Confidentiality (C):** Measures the extent of sensitive information exposure.
      - **3 (High risk):** Exposure of sensitive data such as user data, credentials, or secrets.
      - **2 (Medium):** Exposure of limited or less sensitive information.
      - **1 (Low):** Exposure of non-sensitive information.
      - **0 (No impact):** No information is exposed.
   (b) **Integrity (I):** Measures the extent to which data can be modified or manipulated.
      - **3 (High risk):** Full modification of data or system control is possible.
      - **2 (Medium):** Partial modification of data is possible.
      - **1 (Low):** Minimal impact on data
      - **0 (No impact):** No data modification is possible.
   (c) **Availability (A):** Measures the impact on system availability:
      - **3 (High risk):** Complete system failure or denial-of-service.
      - **2 (Medium):** Partial disruption of functionality.
      - **1 (Low):** Limited impact on system performance.
      - **0 (No impact):** No effect on availability.

3.5.4 Severity Mapping

Based on the final Risk score, obtained by combining Likelihood and Impact, each vulnerability was classified into one of the following severity levels:

- **0 to $<$ 1 (No Risk):** No exploitable vulnerability or meaningful impact has been identified in the system.
- **1 to $<$ 3 (Low):** The vulnerability has limited impact or exploitability and does not pose a significant risk to the system.
- **3 to $<$ 5 (Medium):** The vulnerability may be exploitable under certain conditions and can have a noticeable impact on the system.
- **5 to $<$ 7 (High):** The vulnerability is relatively easy to exploit and may lead to serious security consequences.
- **7 to 9 (Critical):** The vulnerability is easily exploitable and has severe and widespread impact on the system.

This classification indicates that higher scores correspond to vulnerabilities that are both easier to exploit and have a greater impact on the overall security of the system. Vulnerabilities classified as High or Critical require immediate attention and mitigation.

4

Results

The results were divided into the main parts of ethical hacking, along with a section describing TKey specific findings. This chapter starts with the findings from the reconnaissance step, followed by the results from the scanning step, vulnerability assessment, exploitation, TKey-specific findings, and finally post exploitation where there are tables (4.2, 4.3, 4.4, 4.5) with risk evaluation score tables for each vulnerability found.

4.1 Reconnaissance results

The analysis of the source code in Project one revealed multiple issues that were causes of concern. The most obvious issue was that hard-coded credentials were found as shown in figure 4.1.

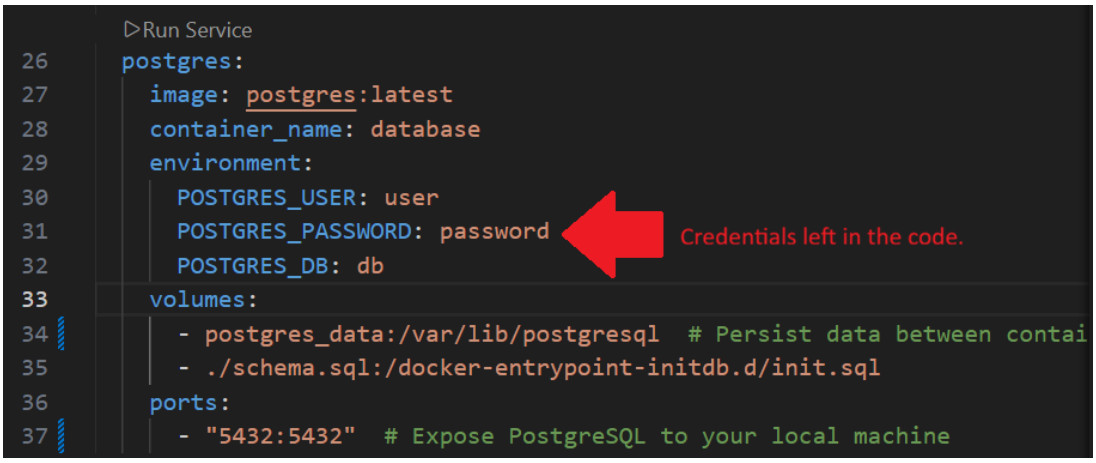


Figure 4.1: Hard coded credentials in the code [29]. Still valid for anyone who wants to use them.

Another issue found while observing the code was that the database in Project one stored all information in clear text (figure 4.2). An inspection of the database code revealed that the `"user"` table contained the following data fields: `id`, `username`, `publickey`, `secret`, `salt`, and `hash`. The data stored in several of these fields were therefore sensitive. The `username` could be used to identify and enumerate valid users. The `publickey`, while not secret, was part of the authentication mechanism and could help an attacker better understand the design of the system. The most critical field was `secret`, which represented the Time-based One-Time Password (TOTP) secret for a specific user. If the TOTP was compromised, an attacker could independently generate valid TOTP codes without access to the user's device, bypassing the two-factor authentication mechanism. The `salt` and `hash` fields also contained sensitive information related to credential storage, which could

potentially be used for attacks or offline analysis depending on the hashing scheme.

Figure 4.2: The code handling how the database stores its given data [30]. The data is sent unencrypted, to later be stored in the database without any further encryption.

In contrast, a review of the database code for Project two revealed a more cautious approach. The application in Project two used MongoDB with two collections: `users` and `user_notes`. The `users` collection stored each user's identifier, username, and a list of base64-encoded public keys associated with their hardware authentication device. In particular, no passwords, secrets, or hashed credentials were stored, as authentication was handled entirely through the hardware-based TKey. The `user_notes` collection stored notes in plaintext, containing the fields `username`, `name`, and `note`.

Initial tests of Project two revealed discrepancies that indicated unintended behavior. Although user registration and login worked as expected, several other features exhibited inconsistent behavior or failed intermittently. An example of this was that the logout feature sometimes failed silently, while at other times the user was successfully logged out without any issues. Further investigation revealed that failed logout attempts resulted in an "invalid CSRF-token" error, although no corresponding feedback was displayed in the user interface (see Section 4.3.6). The delete account and note saving functionalities exhibited similar behavior.

Testing also revealed that the key label field on the settings page was not functional, likely due to incomplete implementation. As a result, this field was omitted from later tests.

Some of these faulty features did not provide any feedback to the user. In particular, when logout failed, the user did not receive any indication of whether the action was successful or not.

4.1.1 Error handling

When the error handling of Project one was investigated, an instance of implicit information disclosure was discovered. The information leak was caused by two closely related error messages. One of the error messages was given during login if a non-existing username and an invalid TOTP were provided, and stated *Invalid credentials* (figure 4.3). The other message was given during login if the username existed but the TOTP was invalid, and instead stated *Invalid TOTP code* (figure 4.4). Together, they implicitly reveal if there exists an entry with that username within the database of the application, which could potentially be used by an attacker to gather information about usernames within the database. This could have been avoided by making both error messages return the same response message.

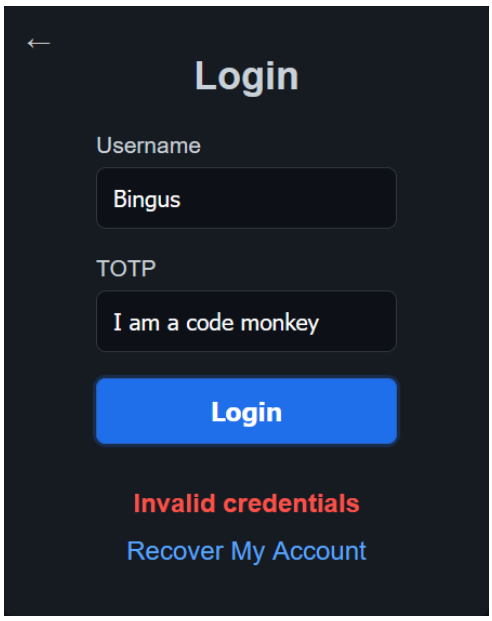


Figure 4.3: User *Bingus* does not exist in the database.

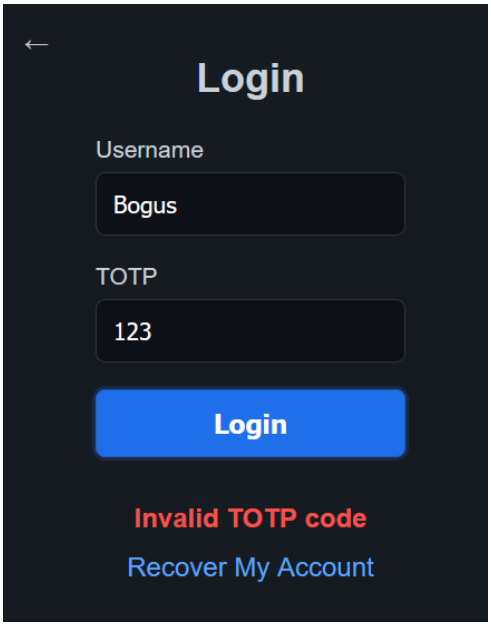


Figure 4.4: User *Bogus* exists in the database but the TOTP is invalid.

