CS5331 - Assignment 3: Scan them all!

Due date: Monday, 20 April 2015, 23:59 pm SGT

1   Assignment overview

Bug scanner   You are expected to implement a scanner to scan all the web vulnerabilities belonging to one particular category. The scanner should be implemented on your own, from scratch. You can use all supporting utility libraries which have the MIT or BSD license but the logic of the scanner needs to be implemented by you.

   Implementation Language: Only `Node.js` or `Python`.

Vulnerability categories   Each scanner will scan a particular bug category. All the categories are listed in Table 1. To make sure that every category is touched, we limit the number of groups working in each category as in Table 1. At 23.00 PM on Tuesday 17 March, there will be a post in IVLE forum for you to select the category you want to work on. This will be a first-come first-serve rule, the group makes a reply to that post earlier will get their desired category. Each group should make only one reply to select their category. At 23:30 PM that day, the TA will finalize the bug category distribution. The available categories will be randomly assigned to the groups who haven't made their selection.

Benchmark creation   A public benchmark will be given to you so that you can test your scanner. However, we expect you to create your own benchmark while building the scanner. The benckmark can be constructed as simple as the testcases we gave you in Assignment 2, or can be the real bugs reported in real web applications. We will ask you to submit your benchmark along with your scanner. It will be graded and account for 5 points.

Category	Reflected XSS	Persistent XSS	CSRF flaw	SQL Injection	Sever side Code Injection	Directory traversal	Unvalidated redirection	Cookies management
No. of maximum groups	2	2	3	2	2	2	2	2

Table 1: Bug categories and number of maximum groups can work on each category. Note that Server side code injection and Directory traversal are included in one scanner.

2   Scanner description

This is a general description about the structure of a scanner. You should follow all the steps below while building your scanner. Typically, the scanner will consist of 4 phases. The communication between these phases *must* be via files in `JSON` format.

2.1 Crawling and identification of Injection points in web pages

Crawling web pages The aim of this phase is to find out as many pages in the web application as possible. The crawling function takes in the start URL, the depth to crawl (optional), maximum number of URLs (optional). You can use open source crawlers like Scrapy or implement your own crawler. We encourage you to use Scrapy, so that you can later on discuss on forum about how to do things with Scrapy. We also prepare a simple Scrapy script to crawl one example in the code base that we give you. You can consider it as one reference to get used to with Scrapy. Nonetheless, there are tons of helpful tutorials about how to use Scrapy on the Internet.

Remember that most of the pages in your application are only available to the scanner once you log in. You are given the account credentials for all the apps in the **readme** file in the VM. Use this information to configure your crawler to scan for more pages after logging in. The more pages you are able to crawl for, the more vulnerabilities your scanner will be able to find.

Identification of Injection points After crawling a web page, you will need to identify injection points in each of the downloaded pages. An attacker can inject his attack vector at numerous points. Some of the injection points are (but are not limited to):

- GET requests parameters

- POST requests parameters

- Request Headers (including cookies)

The json output file, which will be used in the subsequent phases will look similar to the one given below:

```
[{
        "/page.html": [{
                "type":"GET",
                "param":"foo"
        }],
        "/page2.html":[{
                "type": "Header",
                "param":"X-Requested-By"
        }]
}]
```

2.2 Payload generation

This phase is specific to your vulnerability category. These payloads will be injected in the injection points that you have identified in the previous phase. This phase should return a set of exploits to be tested. Pass information that you think would be necessary for you to better choose the exploits you want to try out. For example, a SQL Injection scanner could take in information such as the Database type, version and the current server information to return the payloads (no point returning No-SQL payloads for a MYSQL Database). The output of this phase for a SQL Injection Scanner for a LAMP installation would be something like what is shown below

```
["' or '1'='1", "your other payloads", ...]
```

2.3 Payload Injection

The next step would be to inject every payload generated in Phase 2 into your injection points discovered in Phase 1. After each injection, you need to find out which of these (injection point, payload) pairs are exploitable. The output of this phase would be the list of confirmed exploits in the website. The output format for each vulnerability is described in Section 3 of the handout. A sample output for the SQL Injection scanner will look like this.

```
[{
        "http://example.com/index.html" : [{
                method: GET/POST,
                params:
                        [
                        key: param1
                        value: ' or '1'='1
                        ]
                }]
}]
```

2.4 Generate automated verification script

For each of the exploits you have printed in Phase 3, you need to generate an automated script which proves that the exploit is indeed vulnerable. Typically, you will have to demonstrate that you can achieve the scope of exploit for each category mentioned in Section 3. For some category, we will discuss more on how to demonstrate the exploit. The generated scripts from this phase should be similar to the scripts that you generated for the previous assignment. You can consider using Selenium to make all the process automated. Failing to automate any component in this script will not get full grade.

This phase is also to make sure that your scanner will not report false positive exploits — exploits that are not exploitable. We will deduct up to 5 points if your scanner raises too many false positive reports.

3 Scope and format of exploit in each category

3.1 Reflected XSS

Scope You are expected to execute an alert in the context of the current domain and determine whether it is exploitable with Chrome XSS-auditor turned on.

