# Exercises
## Web Vulnerabilities

Computer Security

Watson Files is a new system that lets you store files in the cloud. Customers complain that old links often return a "*404 File not found*" error. Sherlock decides to fix this problem modifying how the web server responds to requests for missing files. The Python-like pseudocode that generates the "missing file" page is the following:

```
1  def missing_file(cookie, reqpath):
2      print "HTTP/1.0 200 OK"
3      print "Content-Type: text/html"
4      print ""
5      user = check_cookie(cookie)
6      if user is None:
7          print "Please log in first"
8          return
9
10     print "<p>We are sorry, but the server could not locate file" + reqpath
11     print "<br>Try using the search function.</p>"
```

where `reqpath` is the requested path (e.g., if the user visits *https://www.watsonfiles.com/dir/file.txt*, the variable `reqpath` contains `dir/file.txt`). The function `check_cookie` returns the username of the authenticated user checking the session cookie (this function is securely implemented and does not have vulnerabilities).

| *Vulnerability class* | *Is there a vulnerability belonging to this class in Sherlock's code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **cross-site scripting (XSS)** | | |
| **Cross site request forgery (CSRF)** | | |

| *Vulnerability class* | *Is there a vulnerability belonging to this class in Sherlock's code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **cross-site scripting (XSS)** | Yes, an attacker can supply a filename containing e.g., <script>alert(document.cookie)</script>, and the web server would print that script tag to the browser, and the browser will run the code from the URL. | The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the "book" variable. |
| **Cross site request forgery (CSRF)** | No, there is no state-changing action in the page that needs to be protected against CSRF. | |

To download the files stored in Watson Files, users visit the page */download*, which is processed by the following server-side pseudocode:

```
 1  def download_file(cookie, params):
 2    # code to initialize the HTTP response
 3    user_id = check_cookie(cookie)
 4    if user is None:
 5      print "Please log in first"
 6      return
 7    filename = params['filename']
 8    query = "SELECT file_id, data FROM files WHERE FILENAME = '" +
filename + "';"
 9    result = db.execute(query)
10    # code to print result['data']
```

where `params` is a dictionary containing the GET parameters (e.g., if a user visits */download?filename=holmes.txt*, then `params['filename']` will contain '`holmes.txt`'). The database queries are executed against the following tables:

users

| user_id | username | password |
|---|---|---|
| 1 | Aria | 1234 |
| 2 | John | password |
| 3 | Tyrion | bestpass |
| 4 | Daenerys | unknown |

files

| file_id | filename | data |
|---|---|---|
| 1 | unk | ... |
| 2 | secret.txt | ... |
| 3 | stuff.jpg | ... |
| 4 | summer17.jpg | ... |

**1. [2 points]** Identify the class of the vulnerability and briefly explain how it works **in general**.

**2. [2 points]** Write an exploit for the vulnerability just identified to get the **password** of user **John**:

**1. [2 points]** Identify the class of the vulnerability and briefly explain how it works **in general**.

***SQL Injection***. *See slides for the explanation.*

There must be a data flow from a **user-controlled HTTP variable** (e.g., parameter, cookie, or other header fields) **to a SQL query**, **without appropriate filtering and validation**. If this happens, the **SQL structure of the query can be modified**.

**2. [2 points]** Write an exploit for the vulnerability just identified to get the **password** of user **John**:

```
' UNION SELECT user_id, password FROM users WHERE name = 'John';--
```

A web page contains the following code:

```
//get the username from the HTTP request
var username = request.get['username'];
```

which is processed by the following query:

**"SELECT * FROM users WHERE name='" + username + "'";**

which is executed on these tables →

| id | name | gender | surname |
|---|---|---|---|
| 1 | Adams | M | Ansel |
| 2 | Baker | M | Toast |
| 3 | Marco | M | Santa |
| 4 | Dood | F | Lame |

| user_id | role | salary |
|---|---|---|
| 1 | Prof | 80 |
| 2 | Student | 20 |
| 3 | Prof | 100 |
| 4 | Admin | 70 |

- Identify the class of the vulnerability and briefly explain how it works.
- Write an exploit to get the **salary** of Marco.
- You are the database administrator and have **no way to modify the above code**. How would you alternatively mitigate the damage that an attacker can do?

1. SQL injection. Unfiltered, user-controlled data is concatenated to a query, allowing an attacker to modify such query.
2. We can first let a query run to figure out the id, and then simply do: ’ UNION ALL SELECT salary, name, gender, surname FROM users JOIN role ON user_id=id WHERE name=Marco;
3. Assign the roletable different permissions than those of the users table, or simply deny the UNION operations on certain queries.