At this step it became clear that Project one left the application running in debug mode, as evidenced by the terminal printout shown in figure 4.5. This meant that the framework's debugger interface was exposed to the user if an error occurred. The frontend JavaScript attempted to mask this behavior by utilizing a try-catch block to suppress the debugger output. Whenever a 500 Internal Server Error was detected (figure 4.6), the client-side code intercepted the exception and instead displayed a generic error message (figure 4.7). A bypass of this was possible by slightly modifying the client-side JavaScript via built-in web browser developer tools, which forced the application to render the full debugger interface upon receiving a 500 error.

```
flask-app  |  * Debugger is active!
flask-app  |  * Debugger PIN: 244-497-982
```

Figure 4.5: Terminal printout on startup, showing that the debugger is active.

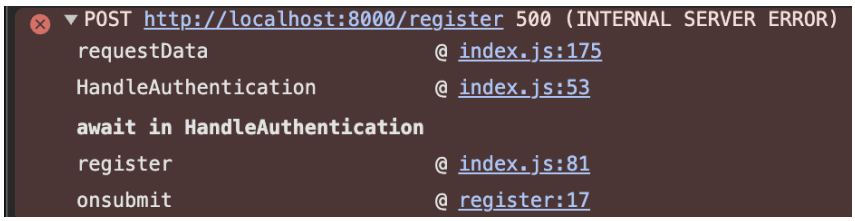


Figure 4.6: 500 (INTERNAL SERVER ERROR) in console

It was possible to trigger such an error by physically disconnecting the TKey from the client computer while a signing operation was in progress. The resulting crash generated a detailed stack trace that was subsequently revealed to the user. The crash log did not

```javascript
async function requestData(message, url, contentType, csrf) {
    try {
        const headers = {
            "Content-Type": contentType,
            ... (csrf && {'X-CSRFToken': csrf})
        };
        const response = await fetch(url, {
            method: "POST",
            headers: headers,
            body: contentType == ContentType.JSON ? JSON.stringify(message) : message
        });
        const data = await response.json();
        return { ok: response.ok, data: data };
    } catch (e) {
        return { ok: false, data: { error: "Communication error" } };
    }
}
```

Figure 4.7: Javascript code to mask errors [31].

expose any directly sensitive data, but could have been valuable to an attacker if the source code was not publicly available.

The debugger interface also featured a console that could be accessed by inputting a debugger PIN, granting an attacker access to an interactive Python terminal directly on the server. An overview of this debugger window is provided in figure 4.8. The developers of the debugger explicitly warn against this misconfiguration, stressing that the debugger allows for the execution of arbitrary code and must never be exposed in production environments [32].

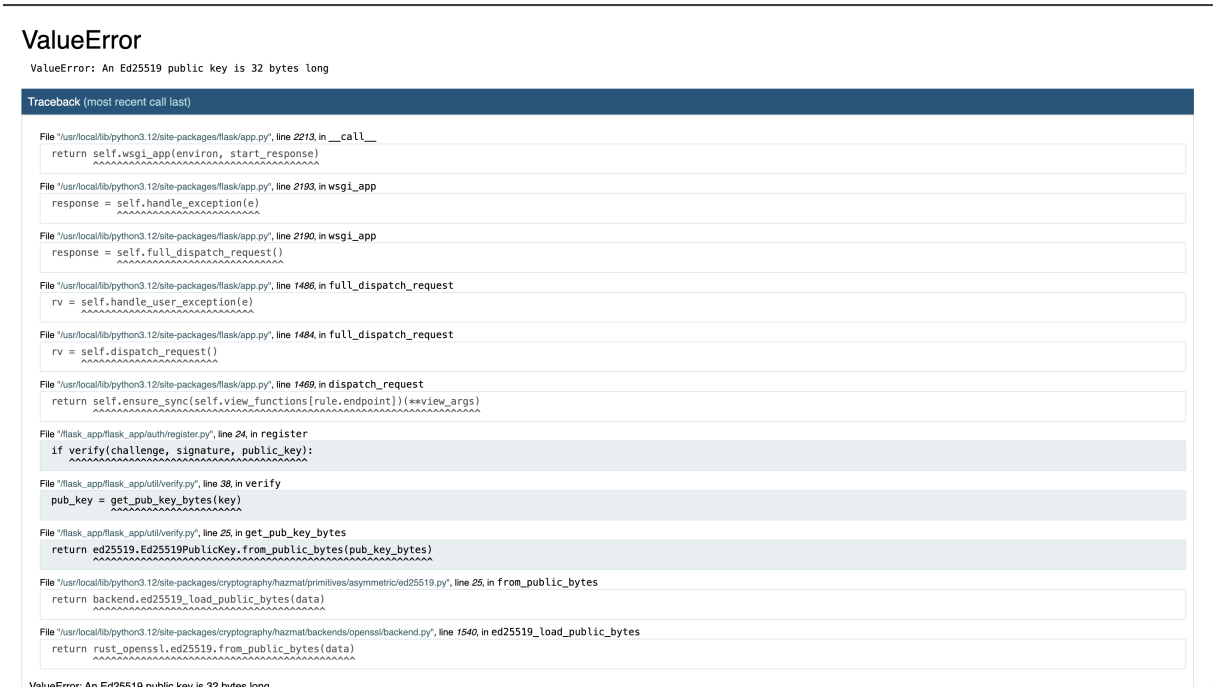


Figure 4.8: Exposure of the interactive debugger following a 500 Internal Server Error.

In Project two, when attempting to register a new account with the same TKey, if a user already exists the application will display “User already exists”.

During login, entering an existing username without the TKey being connected results in the application displaying “no TKey connected". However, entering a non-existent username results in the application displaying “User not found”. Since the application responds with different messages, an attacker could test usernames to identify all users.

4.1.2 Vulnerable communication

The traffic analysis of communication revealed that the applications in both projects relied on HTTP (Hypertext Transfer Protocol) for communication. HTTP turned out to be the cause of multiple critical security weaknesses, since HTTP does not provide any form of encryption. Therefore, any attacker with access to the network could intercept, read, and modify communication using MITM techniques.

Traffic analysis also revealed that multiple functions sent requests that transmitted highly sensitive values in plaintext. In Project one, there were authentication requests that contained usernames, TOTP codes, and signatures in plaintext, while registration requests contained usernames, signatures, and public keys in the same way. Even the two-factor registration process exposed the TOTP value directly, and the final registration step revealed all the mnemonic recovery words within the transmitted data.

In Project two, there were authentication requests that contained usernames and signatures in plaintext, registration requests that contained usernames and key labels in plaintext, and all requests also contained session cookies and CSRF-tokens in plaintext. An attacker capable of intercepting network traffic could therefore directly obtain critical authentication-related information without the need to compromise the system itself.

4.1.3 Attacking through the browser

When testing for access to sensitive information, with the Mozilla Firefox browser, there was nothing noteworthy found for either projects. All attempts of trying to change the url to gain access to authentication protected pages turned out to be unsuccessful for both projects, and the applications therefore seemed to have implemented proper protection against such tampering.

4.2 Scanning

A scan of the host computer running the applications was performed using the tool *Nmap* with the goal of revealing as much information as possible that could be used for later tests. The results showed that the applications were running multiple services on different ports that needed further investigation (figure 4.9).

```
C:\Users\User>nmap -sC -p- "127.0.0.1"
Starting Nmap 7.98 ( https://nmap.org ) at 2026-03-06 10:53 +0100
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00011s latency).
Not shown: 65513 closed tcp ports (reset)
PORT      STATE    SERVICE
135/tcp   open     msrpc
137/tcp   filtered netbios-ns
445/tcp   open     microsoft-ds
1337/tcp  open     waste
5040/tcp  open     unknown
5426/tcp  open     devbasic
5432/tcp  open     postgresql
6379/tcp  open     redis
6463/tcp  open     unknown
7680/tcp  open     pando-pub
8000/tcp  open     http-alt
|_http-title: T-95
8081/tcp  open     blackice-icecap
```

Figure 4.9: The returned results of a simple scan using Nmap on a host running Project one.

In the case of Project one, the Nmap scans showed (see figure 4.9) that the database ran on port 5432, the application itself used port 8000 and used HTTP, and that the web-server ran on port 6379 and used Redis.