Final exploit format

```
[{
        "http://example.com/index.html" : [{
                method: GET/POST
                params:
                [
                        key: param1
                        value: attack1
                ]
        }]
}]
```

3.2 Persistent XSS

Scope Execute an alert in the context of the current domain. You can assume that the browser-XSS-auditor is turned off.

Final exploit format

```
1  [{
2        method: GET/POST,
3        params:
4        [
5                key: param1
6                value: attack1
7        ],
8        reflected_page: "http://example.com/page2.html"
9  }]
```

3.3 CSRF

Scope Carry out a sensitive operation on behalf of other users by exploiting the lack of proper csrf token/mechanism. Sensitive operation is the operation operated on sensitive data and generally is carried out by sensitive form. Note that normal form can be unprotected by CSRF if there is nothing sensitive that can be achieved by submitting that form.

Final exploit format You will need to output the page and name of the sensitive form which is vulnerable to CSRF attack. The format of the output should look similar to this.

```
1  [{
2        url: "http://example.com/page.html"
3        name: "form1"
4  }]
```

Exploit verification We leave how to validate the attack open ended and let the group decide. One possible approach is to demonstrate that user A can submit a sensitive form on behalf of B, then show the information changed from B perspective before and after A submits the form.

3.4 SQL Injection

Scope Output a database error message to the screen by injecting a syntax error command.

Final exploit format Your output for phase 3 should look similar to this.

```
1  [{
2        "http://example.com/index.html" : [{
3                method: GET/POST,
4                params:
5                [
6                        key: param1
7                        value: ' or '1'='1
8                ]
9        }]
10 }]
```

4

3.5 Server Side Code Injection

This category includes all variants of Local file inclusion, remote file inclusion, PHP code injection attacks.

Scope Inject a script (through LFI / RFI / Php code injection) which echoes the string "pawned" into the page

Final exploit format

```
1  [{
2         "http://example.com/index.html" : [{
3                 method: GET/POST
4                 params:
5                 [
6                         key: param1,
7                         value: ";echo "pawned"
8                 ]
9         }]
10  }]
```

3.6 Directory traversals

Scope Page should have the contents of the /etc/passwd file included in the response

Final exploit format

```
1  [{
2         "http://example.com/index.html" : [{
3                 method: GET/POST
4                 params:
5                 [
6                         key: param1,
7                         value: ../../etc/passwd
8                 ]
9         }]
10  }]
```

3.7 Unvalidated redirects

Scope Navigate the page to https://google.com.

Final exploit format

```
1  [{
2         "http://example.com/index.html" : [{
3                 method: GET/POST,
4                 params:
5                 [
6                         key: param1,
7                         value: https://google.com
```

```
8                  ]
9              }]
10  }]
```

3.8 Cookies management

Scope of exploit   You are expected to get the cookie of a logged in user. Your scanner should cover the following vulnerability sub-categories to steal the cookie or fix the session of the user.

- Session Fixation

- Session Hijacking, e.g., by MITM attack

- Predictable Cookies

Final exploit format   You need to report the page and the name of the cookie which makes the app vulnerable to any of the above mentioned attacks (the session cookie) and the page on which it is found. Note that you only need to report each cookie only once. For example, if the same session cookie is found on multiple pages of the application, you only need to report each cookie once.

```
1  [{
2         page: http://example.com/index.html,
3         cookie:[{
4                name:PHPSSESSID,
5                secure: true/false, // the secure flag
6                httpOnly: true/false, // the httpOnly flag
7                attack: sessionFixation || sessionHijacking || predictableCookies
8         }]
9  }]
```

Verification   The exploit script should be a selenium script which

1. automates the login of the victim (you can use the script that has been provided to you for Phase 1)

2. perform the attack as the attacker (using any of the 3 strategies mentioned above)

3. Log in as the victim or fix the session for the victim

4   Grading

This assignment accounts for 50% (50 points) of your final grade. The following scheme will be applied when we grade your scanner.

- Did you follow the given code structure? [10 points] — TA will check the output format of each of the 4 steps manually.

- How many bugs does your scanner automatically find on the given (public) benchmark? [10 points]

- Additional benchmark/testcases that you created to test your scanner? [5 points]

- How many bugs does your scanner find from our hidden benchmark? [20 points]

- Does your scanner raise false positives on our hidden benchmark [5 points - 0.5 points for each FP raised]?

More details on how the grading is conducted will be updated later.

5   Supporting resources

Public benchmark   You will be given a VM which includes all the bugs. We have enabled FTP on the course server so that you will be able to access it. To download the resources via FTP, use the following URL: `ftp://group0@andromeda.d2.comp.nus.edu.sg.` The password for accessing the FTP is `group0`. Please fetch the VM in the folder `Assignment_3`. A proper description of the VM and benchmark can be found in that folder also.

Code template   A simple code template in `Python` can be fetched from this repository.

6   Miscellany

Important dates

- Category selection: Tuesday, 17 March 2015, 23:00 PM SGT.

- Submission deadline: Monday, 20 April 2015, 23:59 PM SGT.

Contact

- TAs : Loi Luu & Li Guodong (cs5331.ta@gmail.com)

- IVLE forum