“SHIPSTUFF” is a new online service that allow registered users to send stuff to other registered users by filling a form. The form must contain the `product_id` of the product to send and the `receiver_id` of the receiver. After clicking on the submit button, the web browser will make the following GET request to the web server:

```
https://shipstuff.org/ship?product_id=<product_id>&receiver_id=<receiver_id>
```

The Python-like pseudocode that will handle the shipment is the following:

```
1  def ship_stuff(request):
2      # ... code to send HTTP header (not relevant) ...
4      user = check_cookie(request.cookie)
5      if user is None:
6          print "Please log in first"
7          return
8
9      product_id = request.params['product_id']       # GET parameter
10     receiver_id = request.params['receiver_id']     # GET parameter
11
12     query1 = 'SELECT p_id, product_name FROM warehouse, ownership \
13         WHERE p_id = ' + product_id + \
14         'AND user.id = ' + ownership.u_id + \
15         'AND ownership.p_id = ' + warehouse.p_id + ';'
16     db.execute(query)
17     row = db.fetchone()
18     if row is None:
19         print "Product", product_id, "is not existent"
20         return
21
22     query2 = 'SELECT u_id, username FROM accounts \
23         WHERE u_id =  ' + receiver_id + ';'
24     db.execute(query)
25     row = db.fetchone()
26     if row is None:
27         print "User", receiver_id, "is not existent"
28         return
29 # ....... code to actually send the product (not relevant) ........
```

The above code checks if the user is logged in using the function `check_cookie`, which returns the username of the authenticated user checking the session cookie. Then, the code attempts to retrieve the `product_id` or the `receiver_id` from the database and, if they cannot be located the page will contain an error message.

Assume that `request.params['product_id']` and `request.params[receiver_id']` are controllable by the user, and that all the functionalities concerning the user authentication (i.e., `check_cookie`) are securely implemented and do not contain vulnerabilities.

**1. [4 points]** Only considering the code above**,** identify which of the following classes of web application vulnerabilities are present:

| *Vulnerability class* | *Is there a vulnerability belonging to this class in the code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **Stored cross-site scripting (XSS)** | | |
| **Reflected cross-site scripting (XSS)** | | |
| **Cross site request forgery (CSRF)** | | |
| **Sql Injection** | | |

| *Vulnerability class* | *Is there a vulnerability belonging to this class in the code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **Stored cross-site scripting (XSS)** | No, the above code does not allow to store information on the server that can be exploited. | |
| **Reflected cross-site scripting (XSS)** | Yes, an attacker can supply a product_id or the receiver_id containing e.g., <script>alert(document.cookie)</script>, and the web server would print that script tag to the browser, and the browser will run the code from the URL. | The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the vulnerable variable. For example: product_id |
| **Cross site request forgery (CSRF)** | Yes, an attacker can send a link to a victim and let the victim ship a product to him by just visiting the link. For example: https://shipstuff.org/ship?product_id=<a_product_i_want>&receiver_id=<my_u_id>) | Use CSRF token. |
| **Sql Injection** | Yes, because user-controlled data is concatenated a query, allowing an attacker to modify such query. | The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the vulnerable variable. For example: product_id |

Now assume that SHIPSTUFF executes all the database queries against the following tables:

**accounts**

| u_id | username | password |
|---|---|---|
| 1 | Rick | 1234 |
| 2 | Morty | password |
| 3 | PickleRick | bestpass |
| 4 | Mr meeseeks | unknown |
| ... | ... | ... |

**ownership**

| u_id | p_id |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |
| 4 | 4 |
| ... | ... |

**products**

| p_id | product_name |
|---|---|
| 1 | nothing |
| 2 | paper |
| 3 | flag |
| 4 | excalibur |
| ... | ... |

**2. [2 points]** Write an exploit for one of the vulnerability/ies just identified to get the **username** and the **password** of the **only** user that owns the product 'excalibur'.

By assuming that products are unique, state all the necessary steps and conditions for the exploit to take place.

Now assume that SHIPSTUFF executes all the database queries against the following tables:

**accounts**

| u_id | username | password |
|---|---|---|
| 1 | Rick | 1234 |
| 2 | Morty | password |
| 3 | PickleRick | bestpass |
| 4 | Mr meeseeks | unknown |
| ... | ... | ... |

**ownership**

| u_id | p_id |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |
| 4 | 4 |
| ... | ... |

**products**

| p_id | product_name |
|---|---|
| 1 | nothing |
| 2 | paper |
| 3 | flag |
| 4 | excalibur |
| ... | ... |

**2. [2 points]** Write an exploit for one of the vulnerability/ies just identified to get the **username** and the **password** of the **only** user that owns the product 'excalibur'.