In the case of Project two, the scans showed that the server ran on port 8080 with HTTP, Node.js on port 3000 with HTTP, and that the database ran on port 27017 and used MongoDB.

4.3 Vulnerability Assessment

The vulnerability assessment step started with the investigation of the information gathered from the Nmap scans made during the scanning step.

4.3.1 Data Exposure

The web-server in Project one used a Redis service on port 6379. Further investigation showed that the Redis service was tasked with storing Flask sessions, challenges, recovery states, and rate-limiting data. Therefore, the web-server was a high value target in the system, since the data stored within it was directly tied to authentication, access control, and other protective mechanisms of the application.

The Redis service turned out to be accessible from the network without any effective authentication, allowing attackers to interact with the backend services directly. This was verified by interacting directly with the Redis instance through the container environment (figure 4.10).

```
tiam@Tiams-MacBook-Pro ~ % docker exec -it web-application-redis-1 redis-cli ping
PONG
```

Figure 4.10: By executing commands such as *docker exec -it web-application-redis-1 redis-cli ping*, the service responded with *PONG*, confirming that it was reachable and accepted connections.

Further testing demonstrated that an attacker was able to modify and delete stored data within the Redis service, such as resetting rate-limiting counters (figure 4.11). This vulnerability was caused by a backend component that completely lacked proper access control. The service was assumed to only interact with trusted components, but no enforcement of such assumptions was done at all. Such assumptions were never enforced in practice for requests that avoided the user interface, which caused the vulnerability.

```
tiam@Tiams-MacBook-Pro ~ % docker exec -it web-application-redis-1 redis-cli keys "*"
1) "session:1b05a334-dcd1-4903-a651-9f48c6d2e9c2"
2) "LIMITS:LIMITER/192.168.65.1:None/auth.login/5/1/minute"
tiam@Tiams-MacBook-Pro ~ % docker exec -it web-application-redis-1 redis-cli flushall
OK
```

Figure 4.11: The command *docker exec -it web-application-redis-1 redis-cli flushall* was executed to test if it was possible to modify stored data within the Redis service. The service responded with *OK*, confirming that it was both reachable and modifiable.

4.3.2 Proxy Exposed to Local Area Network

In Project one, the proxy had a proper CORS implementation, meaning that only the intended website could legitimately send a challenge to be signed by the TKey. This CORS implementation was enforced entirely at the web browser level, where the challenge was processed regardless, and the browser determined whether to discard the data due to a CORS violation or not.

The CORS implementation could therefore be bypassed by avoiding the browser and instead sending requests directly using a terminal script or custom client. If a command was sent from another computer directly to the host's IP address and the proxy's port, a challenge could be submitted for signing. The proxy signed the challenge and returned the message along with the HTTP CORS violation headers. Since a terminal script is not a browser, the attacker could simply ignore the headers and read the signed challenge directly from the response body.

This turned out to be caused by two distinct misconfigurations in the proxy codebase. First, rather than restricting traffic to the local loopback address, the server was insecurely configured to listen to all available network interfaces by binding to `":8081"` (see Listing 4.1). Second, the application logic proceeded to sign the challenge even if the request's origin was not listed in the allowed CORS origins.

Listing 4.1: Insecure network binding to all interfaces [33]

```
fmt.Println("Server running on port 8081")
log.Fatal(http.ListenAndServe(":8081", nil))
```

4.3.3 SQL-injection attacks

The application code of Project one showed signs of having implemented parameterization in an attempt to prevent SQL-injection. All attempts to perform SQL-injection failed to produce any results, indicating that the applications were not vulnerable to SQL injection.

The protection was caused by proper implementation of parameterized queries when interacting with the database. Instead of directly inserting user input into SQL statements, inputs were passed separately using placeholders such as %s, ensuring that they were treated strictly as data rather than executable code (figure 4.12). This effectively prevented attackers from altering the structure of SQL queries.

```
cursor.execute(
    'SELECT username FROM "user" WHERE id = %s',
)
```

Figure 4.12: Code that sends a properly parameterized query to the database in Project one by using the cursor.execute function. The structure of the query shows that it is using proper parameterization [34].

The Nmap scans later revealed that Project one used a PostgreSQL-language database running on port 5432 for windows or 5434 for mac-OS. Attempts to send HTTP requests to the database using curl proved that the database responded to requests without doing any authentication checks. Therefore, the database suffered from a vulnerability of the Broken-Authentication-Logic type, as there were no effective access restrictions in place. This meant that anyone could interact with the database directly without going through the application layer.

Project two had the same issues but instead used a NoSQL-language database, MongoDB. NoSQL-injection tests were performed using Burp Suite and mongosh. The database did not require any authentication during communication, just as in Project one. Therefore, in both projects, it was possible for malicious users to access the database to extract, manipulate, or delete information, despite that the databases used different languages and that their code had proper parameterization.

The application in Project two only allowed alphanumerical characters in its input fields when registering users, which successfully removed the possibility to input NoSQL-injection prompts, since such prompts required other characters. Therefore, the application in Project two was protected against NoSQL-injection. However, later tests found that it was possible to use other characters in the note-function of the application. This enabled the creation of a note with the name *1'||'1'=='1* that the application accepted (figure 4.13). Despite this being possible, the note was still treated as a string and did not enable any NoSQL-injection. The application responded later with a terminal entry that the application recognized the input not being sanitized and therefore rejected it (figure 4.14). This further underlined that Project two was successfully preventing attempts to perform NoSQL-injection.

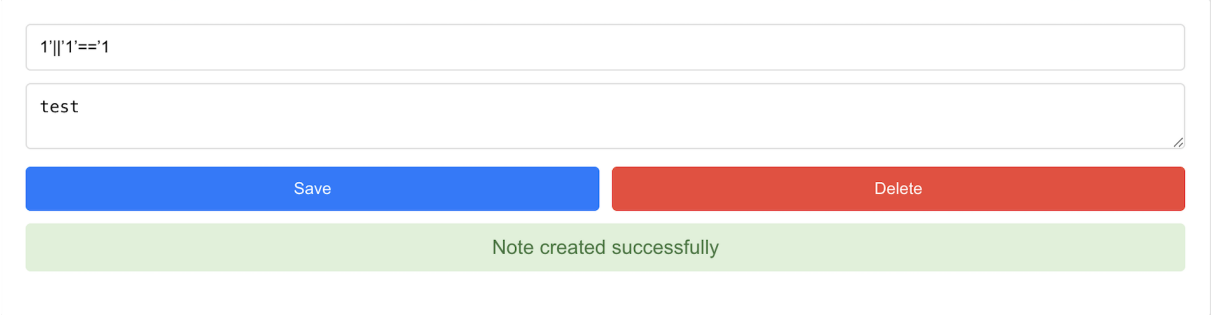


Figure 4.13: Note with NoSQL-injection input created

```
Received registration request for user: 1'||'1'=='1
Input is not sanitized
```

Figure 4.14: Database terminal showing application acknowledging the input is not sanitized

4.3.4 Challenge mechanism

The challenge-response mechanism in Project one exhibited design weaknesses that could lead to potential security issues. The generated challenge was stored in the session and was neither invalidated nor regenerated when authentication attempts failed, such as when a signature or TOTP verification was unsuccessful. Additionally, although the challenge included a timestamp, there was no validation of its freshness or any enforcement of an expiration window. This resulted in challenges remaining valid indefinitely.

This behavior was observed in multiple parts of the system, including the login, registration, and recovery flows, indicating a weakness in the authentication design. Furthermore, the generated challenge was never bound to any security context, meaning that it had no association with specific information such as a user, domain, operation type, or service. Consequently, a valid signature only proved that the TKey signed a given string, without guaranteeing what that string represented or in which context it was to be used.

At a lower level, the proxy server introduced an additional weakness by performing blind signing of incoming data. It was observed to process an HTTP request body and forward it for signing without validating its structure or ensuring that it matched an expected challenge format. This increased the risk of attacks such as spoofed challenges or signing confusion, as no guarantee was provided that the signed data was a legitimate challenge generated by the system.

These issues were caused by flaws in the design and implementation of the authentication mechanism, particularly within the challenge-response workflow. The issues could enable attacks such as replay attacks, misuse of signatures, and possibly even authentication bypass. For instance, since challenges do not expire, an attacker may reuse previously issued challenges or repeatedly attempt authentication against the same challenge. Additionally, due to the lack of context binding, Signatures could potentially be reused across different services if identical or compatible challenge values are accepted.

4.3.5 Docker configuration issues

In Project one, several Docker-related issues were identified. By observing the internal files using the Docker desktop application, files with overly permissive permissions were found.

