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 are restricted to use only utility libraries which have the MIT or BSD license. Specifically, the core logic of the scanner needs to be implemented by you. Note that you **will not have access** to the web site backend source code, or have any assumption about the server architecture, etc.

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

Vulnerability categories Each scanner will scan for 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 mentioned in Table 1. At 22.00 PM on Tuesday 17 March, there will be a post in the 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. Invalid entries (entries which exceed the specified limit) will not be considered. At 23:00 PM that day, the TA will finalize the bug category distribution. The remaining 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 benchmark can be as simple as the testcases we gave you in Assignment 2, or can be 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. You are expected to have at least 10 new testcases for your own benchmark, with good coverages to your bug category.

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.

2 Scanner structure

This is a general description about the structure of a scanner. You should follow all the steps below while building your scanner. 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

- **Input**: list of web URLs.
- **Output**: a JSON file(s) of injection points.

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 the forum about how to do things with `Scrapy`. We also prepare a simple `Scrapy` script to crawl one example in the code base for you. You can consider it as one reference to get started 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 crawled 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 output of this phase is a json file containing the pages and the injection points found in the page. This 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

- **Input**: Bug category.
- **Output**: a JSON file(s) of all possible payloads.

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 of all possible Database types, versions and the current server information to return the payloads. 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

- **Input**: Output of Phase 1 and Phase 2.
- **Output**: a JSON file(s) of all possible exploits.

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 7 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

- **Input**: Output of Phase 3
- **Output**: list of Selenium scripts to automate the validated exploit.

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 the category you have chosen in Section 7. For some categories, we will discuss more on how to demonstrate the exploit. Nonetheless, we leave how to verify the exploit for the group to decide, you may not follow the instruction in Section 7 and come up with your own approach. The generated scripts from this phase should be similar to the scripts that you generated for the previous assignment. Consider using Selenium to make all the process automated. Failing to automate any component in this script will not get you the full grade.

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

3 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/show the strengths of your scanner? [**5 points**]
- How many bugs does your scanner finds from our hidden benchmark? [**20 points**]
- Does your scanner raise false positives on our hidden benchmark [**5 points**] if no FP raised. We deduct 0.5 points for each FP raised.

Note that we will run your scanner from the host machine to scan the benchmark from the VM. Thus, again, do not make any assumption about the code, database, etc. of the site that your scanner will scan.

4 Deliverables and Submission

You are required to submit

- Your scanner source code, with proper documentation about how to compile, how to use, and other requirements. Your group should include the scanner source code, documentation in a zip file `GROUP_X.zip` and submit it to IVLE.
- Your own benchmark with a proper report similar to what you received from us for the public benchmark. The new benchmark should be included in the VM we gave you in this assignment, and submitted back to us. Please leave every instruction/information we need to know in the VM readme so that we can grade it properly.
- The results of your scanner for a) the given benchmark and b) your own benchmark (if available), including the final json output and the exploitable script.

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. The `md5` checksum of the zip file is `71482fbdc62685c9a5fb6b45e5ba53a5`

Code template A sample crawler code snippet in `Python` can be fetched from this repository.

6 Miscellany

Important dates

- Category selection: Tuesday, 17 March 2015, 22:00 SGT.
- Submission deadline: Monday, 20 April 2015, 23:59 SGT.

Contact

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

7 Scope and format of exploit in each category

7.1 Reflected XSS

Scope You are expected to execute an alert in the context of the current domain. It is more interesting, but not required, that your scanner can 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
                ]
        }]
}]
```

7.2 Persistent XSS

Scope Execute an alert in the context of the current domain.

Final exploit format

```
[{
        method: GET/POST,
        params:
        [
                key: param1,
                value: attack1
        ],
        reflected_page: "http://example.com/page2.html"
}]
```

7.3 CSRF

Scope Carry out a sensitive operation on behalf of other users by exploiting the lack of proper csrf token or other CSRF protection mechanisms. Sensitive operation is the operation operated on sensitive data and it is left to you to define which data is sensitive. 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.

```
[{
        url: "http://example.com/page.html",
        name: "form1"
}]
```

Exploit verification We leave how to validate the attack as 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's perspective before and after A submits the form.

7.4 SQL Injection

Scope Inject a SQL query to the database and return the payload to the user.

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

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

Verification We leave it to the group to decide how to verify if the attack works. Some possible approaches include injecting an SQL syntax error query to get the error payload back (but some app will truncate the error message, so you won't see it), or injecting a SQL sleep command (or blind SQL injection) to slow down the server. You are expected to find a proper way to verify your exploit.

7.5 Server Side Code Injection

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

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

Final exploit format

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

7.6 Directory traversals

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

Final exploit format

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

7.7 Unvalidated redirects

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

Final exploit format

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

7.8 Cookies management

Scope of exploit You are expected to get the cookie of a logged in user. Your exploit should raise no SSL error indicator/warnings which user can see. Your scanner should cover the following vulnerability sub-categories to steal the cookie or fix the session cookie of the user.

- Session Fixation
- Session Hijacking, e.g., by MITM attack
- Predictable Cookies

Note that you can steal cookies by exploiting other vulnerabilities, e.g., XSS bugs, however it will not be counted as cookie management bugs but XSS bugs. In general, the bug classification will be based only on the entry bug. You are allowed to use network attack, e.g., package sniffing, MITM, to steal/fix the 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.

```
[{
        page: http://example.com/index.html,
        cookie:[{
                name:PHPSSESSID,
                secure: true/false, // the secure flag
                httpOnly: true/false, // the httpOnly flag
                attack: sessionFixation || sessionHijacking || predictableCookies
        }]
}]
```

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 in Phase 1)
2. Performs the attack as the attacker (the attack can belong to any of the above 3 sub-categories)
3. Logs in as the victim or fix the session for the victim