By assuming that products are unique, state all the necessary steps and conditions for the exploit to take place.

```
' UNION SELECT username, password FROM accounts AS a,
ownership AS o, products AS p WHERE o.u_id = a.u_id
AND o.p_id = p.p_id AND p.product_name =
'excalibur';--
```

A web application contains three pages to handle login, post comments, and read comments, all served over a secure HTTPS connection. Here you can find code snippet of these pages:

```
Show comments: handler for the GET request /comments?id=<id>00
A.01  var id = request.get['id'];
A.02  var prep_query = prepared_statement("SELECT username FROM users WHERE id=? LIMIT 1");
A.03  var username = query(prep_query, id);
A.04  var prep_query = prepared_statement("SELECT * FROM comments WHERE username=?");
A.04  var comments = query(prep_query, username);
A.06  for comment in comments {
A.07        echo htmlentities(comment);
A.08  }
Login: handler for the POST request /login
B.01  var password = md5(request.post['password']);
B.02  var username = request.post['username'];
B.03  var prep_query = prepared_statement("SELECT username FROM users
B.04                                          WHERE username=? AND password=? LIMIT 1");
B.05  var result = query(prep_query, username, password);
B.06  if (result) {
B.07        session.set('username', username);
B.08        echo "Logged in.";
B.09  } else {
B.10        echo "User" + username + "does not exists!";
B.11  }
Write comment: handler for the GET request /write?comment=<text_of_the_comment>
C.01  var username = session.get['username'];        // You need to be logged in
C.02  var comment = request.get['comment'];
C.03  var res = query("INSERT INTO comments (username, comment, timestamp)
C.04                              VALUES ( “ +  username + ” , “+  comment + ” , NOW())");
C.05        echo "Comment saved.";
```

Assume the following:

- The framework used to develop the web application securely and transparently manages the users' sessions through the object `session`;
- The dictionaries `request.get` and `request.post` store the content of the parameters passed through a GET or POST request respectively;
- the function `htmlentities()` converts special characters such as `<`, `>`, `"`, and `'` to their equivalent HTML entities (i.e., `<`, `>`, `&quot;` and `&apos;` respectively).

As it is clear from the code, this application uses a database to store data. These are tables of the database:

**users**

| *id* | *name* | *password* |
|---|---|---|
| 1 | Rick | 76ceaaa34826979e77 |
| 2 | Marcello | 563c39089151f9df26 |
| 3 | admin | d1e576b71ccef5978d |

**comments**

| *id* | *user* | *comment* | *timestamp* |
|---|---|---|---|
| 1 | admin | "Welcome" | 03:23:21 24/12/2000 |
| 2 | Marcello | "malcontento..." | 18:31:62 17/02/2015 |

**1. [4 points]** Only considering the code above**,** identify which of the following classes of web application vulnerabilities are present:

| *Vulnerability class* | *Is there a vulnerability belonging to this class in the code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **Stored cross-site scripting (XSS)** | **Lines:** | |
| **Reflected cross-site scripting (XSS)** | **Lines:** | |
| **Cross site request forgery (CSRF)** | **Lines:** | |
| **Sql Injection** | **Lines:** | |

**1. [4 points]** Only considering the code above, identify which of the following classes of web application vulnerabilities are present:

| *Vulnerability class* | *Is there a vulnerability belonging to this class in the code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **Stored cross-site scripting (XSS)** | **Lines:**<br>No, ... | No vulnerability |
| **Reflected cross-site scripting (XSS)** | **Lines: B.10**<br>*Yes, an adversary can set up a form (hidden form) that submits a request with an username containing a malicious script e.g., <script>alert(document.cookie)</script>, and the web server would print that script tag to the browser, and the browser will run the code.* | *The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the "username" variable.* |
| **Cross site request forgery (CSRF)** | **Lines: C01-C04**<br>*Yes, an adversary can set up a form that submits a request to send a message, as this request will be honored by the server.* | *To solve this problem, include a CSRF token with every legitimate request, and check that cookie['csrftoken'] ==param['csrftoken'].* |
| **Sql Injection** | **Lines: C0.3-C0.4**<br>Yes, ... | *The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the "comment/username" variable.* |

**2. [2 points] Exploiting one of the vulnerability detected before, write down an exploit to get the hash of the password of admin. You must also specify all the steps and assumptions.**