The file *init.sql* was found with the permissions *-rwxrwxrwx* (or 777) set, allowing anyone to modify it (figure 4.15). The file was contained inside the *docker-entrypoint-initdb.d* directory, meaning that init.sql would be run on initialization of the database with database privileges. Furthermore, the init.sql file showed the *mount* symbol, meaning that the file was from the host system, and not the Docker container environment. This not only indicated that the init.sql was host-controlled and persistent between container lifecycles, but also that the container was not self contained. Consequently, init.sql might allow attackers to inject or modify startup SQL code, and thereby initialize the database with attacker-chosen logic.

Figure 4.15: Init.sql allows anyone to modify it, potentially allowing anyone to modify database initialization behavior.

Another similar misconfiguration was found. The directory *postgresql* found at */var/lib/-postgresql* was used for database storage and had its permissions set to *dtrwxrwxrwx* (figure 4.16). These settings allowed anyone to read and add files within that directory. Furthermore, the directory had the *volume* symbol, meaning that the directory data was stored outside of the container and could therefore persist on the host or backend.

Figure 4.16: The directory "postgresql" allowed anyone to read and add files, while the sticky bit restricted modification and deletion of files to the owner of a file and root.

Tests also proved that the database and Redis were directly published to the host, even though they were intended to be only accessible from within the internal container network. This was confirmed by performing direct interaction tests with the exposed services. The PostgreSQL database was found to be accessible on port 5432 or 5434 (Windows or mac-OS), which was verified using network scanning tools (e.g., nc -zv localhost 5434). A successful connection confirmed that the database port was exposed. Further interaction using psql demonstrated that it was possible to connect directly to the database from outside the application by using known credentials, and query sensitive data such as user information.

Similarly, Redis was found to be exposed on port 6379 as defined in the Docker configuration. This exposure was verified by connecting to the Redis instance from a separate device on the same local network, targeting the host machine's local IP address using redis-cli (e.g., redis-cli -h 192.168.1.75 -p 6379). Upon connection, it was possible to access stored data such as session information and rate-limiting counters without any form of authentication. These findings confirmed that both Redis and the database were not properly restricted to the internal Docker container network, but were instead exposed

directly to the wider local network through the host's network interface.

The root causes of these issues were improper design and failure to follow the principle of least privilege. Instead of restricting internal services to trusted communication channels within the container network, they were unnecessarily exposed to the host environment, which resulted in a situation where internal components such as Redis and PostgreSQL became directly accessible to attackers.

4.3.6 Cross-Site Request Forgery

In Project one, multiple weaknesses related to CSRF were identified during the traffic analysis of its communication. The most notable issue was that the logout endpoint was implemented using the HTTP GET method, despite logout being a state-changing operation (figure 4.17). This vulnerability enables an attacker to embed a link or resource, such as an image or iframe pointing to /logout, which will cause the victim's browser to automatically send the request and log the user out without their consent. This is commonly referred to as a Logout CSRF.

Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	HTTP	617	GET /logout HTTP/1.1
127.0.0.1	127.0.0.1	HTTP	597	HTTP/1.1 302 FOUND (text/html)
127.0.0.1	127.0.0.1	HTTP	557	GET / HTTP/1.1
127.0.0.1	127.0.0.1	HTTP	698	HTTP/1.1 200 OK (text/html)

Figure 4.17: Wireshark results showing the use of HTTP in communication. Note the topmost logout request being a HTTP GET-request.

Furthermore, although attempts were made to implement a CSRF protection mechanism using CSRF tokens, the mechanism was undermined by the use of HTTP. Due to the communication being unencrypted, an attacker could steal the CSRF tokens of a user by performing a MITM-attack. The attacker could then reuse the tokens to perform authenticated actions as that user. Consequently, even endpoints that utilize CSRF tokens remained vulnerable.

Project two also relied on CSRF tokens for protection. Testing was performed manually using Burp Suite by sending requests with missing or invalid tokens to each endpoint. A request without a token returned: *forbidden - CSRF token not found in request*, while a request with an invalid token returned: *forbidden - CSRF token invalid*.

All endpoints in Project two except `/api/register`, `/api/login` and `/api/verify` are protected against CSRF. The `/api/register` endpoint requires no protection as it has no meaningful attack surface for this type of attack. The `/api/login` and `/api/verify` endpoints instead rely on cookies for authentication and are therefore not protected using CSRF tokens in this implementation. However, as with Project one, the use of unencrypted HTTP means that CSRF tokens could be intercepted by an attacker.

In addition to these structural observations, tests revealed implementation-specific flaws that caused unintended behavior in Project two. As described in the reconnaissance phase

(see Section 4.1), features such as logout, note saving, and account deletion failed intermittently, producing "invalid CSRF-token" errors without any user feedback.

A code analysis identified the cause of this behavior. The application implemented a client-side CSRF token cache in `csrf.js`, where a token was fetched once and stored in memory. Although a `clearCsrfToken()` function was available to invalidate this cache, it is never imported or invoked anywhere in the application, including in the component responsible for handling logout (see figure 4.18).

```
export const clearCsrfToken = () : void  => {
  csrfTokenCache = null;
};
```

Figure 4.18: The `clearCsrfToken()` function is defined but never invoked anywhere in the application, resulting in stale CSRF tokens being reused across requests [35].

When a user logs out, the server invalidates the session and its associated CSRF token. However, since the client-side cache is never cleared, the application continues to use the stale token for subsequent requests, causing the server to reject them with an "invalid CSRF-token" error. Additionally, these errors are only logged to the browser console via `console.error()` and no feedback is presented in the user interface, leaving the user without any indication of whether the requested action was successfully completed or not.

4.4 Exploitation

The exploitation step had the goal of investigating if the information gathered during the reconnaissance, scanning, and vulnerability assessment steps could be utilized to develop some sort of exploit. The implications of such vulnerabilities were also considered.

4.4.1 Vulnerable Database

The databases had no authentication requirements, which resulted in that all requests sent to them were accepted. This could be used as an exploit on its own, as any request will be carried out.

4.4.2 Manipulate Communication Data

The usage of HTTP instead of HTTPS (HyperText Transfer Protocol Secure) made the projects vulnerable to several different types of exploits. Many of these belonged to the MITM type of attacks, and therefore compromised the integrity and authenticity of the communication.

This enabled attackers to perform packet sniffing to capture sensitive information such as session cookies, authentication data, and CSRF tokens. With access to a valid session cookie, the attacker could impersonate a legitimate user and gain unauthorized access to their account, resulting in session hijacking. Additionally, since the communication

was not protected, an attacker could manipulate the communication in transit, enabling spoofing attacks that could alter the resulting behavior of the communication. Messages could also be intercepted and be resent in order to make replay attacks.

4.4.3 Exposed database storing data in cleartext

As mentioned above, the databases turned out to store sensitive data in clear text and were also found to have no authentication checks (figure 4.19). These two misconfigurations could be used together to create very powerful exploits. Together, they allow an attacker to gain full access to the database and also to read, modify, or delete stored data, resulting in a complete compromise of confidentiality, integrity, and availability.

```
db=# \d "user"
                                Table "public.user"
  Column   |  Type   | Collation | Nullable |             Default
-----------+---------+-----------+----------+-----------------------------------
 id        | integer |           | not null | nextval('user_id_seq'::regclass)
 username  | text    |           | not null |
 publickey | text    |           | not null |
 secret    | text    |           | not null |
 salt      | text    |           | not null |
 hash      | text    |           | not null |
Indexes:
    "user_pkey" PRIMARY KEY, btree (id)
    "user_publickey_key" UNIQUE CONSTRAINT, btree (publickey)
    "user_username_key" UNIQUE CONSTRAINT, btree (username)

db=# SELECT id, username, publickey, secret, salt, hash FROM "user";
 id | username |                  publickey                  |              secret              |               s
alt              |                              hash
----+----------+---------------------------------------------+----------------------------------+----------------
-----------------+----------------------------------------------------------------
  1 | tiam1234 | luAREbfOA1bzZUxrnHymYq4gG1fwpCqOROWk1biuDso= | R5BTAWTZBZPX2FK4XHIMJLEHRUWAU74N | 210ed1f548cdd9a
91e71941c8366937a | 4bb11a27244185799103bac8e65f4d57585f035f3c82836431c2a7834c248b15
(1 row)
```

Figure 4.19: Project one, extract from the PostgreSQL `"user"` table showing directly accessible authentication-related fields, including a plaintext secret.