**2. [2 points] Exploiting one of the vulnerability detected before, write down an exploit to get the hash of the password of admin. You must also specify all the steps and assumptions.**

*... comment = ‘( SELECT password from users where name = “admin”)*

As part of its core functionalities, Watson Files allows every user to retrieve a shareable public download link to every file he\she can access.
The way this works is the following: when a user wants to retrieve the download link for, as an example, the file '*secret.txt*', he\she will visit a page ('/link') passing the filename as a GET parameter (i.e., https://watsonfiles.com/link?filename=secret.txt). Then, the backend code will perform access control checks and redirect him\her to https://watsonfiles.com/files/sFPVMasg , where sFPVMasg is the *token* associated with the file (every file is associated with a *pre-generated token*), the download link is accessible without authentication.

The Python-like pseudocode of the backend that manages the link retrieval is the following:

```
1  def retrieve_download_link(request):
2    # .... code to initialize the HTTP response and to retrieve
3    # the user's session information into the object "user" (see previous point)
4
5    filename = request.params['filename']
6    query = 'SELECT filename, token FROM files, permissions WHERE filename = ' +
filename + ' AND permissions.u_id = ' + user.id + ' AND permissions.f_id =
files.f_id;'
7    db.execute(query)
8    row = db.fetchone()
9
10   # ... code to generate a redirect to https://watsoncode.org/files/<row.token>
```

where `params` is a dictionary containing the GET parameters (e.g., if a user visits */link?filename=holmes.txt*, then `params['filename']` will contain `'holmes.txt'`).
The output of the query is limited to only one result.

**3. [1 point] You are the database administrator and have no way to modify the above code. How would you mitigate the damage that an attacker can do?**

**3. [1 point] You are the database administrator and have no way to modify the above code. How would you mitigate the damage that an attacker can do?**

*As this page\application needs only to read data from the users table, we could restrict, at the database level, the privileges of the user of this application to only perform SELECTs involving the user table (and no operation involving the account_balance table).*

**4 [1 point]. Consider the implementation of the session management mechanism:**

```
function session.set(key, value) {
    response.add_header("Set-Cookie: " + key + "=" + value);
}
```

**Likewise, the `session.get()` function parses the Cookie header in the HTTP request and returns the value of the cookie with the specified key. Describe how an attacker can exploit a vulnerability in this implementation to authenticate as an existing user, and suggest a way to change the function `session.set()` to fix this vulnerability.**

**4 [1 point]. Consider the implementation of the session management mechanism:**

```
function session.set(key, value) {
    response.add_header("Set-Cookie: " + key + "=" + value);
}
```

**Likewise, the `session.get()` function parses the Cookie header in the HTTP request and returns the value of the cookie with the specified key. Describe how an attacker can exploit a vulnerability in this implementation to authenticate as an existing user, and suggest a way to change the function `session.set()` to fix this vulnerability.**

Cookie: username=any existing username

Fix:

- Encrypt the cookie with a key stored on the server (with a nonce and an expiration to avoid replay attacks)
- Store the session data somewhere (e.g., file, database, …) indexed by a random value and set the random value in the cookie instead of the username

“Watson Files” is a cloud computing services to store files similar to Dropbox or Google Drive.
Customers complain that old links often return a “*404 File not found*” error. Sherlock decides to fix this problem modifying how the web server responds to requests for missing files.

The Python-like pseudocode that generates the “missing file” page is the following:

```
1  def missing_file(request):
2      print "HTTP/1.0 200 OK"
3      print "Content-Type: text/html"
4      print ""
5      user = check_cookie(request.cookie)
6      if user is None:
7          print "Please log in first"
8          return
9
10     print "<p>We are sorry, but the server could not locate file" + request.path
11     print "<br>Try using the search function.</p>"
```

The code checks if the user is logged in using the function `check_cookie`, which returns the username of the authenticated user checking the session cookie. If the user is authenticated, the page will contain a message detailing the path of the file that could not be located.
Assume that the `request.path` variable is populated by the web server with the path requested by the user (e.g., if the user visits *https://www.watsonfiles.com/dir/file.txt*, the variable `request.path` contains `dir/file.txt`), and that all the functionalities concerning the user authentication (i.e., `check_cookie`) are securely implemented and do not contain vulnerabilities.