This could also be used to obtain the TOTP secret stored within the database, allowing an attacker to generate valid TOTP codes and thereby fully bypass the two-factor authentication mechanism. As a result, user accounts could be taken over, and thereby make the whole authentication system of the application ineffective. The attacker could also use this exploit to delete all database content, including tables, as a Denial-of-Service attack.

4.4.4 Cross Site Scripting

In Project one, attempts were made to identify potential XSS vulnerabilities by crafting user input that, if improperly handled, would be interpreted as executable code in the application's output, particularly in scenarios where data was stored in the database and later rendered (stored XSS). However, none of these tests were successful, and no XSS vulnerabilities were found.

The reason for this resistance was that user input was properly handled and treated strictly as data and was also safely processed when rendered, thereby preventing it from being interpreted as executable code. This was discovered during the testing for SQL-injections. This indicated that appropriate protections against XSS were implemented in

the application.

Project two was also tested for potential XSS vulnerabilities. The project was built using React, a JavaScript library for building user interfaces. React automatically escapes output before rendering content in the Document Object Model (DOM), which helped prevent XSS by default. As a result, the injected payloads were displayed as plain text rather than executed as JavaScript.

Most input fields were protected by front-end validation that restricted input to letters and numbers using the regular expression `[^a-zA-Z0-9]`. This was implemented for both the username field on the registration page and the username field on the login page. This restriction prevented basic XSS payloads from being entered through the user interface. However, it was possible to bypass it by intercepting and modifying HTTP requests, revealing that the implemented client-side validation alone did not provide sufficient protection against XSS attacks.

The notes functionality was of particular interest for testing, as it accepted arbitrary user input that was stored in the database. A payload was inserted into both the name and note fields and submitted using Burp Suite. Although the request was transmitted in plaintext, the backend returned the data with Unicode encoding, converting characters such as `<` and `>` into `\u003c` and `\u003e`.

The notes field was implemented as a `<textarea>` element, which inherently renders content as text rather than HTML. This provided an additional layer of protection against script execution. Overall, the notes functionality was protected by multiple layers, including backend encoding, React's default output escaping, and safe HTML input handling.

A search of the application source code for `dangerouslySetInnerHTML`, a React function commonly used to bypass built-in XSS protection, was also conducted. The search returned no results outside of `node_modules`, confirming that there was no intentional bypass of the default escaping mechanisms that React uses anywhere in the application.

Reflected XSS was also tested by attempting to inject malicious payloads through both query parameters and URL paths. These attempts resulted in the application not matching any valid route, leading to a blank- or error page, but no JavaScript execution occurred. Therefore, neither stored XSS nor reflected XSS were viable attack vectors in Project two.

As a result, no XSS vulnerabilities were found in Project two. The backend encoded user data used Unicode escaping, React automatically escaped all output before rendering, and `dangerouslySetInnerHTML` was not used anywhere in the source code.

4.4.5 Brute Force attack

Brute force attacks were effectively limited in both projects. The main reason was that the authentication process relied on the TKey, where each authentication attempt required the generation of a signature using the TKey. This process involved physical user interaction (touch), which significantly reduced the number of requests that could be per-

formed within a given time frame. As a result, even if an attacker would attempt to brute force TOTP codes, the limited validity period of TOTP values (typically around 30 seconds), combined with the slow rate of signature generation, makes it practically impossible to try enough combinations. The only way this type of attack could become a realistic attack vector would be if there was some way to circumvent the need for touch interaction.

From a theoretical perspective, if rate limiting mechanisms were removed, for example through access to Redis, and if the attacker had access to a TKey or a substitute, brute forcing TOTP could become possible. However, even in such a scenario, the inherent limitations of the TKey, including required user interaction and signing latency, would still hinder the attack. Consequently, this attack vector only becomes viable if an emulator is used to bypass these hardware constraints, such as eliminating the need for physical touch interaction.

Therefore, the system can be considered resistant to brute force attacks, where the resistance primarily stems from the physical and hardware-enforced constraints of the TKey rather than from software-based protections.

4.4.6 Denial of Service

In Project one, at the communication layer between the host and the TKey, the use of the `io.ReadFull()` function caused the system to block until the full payload had been received. This function was designed to continue reading until the expected number of bytes had been obtained and, in doing so, might override previously configured timeouts. As a result, if the TKey failed to return a complete response or if the communication was disrupted, the reading process could remain blocked indefinitely. This could in turn potentially lead to resource exhaustion or stalled processing. This behavior could be exploited in scenarios that involve unstable communication, faulty device behavior, or in cases where an attacker can interfere with the communication channel.

In another part of the system, within the signing process, a maximum input size (MaxSignSize = 4096) was defined, but this was never enforced on the host side before sending data to the device. This allowed arbitrarily large inputs to be transmitted in chunks to the TKey, which could result in a high number of chunked requests being processed by the device for large inputs, which could significantly increase computational load and degrade system performance. Although the Go runtime includes bounds checking that prevent crashes, and the signer application on the TKey appears to enforce its own size limits, this behavior could still lead to performance degradation and resource exhaustion rather than a complete failure.

Overall, these issues were best categorized as weaknesses in resource management and input validation rather than critical vulnerabilities. However, under certain conditions, they may impact system availability and lead to Denial-of-Service scenarios.

4.4.7 Buffer overflow

In Project one, the analysis indicated that appropriate safeguards were in place to prevent buffer overflow attacks. Specifically, within the signer application running on the TKey, a strict limit is defined for the size of input messages (`Max_Sign_Size` = 4096 bytes) (figure 4.20). If a message exceeded that limit, the request was rejected entirely and no further processing was performed. This behavior was enforced to ensure that oversized input were not accepted (figure 4.21).

```
// Touch timeout in seconds
#define TOUCH_TIMEOUT 30
#define MAX_SIGN_SIZE 4096
```

Figure 4.20: The signer application defines `MAX_SIGN_SIZE` as 4096 bytes, limiting the maximum size of messages accepted for signing [36].

```
if (local_message_size == 0 ||
    local_message_size > MAX_SIGN_SIZE) {
    debug_puts("Message size not within range!\n");
    rsp[0] = STATUS_BAD;
    appreply(pkt.hdr, RSP_SET_SIZE, rsp);

    state = STATE_FAILED;
    break;
}
```

Figure 4.21: The signer application checks the message size before processing [37]. If the message is empty or exceeds `MAX_SIGN_SIZE`, the request is rejected, `STATUS_BAD` is returned, and the state is set to `STATE_FAILED`.

Furthermore, the handling of input data ensured that only the portion of data within the allowed size was processed, and no out-of-bounds memory operations were performed. This prevented the possibility of memory overwrites or buffer overflow conditions. As a result, even if large inputs were sent to the system, they were safely handled within the defined constraints.

4.4.8 Proxy Exposed to Local Area Network

In Project one, the proxy had a proper CORS implementation, meaning that only the intended website could legitimately send a challenge to be signed by the TKey. To exploit this vulnerability, only the victim's username and access to the same local area network (LAN), such as a shared workplace network, were required. During testing, the login procedure was initiated on the web application using the victim's username in order to retrieve the authentication challenge. The challenge was then sent directly to the victim's computer over the LAN, where the daemon signed it automatically. Finally, the signed challenge was forwarded back to the web application, which resulted in successful authentication as the victim user.

This approach was initially blocked by the target host's internal firewall, which blocked the application from external network requests. The first time the target host computer received such a request from an external source, it prompted the user to allow the connection, often with the "allow" button pre-selected. Thus, it was possible to bypass this firewall if the user unknowingly accepted the prompt. The second roadblock was that the TKey required physical touch to actually sign the challenge, and if it was not touched within 30 seconds, the signing request timed out. This could also be circumvented, a confused user might simply touch the device as soon as it blinked green, or the attacker could physically approach the desk and touch it. Alternatively, by sending the request repeatedly, the device would continuously blink green, potentially prompting the user to eventually touch it to stop the flashing. Additionally, a potential design flaw in the TKey was identified, if an unauthorized signing was initiated and the user attempted to disconnect the TKey from the computer to stop it, the device might register the physical handling as a valid touch and sign the challenge just before being removed. This resulted in an involuntary signing that was not intended by the user.

4.5 TKey

The TKey itself was also investigated in this security assessment. The investigation of the TKey focused on analyzing the open source code and behavior of the device, while the algorithms used to create key pairs and sign keys were considered out of scope and therefore safe in this analysis.

4.5.1 Wrong bitmask

An issue was found in the protocol that handles the USS and frames it to send with the TKey's transmission protocol (table 4.1).

Table 4.1: Structure of the `FW_CMD_LOAD_APP (0x03)` frame

Name	Size (bytes)	Comment
size	4	Integer (LE)
USS-provided	1	0 = false, 1 = true
USS	32	Ignored if above is 0

In that protocol, there is one byte that is used to tell the TKey if there is a USS supplied by the user, and the following 32 bytes are used for the USS itself. The tests showed that if a USS was supplied, the USS byte was set to one as intended, but the USS was sent one byte early in the framing protocol, overwriting the USS bit (Listing 4.2).

Listing 4.2: Wrong bitmask in protocol implementation [38]

```
if len(secretPhrase) == 0 {
    tx[6] = 0
} else {
    tx[6] = 1
```

```
        // Hash user's phrase as USS
        USS := blake2s.Sum256(secretPhrase)
        copy(tx[6:], USS[:])
    }
```

This resulted in the USS being ignored, despite it being successfully sent by the framing protocol to the TKey, as the byte telling the TKey that the USS existed was overwritten (Listing 4.3).

Listing 4.3: TKey firmware logic for USS inclusion [39]

```
// Possibly hash in the USS as well
if (use_USS != 0) {
    blake2s_update(&secure_ctx, USS, 32);
}
```

This in turn resulted in multiple USS values producing the same signature, where the affected USS values shared the condition that the first byte of the hash of the USS only contained zeros. The resulting signature given by an affected USS was identical to the one generated when no USS was provided at all. Additionally, the user remains completely unaware that the USS was never used, as there is no error message or visible unexpected behavior caused by this issue.

An example of this is the USS "football", as shown in listing 4.4. The first byte of this hash consists of all zeros (Listing 4.3), which results in the USS never being used, therefore producing the same signature as when no USS is provided.

Listing 4.4: Go code to get hash

```
        hash := blake2s.Sum256([]byte("football"))
        fmt.Println(hash)

$ go run main.go
[0 34 203 111 50 220 88 188 6 205 207 206 122 169 67 93
143 194 101 43 101 238 38 157 26 176 186 25 146 244 65 69]
```

Further investigation revealed that this was not the only problem caused by this bug. Another result was that it was possible to tell the TKey that no USS was sent, which it accepts as true without question. This meant that the USS was stored in Random Access Memory (RAM) on the TKey but was not part of the measured boot. Furthermore, it was also found that the frame for sending the 32 byte USS was actually 128 bytes. This could be exploited by an attacker bypassing the standard USS input interface (figure 4.22), and thus skip the step where the application normally hashes the input to 32 bytes. Instead, by communicating directly with the framing protocol, the attacker could transmit a large, arbitrary payload disguised as a USS to the TKey. By setting the USS-provided byte to zero, the device would ignore the payload while continuing to process the rest of the frame.

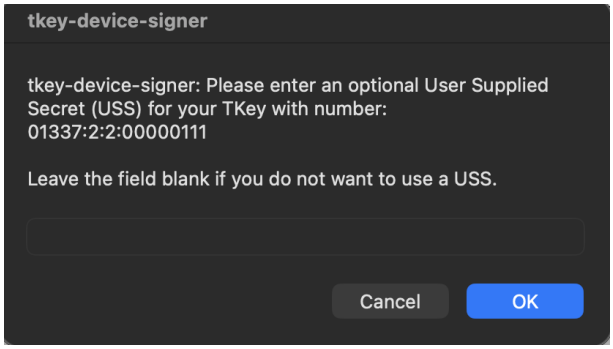


Figure 4.22: USS popup window

4.5.2 Keep it powered

Project two made it optional to provide a USS when registering a TKey to an account (figure 4.22). Testing discovered that once the USS was provided and the TKey had performed its calculations, it was possible to log in to the account without providing the USS again, on the condition that the TKey remained plugged in. The TKey retained the USS in memory because it was loaded with the application, and the USS formed part of the Compound-Device-Identifier (CDI) from which the application derived its keys.

This vulnerability was successfully exploited in two ways by ensuring that the TKey received continuous power which maintained the USS within its volatile RAM. The first scenario involved a shared computer, such as one found in a computer lab or library, where multiple users could log in to different operating system accounts. If the owner of a TKey signed out of both the application and the local account but left their TKey plugged into the shared machine unattended, an attacker could exploit this. The attacker could log in to their own local user account on the shared computer, navigate to the web application, and enter the first user's username. Because the TKey remained plugged in, and therefore powered, the USS of the previous user was still stored on the device, allowing the attacker to successfully sign in to the victim's account without even knowing their USS.

The second scenario involved a modern office environment with unassigned seating and laptop users. Desks in these environments often utilize a "one-cable solution", such as a powered USB docking station or hub, connecting multiple peripherals to the laptop through a single cable. If a user plugged their TKey into the docking station rather than directly into their computer, they might forget it when leaving the desk. Because the docking station was powered by its own external supply rather than the laptop, the TKey never lost power. An attacker could simply disconnect it from the victim's laptop and connect their own laptop to the docking station. The attacker could then navigate to the web application and enter the victim's username. Because the TKey had maintained power, it retained the already generated keys in its memory, allowing the attacker to log in without needing to provide the victim's USS.

4.6 Post exploitation

The vulnerabilities found in the projects and for the TKey were evaluated and the resulting calculation can be seen in table 4.2, 4.3, table 4.4 and table 4.5.

Based on the conducted evaluations, several of the investigated vulnerabilities were found to pose no security risk, and the projects were considered protected against them. The vulnerabilities that were investigated but ultimately classified as having no risk assessment were: SQL-injection, XSS, and buffer overflow.

Brute force attacks were also found to present no practical risk in the evaluated projects. However, unlike the vulnerabilities listed above, brute force attacks were still included in the risk evaluation tables since they remain theoretically possible. Due to the limitations and conditions of the evaluated systems, particularly the use of the TKey and its required physical interaction, such attacks could not be carried out in practice. Nevertheless, from a theoretical perspective, brute force attacks may still be possible under special conditions, and were therefore included in the evaluation.

The Wrong Bitmask vulnerability is not included in the evaluation because its impact only degrades the security of the TKey rather than fundamentally undermining it. Any risk-score calculation using our model would therefore misrepresent the results.

Vulnerability	Metric	Score	Description	Risk	Severity
Information disclosure	AC	3	Low complexity attack	6	High
	RA	3	No access required		
	C	2	Attacks can reveal usernames		
	I	0	No modification of data		
	A	0	No impact on system Availability		
MITM vulnerability	AC	3	Low complexity attack	9	Critical
	RA	3	No authentication required		
	C	3	All data included in the requests is exposed		
	I	3	The data can be fully modified		
	A	3	An attack can completely disrupt the system		
Exposed Redis	AC	3	Low complexity attack	7.5	Critical
	RA	2	Local network access required		
	C	3	Reveals all data in the Redis database		
	I	3	All data can be modified		
	A	3	An attack can cause system failure		
Exposed proxy server	AC	2	Medium complexity attack	6	High
	RA	2	Local network access required		
	C	0	No data would be revealed		
	I	3	Can be used to sign data		
	A	0	No impact on system Availability		
SQL-injection	-	-	No vulnerability was found	0	No risk
Docker misconfiguration	AC	1	An attack would need some way to access the Docker files	1.5	Low
	RA	0	Unknown. Unclear if attacker can access the files		
	C	3	Sensitive data such as source code would be exposed		
	I	3	Database could be be launched with attacker logic		
	A	3	An attack could cause complete system takeover		
CSRF logout vulnerability	AC	3	An attack is easy to perform	3	Medium
	RA	3	No access required		
	C	0	No data would be revealed		
	I	0	No modification of data		
	A	1	Victim would be temporarily logged out		
Exposed database	AC	3	Low complexity attack	7.5	Critical
	RA	2	Local network access required		
	C	3	Reveals all data in the database		
	I	3	All data can be modified		
	A	3	An attack can cause system failure		

Table 4.2: Risk evaluation of Project one – Part 1

Vulnerability	Metric	Score	Description	Risk	Severity
Brute force vulnerability	AC	0	An attack is practically impossible	0	No risk
	RA	0	Not possible		
	C	3	User accounts could be compromised		
	I	0	No modification of data		
	A	3	Can be used to flood the system with requests		
Communication layer vulnerability	AC	2	The attack would be difficult to execute and require knowledge of the TKey	4	Medium
	RA	2	The attack would require the TKey		
	C	0	No data would be revealed		
	I	0	No modification of data		
	A	2	The attack would have a limited impact on the system		
Buffer overflow	-	-	No vulnerability was found	0	No risk