| *Vulnerability class* | *Is there a vulnerability belonging to this class in the code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **<u>Stored</u> cross-site scripting (XSS)** | | |
| **<u>Reflected</u> cross-site scripting (XSS)** | | |
| **Cross site request forgery (CSRF)** | | |

| *Vulnerability class* | *Is there a vulnerability belonging to this class in the code? If so, explain why it is present and specify how an adversary could exploit it.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability.* |
|---|---|---|
| **<u>Stored</u> cross-site scripting (XSS)** | No, the above code does not allow to store information on the server that can be exploited. | |
| **<u>Reflected</u> cross-site scripting (XSS)** | Yes, an attacker can supply a filename containing e.g., <script>alert(document.cookie)</script>, and the web server would print that script tag to the browser, and the browser will run the code from the URL. | The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the "request.path" variable. |
| **Cross site request forgery (CSRF)** | No, there is no state-changing action in the page that needs to be protected against CSRF. | |

The database queries are executed against the following tables:

**users**

| u_id | username | password |
|---|---|---|
| 1 | Rick | 1234 |
| 2 | Morty | password |
| 3 | PickleRick | bestpass |
| 4 | Mr meeseeks | unknown |
| ... | ... | ... |

**permissions**

| u_id | f_id |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |
| 4 | 3 |
| ... | ... |

**files**

| f_id | filename | token |
|---|---|---|
| 1 | unk | 75ZsvBcU |
| 2 | secret.txt | sFPVMasg |
| 3 | flag.txt | 5vcV6xdU |
| 4 | summer.jpg | xmEBFPLU |
| ... | ... | ... |

**3. [2 points]** Identify the class of the vulnerability, briefly explain how it works **in general**.

The database queries are executed against the following tables:

**users**

| u_id | username | password |
|---|---|---|
| 1 | Rick | 1234 |
| 2 | Morty | password |
| 3 | PickleRick | bestpass |
| 4 | Mr meeseeks | unknown |
| ... | ... | ... |

**permissions**

| u_id | f_id |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |
| 4 | 3 |
| ... | ... |

**files**

| f_id | filename | token |
|---|---|---|
| 1 | unk | 75ZsvBcU |
| 2 | secret.txt | sFPVMasg |
| 3 | flag.txt | 5vcV6xdU |
| 4 | summer.jpg | xmEBFPLU |
| ... | ... | ... |

**3. [2 points]** Identify the class of the vulnerability, briefly explain how it works **in general**.

***SQL Injection***. *See slides for the explanation.*

There must be a data flow from a **user-controlled HTTP variable** (e.g., parameter, cookie, or other header fields) **to a SQL query**, **without appropriate filtering and validation**. If this happens, the **SQL structure of the query can be modified**.

**4. [2 points]** Write an exploit for the vulnerability just identified to get the **username** and the **password** of the **only** user authorized to access the file 'flag.txt'. By assuming that filenames are unique, state all the necessary steps and conditions for the exploit to take place.

**4. [2 points]** Write an exploit for the vulnerability just identified to get the **username** and the **password** of the **only** user authorized to access the file 'flag.txt'. By assuming that filenames are unique, state all the necessary steps and conditions for the exploit to take place.

As an authenticated user I make a GET request with the following 'filename' parameter:

```
' UNION SELECT username, password FROM users,
permissions, files WHERE permissions.u_id =
users.u_id AND permissions.f_id = files.f_id AND
files.filename = 'flag.txt';--
```

**5. [1 points]** Explain the simplest procedure to remove this vulnerability.

**6. [2 points]** What is a **blind SQL injection**? Does it mean it cannot be exploited?

**5. [1 points]** Explain the simplest procedure to remove this vulnerability.

Use parameterized queries, ...

**6. [2 points]** What is a **blind SQL injection**? Does it mean it cannot be exploited?

In a blind injection the data retrieved by the modified SQL query is not displayed back to the attacker. This can still be exploited: changes can be blindly executed in the database, or by using side-effects of queries the attacker can guess the answers.

Consider SQL injection vulnerabilities and attacks.

1. (2 points) Explain briefly which conditions must hold on a website in order to expose a SQL injection vulnerability.

2. (2 points) What is a blind SQL injection? Does it mean it cannot be exploited?

Consider SQL injection vulnerabilities and attacks.

1. (2 points) Explain briefly which conditions must hold on a website in order to expose a SQL injection vulnerability.

There must be a data flow from a user-controlled variable (e.g., http parameter, cookie, or other header fields) to a SQL query, without appropriate filtering and validation. If this happens, the SQL structure of the query can be modified.

2. (2 points) What is a blind SQL injection? Does it mean it cannot be exploited?

In a blind injection the data retrieved by the modified SQL query is not displayed back to the attacker. This can still be exploited: changes can be blindly executed in the database, or by using side-effects of queries the attacker can guess the answers.

The End