Table 4.3: Risk evaluation of Project one – Part 2

Vulnerability	Metric	Score	Description	Risk	Severity
Information disclosure	AC	3	Low complexity attack	6	High
	RA	3	No access required		
	C	2	Usernames can be revealed		
	I	0	No modification of data		
	A	0	No effect on availability		
MITM vulnerability	AC	3	Attack is low complexity	9	Critical
	RA	3	No authentication required		
	C	3	All data included in the requests is exposed		
	I	3	The data can be fully modified		
	A	3	An attack can completely disrupt the system		
SQL-injection	-	-	No vulnerability was found	0	No risk
CSRF vulnerability	-	-	No vulnerability was found	0	No risk
Brute force vulnerability	AC	0	An attack is practically impossible	0	No risk
	RA	0	Not possible		
	C	3	User accounts could be compromised		
	I	0	No modification of data		
	A	3	Can be used to flood the system with requests		
Exposed database	AC	3	Low complexity attack	7.5	Critical
	RA	2	Local network access required		
	C	3	Reveals all data in the database		
	I	3	All data can be modified		
	A	3	An attack can cause system failure		

Table 4.4: Risk evaluation of Project two

Vulnerability	Metric	Score	Description	Risk	Severity
Keep it powered	AC	2	Low complexity attack. The device simply remains connected to power	4.5	Medium
	RA	1	Requires physical access or local control over the USB port power		
	C	3	Session keys or RAM contents might be preserved and extracted		
	I	0	No direct modification of device firmware data.		
	A	0	Prevents proper session termination or device zeroization		

Table 4.5: Risk evaluation of the TKey hardware and firmware components

5
Discussion

This chapter discusses and analyzes the results of the security assessment conducted in this project. The established research questions are addressed to draw final conclusions regarding the system's security. Potential areas for future work are presented as well as a section on societal and ethical aspects.

5.1 Key Findings

The security evaluation identified several vulnerabilities across both projects, and also some bugs in the TKey source code. However, the projects were still good proofs of concept for passwordless authentication. Considering that the developers developed the applications to show that it was possible to integrate the TKey into a login service, with functionality as the only focus, the projects could be transformed into secure applications.

The greatest security risks were caused by the use of HTTP instead of HTTPS. This made it possible to perform MITM attacks, steal QR codes used in TOTP setup, intercept network packets, and perform session hijacking. However, this HTTP implementation would unlikely have been used in the final production of such an application.

Other critical findings included a debugger that was still active, missing database authentication, a broken CSRF token mechanism preventing users from logging out, and databases open to global traffic (only relied on internal firewall protection). These findings were severe, but many were easily correctable and therefore possible to fix in future patches.

Regarding the TKey itself, the most serious security finding was a bug where the USS was not used in the measured boot process. Fixing this was simple, as it only required changing a number 6 to a 7 in the code (this fix was implemented in commit 4954dcc on March 17 2026 [40] (figure 5.1)). Another significant finding was that the device retained the computed CDI in its volatile memory after execution, rather than resetting to an initial state. While this was not an issue for the intended use of the TKey, where the TKey was unplugged after each use, it might be a security risk in practice. Users will likely have a tendency to leave it connected to the computer, keeping it powered with the computed CDI.

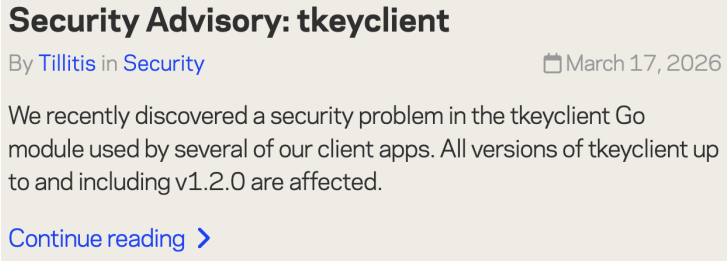


Figure 5.1: Tillitis security advisory published on March 17, 2026, detailing a vulnerability in the USS logic within the tkeyclient Go module.

In summary, although the identified vulnerabilities were serious, they primarily caused by implementation oversights rather than fundamental weaknesses in the TKey architecture. This indicated that the projects could become production-ready through implementation of standard security practices and remediation of the vulnerabilities that were identified during this assessment.

5.2 Problem Statement answered

The security assessment revealed that the applications did not implement enough protection to be in accordance with industry-standard security protocols. Although existing security successfully prevented some attack types, such as SQL-injection, other types were completely overlooked.

Throughout the security assessment a sort of pattern was noted: security implementations seemed to become less thorough when the attack vector was not interacting with the user interface or the browser. This pattern might have been caused by developers having an incomplete understanding of what an attacker is likely to see. The implemented security measures seemed to have been made with the idea that an attacker would see the same perspective as a normal user interacting with the user interface, but perform malicious actions rather than benign ones. The first thing an attacker actually sees might more likely be the result of a port scan, such as the Nmap scans made during the security assessment. This resulted in implementations with proper protection around their user interface, but with little to no protection when attacks were not targeting the user interface at all.

Neither of the tested applications was protected from MITM-attacks due to the use of HTTP instead of HTTPS. If these applications would have used normal password usage, a complete takeover of accounts would have been easy to perform, as all data required could have been acquired using MITM-attacks. Passwordless authentication therefore provided security benefits compared to traditional password usage, as the usage of a physical item rather than a digital passcode for authentication made MITM-attacks insufficient to perform account takeover. Given that the requirements were no longer only digital, it was no longer possible to steal them all using digital means. As a result, it was not possible to steal accounts, despite the entire communication being vulnerable to MITM-attacks.

The investigated projects were developed by students. It is therefore reasonable to assume that the awareness of security vulnerabilities and secure development practices was limited. The applications were also developed with functionality in mind, where security was implemented in the late stages of the project. In a real production environment it might be more likely that the developers have more extensive security knowledge and include security as a fundamental part of the development process rather than as an afterthought. However, this assumption does not always hold in practice. Even in the case of Tillitis and the TKey, minor security weaknesses were identified that could have escalated into more significant risks. Today, the barrier between creation and release of software is low. Therefore, popular applications were not necessarily developed by established companies, but could rather have been developed by sole individuals or small groups with varying levels of security competence. Consequently, it is fully possible that student developers release functional applications for commercial use that contain vulnerabilities comparable to those identified in this study.

There were also some concerns regarding design choices of the TKey itself. During the security assessment it became clear that when the TKey had processed information and remained plugged in, the information it calculated remained in its memory due to the TKey still being powered. This creates security risks if the computer is left unattended, even more so if it is unattended and connected to an external hub instead of the computer directly, as this makes it possible to connect and disconnect different computers without the TKey registering it. This could then be used to authenticate an unaware user on a different computer without ever knowing their USS. Although this is outside the scope of the TKey threat model, it is worth considering, as it leaves an opening for potential exploits. An alternative would be to make the TKey erase the information each time after its usage, but this might make the TKey less user friendly, as it forces the user to interact with the TKey more often to provide input.

5.3 Future work

Despite that several weaknesses were found in this security assessment, the two projects might contain more bugs and weaknesses that were not discovered. Additional security evaluations might therefore make new findings that could improve and work as a complement to this assessment, especially if those evaluations investigated more niche attack vectors that were not covered in this thesis.

Other work that can be done in the future is to develop and deploy patches that fixes the vulnerabilities found in this project. Some of the vulnerabilities are easy to fix while others might require more work. Building a standardized library for the TKey integration that is secure could also be a future idea.

5.4 Societal and ethical aspects

Ethical hacking introduces both societal and ethical issues. The biggest issue is that this project provides a step-by-step guide on what vulnerabilities exist and how to take

advantage of them. This is always a large problem with penetration testing and ethical hacking, but it is often justified by the fact that these vulnerabilities probably would have been found regardless of this report. It is therefore relevant to conduct ethical hacking, as it mitigates risks, allowing for preventative measures that would unlikely have been performed without such security assessments.

While the applications tested were only prototypes and therefore had no users at risk, the Tillitis Authentication Ecosystem was not. This means that users currently using the authentication system could be exposed to the TKey related vulnerabilities detailed in this report. If users were to be exposed, it could hurt the reputation of the Tillitis Ecosystem and also Tillitis itself. This could also lead to financial losses for Tillitis and general distrust to the usage of physical authentication keys, which would be unfortunate, as the use of physical authentication keys reduce security problems that common authentication methods such as passwords normally have.

To conclude, the findings of this analysis are not meant to be considered as a complete security analysis, as there is no guarantee that the removal of the detected vulnerabilities would result in a completely secure product.

References

[1] R. Alomari and J. Thorpe, "On password behaviours and attitudes in different populations", *Journal of Information Security and Applications*, vol. 45, pp. 79–89, 2019, ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2018.12.008.

[2] MojoAuth Research, *The state of passwordless authentication 2026*, 2026. Accessed: Feb. 10, 2026. [Online]. Available: https://mojoauth.com/data-and-research-reports/state-of-passwordless-2026/.

[3] Tillitis AB. "Introduction | tkey developer handbook", Accessed: Feb. 10, 2026. [Online]. Available: https://dev.tillitis.se/intro/.

[4] I. A. Horan, A. Ali, W. Greppe, M. Levin, A. Skeppstedt, and M.-A. Zabateh, "Passwordless authentication system using tkey: Leveraging asymmetric cryptography with tkey for secure passwordless authentication", Bachelor's thesis, Chalmers University of Technology and University of Gothenburg, 2025.

[5] J. Almström, C. Forssell, K. Johansson, L. Lundahl, I. Snecker, and S. W. Green, "Passwordless authentication using the tkey device: Bachelor's thesis in computer science and engineering", Bachelor's thesis, Chalmers University of Technology and University of Gothenburg, 2025.

[6] Parmar et al., "A comprehensive study on passwordless authentication", in *2022 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS)*, 2022, pp. 1266–1275. DOI: 10.1109/ICSCDS53736.2022.9760934.

[7] C. Shearon, "The new standard for cyber security", in *2020 Pan Pacific Microelectronics Symposium (Pan Pacific)*, 2020, pp. 1–9. DOI: 10.23919/PanPacific48324.2020.9059551.

[8] Tillitis AB. "Threat model", Accessed: Feb. 10, 2026. [Online]. Available: https://github.com/tillitis/tillitis-key1/commits/main/doc/threat_model/threat_model.md.

[9] Brendel et al., "The provable security of ed25519: Theory and practice", in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1659–1676. DOI: 10.1109/SP40001.2021.00042.

[10] J. Guo, P. Karpman, I. Nikolić, L. Wang, and S. Wu, "Analysis of blake2", in *Topics in Cryptology – CT-RSA 2014*, J. Benaloh, Ed., Cham: Springer International Publishing, 2014, pp. 402–423, ISBN: 978-3-319-04852-9.

[11] Tillitis AB, "Tkey introduction @ chalmers", Unpublished.

[12] M. Nieles, K. Dempsey, and V. Y. Pillitteri, "An introduction to information security", National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST Special Publication (SP) 800-12, Rev. 1, 2017. DOI: 10.6028/NIST.SP.800-12r1.

[13] Tillitis AB. "Tools & libraries | tkey developer handbook", Accessed: May 13, 2026. [Online]. Available: https://dev.tillitis.se/tools/.

[14] NIST. “Proxy server”, Accessed: Apr. 18, 2026. [Online]. Available: https://csrc.nist.gov/glossary/term/proxy_server.

[15] O. Foundation, *Owasp api security top 10*, 2026. Accessed: Apr. 28, 2026. [Online]. Available: https://owasp.org/www-project-api-security/.

[16] O. Foundation, *Owasp secure headers project*, 2026. Accessed: Apr. 28, 2026. [Online]. Available: https://owasp.org/www-project-secure-headers/.

[17] D. Inc. “What is docker?”, Accessed: May 4, 2026. [Online]. Available: https://docs.docker.com/get-started/docker-overview/#:~:text=it%20in%20production.-,The%20Docker%20platform,simultaneously%20on%20a%20given%20host..

[18] OWASP. “Business logic vulnerability”, Accessed: Feb. 12, 2026. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Business_logic_vulnerability.

[19] PortSwigger, *Lab: Excessive trust in client-side controls*, Accessed: 2026-06-01. [Online]. Available: https://portswigger.net/web-security/logic-flaws/examples/lab-logic-flaws-excessive-trust-in-client-side-controls.

[20] PortSwigger, *Information disclosure vulnerabilities*, Accessed: 2026-06-01. [Online]. Available: https://portswigger.net/web-security/information-disclosure.

[21] PortSwigger, *Lab: Information disclosure in error messages*, Accessed: 2026-06-01. [Online]. Available: https://portswigger.net/web-security/information-disclosure/exploiting/lab-infoleak-in-error-messages.

[22] Stallings, William, Brown, and Lawrie, *Computer Security: Principles and Practice*, 4th ed. New York, NY: Pearson, 2018.

[23] Fortinet. “What is a brute force attack?”, Accessed: May 6, 2026. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/brute-force-attack.

[24] OWASP. “Cross site scripting (xss)”, Accessed: Apr. 14, 2026. [Online]. Available: https://owasp.org/www-community/attacks/xss/.

[25] OWASP. “Sql injection”, Accessed: Apr. 14, 2026. [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection.

[26] OWASP. “Cross site request forgery (csrf)”, Accessed: Apr. 14, 2026. [Online]. Available: https://owasp.org/www-community/attacks/csrf.

[27] EC-Council. “Understanding the five phases of the penetration testing process”, Accessed: May 18, 2026. [Online]. Available: https://www.eccouncil.org/cybersecurity-exchange/penetration-testing/penetration-testing-phases/.

[28] OWASP. “Owasp risk rating methodology”, Accessed: May 7, 2026. [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology.

[29] W. Greppe, themaxlevin, A. Skeppstedt, and A. Ali, *TKey-group-95*, Commit: 605bf09, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/SkepparAbbe/TKey-group-95/blob/605bf0916f77f0176ad2d7450e4c0273e15dda16/Web-application/docker-compose.yml#L26-L32.

[30] W. Greppe, themaxlevin, A. Skeppstedt, and A. Ali, *TKey-group-95*, Commit: 605bf09, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/SkepparAbbe/TKey-group-95/blob/605bf0916f77f0176ad2d7450e4c0273e15dda16/Web-application/flask_app/auth/register.py#L84-L88.

[31] W. Greppe, themaxlevin, A. Skeppstedt, and A. Ali, *TKey-group-95*, Commit: 605bf09, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/SkepparAbbe/TKey-group-95/blob/605bf0916f77f0176ad2d7450

e4c0273e15dda16/Web-application/flask_app/static/js/index.js#L169-L184.

[32] Pallets Projects, *Werkzeug debugging applications*, Accessed: 2024-05-04, 2024. [Online]. Available: https://werkzeug.palletsprojects.com/en/3.0.x/debug/.

[33] W. Greppe, themaxlevin, A. Skeppstedt, and A. Ali, *TKey-group-95*, Commit: 605bf09, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/SkepparAbbe/TKey-group-95/blob/605bf0916f77f0176ad2d7450e4c0273e15dda16/Proxy-server/tkey/main.go#L23-L24.

[34] W. Greppe, themaxlevin, A. Skeppstedt, and A. Ali, *TKey-group-95*, Commit: 605bf09, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/SkepparAbbe/TKey-group-95/blob/605bf0916f77f0176ad2d7450e4c0273e15dda16/Web-application/flask_app/auth/recover.py#L82-L83.

[35] S. W. Green, J. Almström, Luqas, I. Snecker, Flaskhals68, and Carl, *Datx11-bachelors-thesis-project*, Commit: 7bd649c, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/simon-wg/DATX11-Bachelors-thesis-project/blob/7bd649ccb6e82b954fba0ddf136fe6693557063d/application/gui/src/util/csrf.js#L25-L27.

[36] Tillitis AB, *tkey-device-signer*, Commit: b160b68, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/tillitis/tkey-device-signer/blob/b160b68fbdcf4b07899ad8d8fac07d87102e6996/signer/main.c#L27-L29.

[37] Tillitis AB, *tkey-device-signer*, Commit: b160b68, GitHub, 2025. Accessed: May 11, 2026. [Online]. Available: https://github.com/tillitis/tkey-device-signer/blob/b160b68fbdcf4b07899ad8d8fac07d87102e6996/signer/main.c#L190-L197.

[38] Tillitis AB, *tkeyclient*, Commit: 61a8f67, GitHub, 2021. Accessed: May 11, 2026. [Online]. Available: https://github.com/tillitis/tkeyclient/blob/61a8f675286f1bd6568f7240753654ffbaeaa026/tkeyclient.go#L362-L369.

[39] Tillitis AB, *tillitis-key1*, Commit: 2ffd46c, GitHub, 2021. Accessed: May 11, 2026. [Online]. Available: https://github.com/tillitis/tillitis-key1/blob/2ffd46cd31d2d27d69db938535ff970805ddbe5c/hw/application_fpga/fw/tk1/main.c#L146-L149.

[40] Tillitis AB, *Security Advisory: tkeyclient*, Tillitis Blog, Security, 2026. Accessed: May 11, 2026. [Online]. Available: https://www.tillitis.se/blog/2026/03/17/tkeyclient-advisory/.