Lab 7: Web - Security Tools and Techniques

Jonathan Harijanto

March 1, 2017

CS 373: Defense Against the Dark Arts - Winter 2017

Abstract

The purpose of this blog is to describe my observation from the labs provided by Intel team. This blog will cover the testing methodology and the knowledge that was gain from completing the lab.

I. BLOG

A. What you looked at:

The topic for week 8 was about security on the web. There are two labs provided by Cedric and Kevin (the instructors), that could help the students understand the application of a web security concept. The first lab is about penetration testing, where I had to attack a login page and find the security weakness. The tool required for this lab was WebGoat; it is a web application that's purposefully designed to be insecure so that the user could exploit the vulnerability and learn how to fix it.

On the other hand, the second lab is quite different. The assignment is about URL classification, where I had to classify an URL to be either malicious or clean only by looking at the URL's characteristics (name, length, domain age, etc.). The tools that I used for this lab was a text editor, Sublime, to open up a python script called `readcorpus.py`.

B. How you looked at it & What you found:

1) Lab 1: WebGoat:

NOTE: I didn't see the WebGoat lab assignment on the VM and TopHat. Thus, I decided to do the WebGoat activities that were only being mentioned in the lecture video which are: Improper Error Handling, Stored XSS, and SQL Injection.

The first step required for this lab is to run the WebGoat. To do that, I opened a terminal and went to WebGoat directory. The, I need to type `sudo sh ./webgoat.sh start8080` to execute the WebGoat on port 8080. Finally, a link like `http://guest:guest@127.0.0.1:8080/WebGoat/attack` will appear, and when I clicked the link, a WebGoat's main page appeared in my Mozilla browser.

The first lesson that I need to do in WebGoat was called Fail Open Authentication Scheme under Improper Error Handling section. The most important thing to do was to open Tamper Data plugin so it could trace and time an HTTP response/ request. Next, I entered the word "guest" in the username and password field to see the list of HTTP response/ request in Tamper Data. The plugin showed that the 'fake' website created the login page correctly because it was done using POST request. Now, I need to figure out a way to bypass the login page without entering the password. According to Cedric, I need to inspect the element of the password field and changed the field name from 'Password' to 'Password2'. After doing that, I went back to the login page and entered 'guest' for my username and " for my password. Then, I used one of the Tamper Data's feature, by pressing the 'Start Tamper', to intercept and modify the HTTP traffic. After I had hit enter, I saw a tamper popup. Essentially, the popup showed me that I'm going to send `username: guest and password:` as requests to the web server. In the end, it was surprising to know that my login was successful. I believe this lesson wanted to show the vulnerability of login page that could be bypassed by changing the name of the elements (username and password).

The next lesson was Stored XSS under Cross-Site Scripting (XSS) section. The objective of this lesson was to show how to inject a script into someone's database permanently. The first step that I need to do was to log in as a person named Tom. This person's position was a manager, and we wanted to utilize our power to inject a script into our's employee profile page. Our target was a staff named Jerry, so I need to go to 'Search Staff' field and typed his name. In Jerry's profile, I clicked 'Edit' button and replaced his street address into `<script>alert("document.cookie");</script>` Then, I clicked 'Update' button to stored that data (script) permanently in his profile. Suddenly, I noticed there's a popup box with a string 'document.cookie' appeared. Hence, this shows that the 'Edit Profile' web page has a vulnerability because I could inject a script instead of a string into one of the data. Luckily, the script was just a popup saying "document.cookie". What if the script was malware, could you imagine what would happen to the web server for storing that script?

The last lesson was about SQL Injection under Injection Flaws section. I heard the term SQL Injection pretty often, but I never had a chance to do it. Thus, I was glad when WebGoat has this lesson. Anyhow, the objective of this lesson was to show that a login page could by bypassed by injecting an SQL query. My target was Neville's account because he is an admin of the website and I wanted to do evil stuff using his account. Thus, I had to set the username as Neville and inject some useful SQL query to bypass

the password. Fortunately, WebGoat provided me a hint to enter `smith' OR '1' = '1` as the SQL command. However, I just realized that the password field length provided by the web page is only 8 characters. Thus, I had to inspect the Password element and change the 'maxlength' parameter from 8 to 20. After I had done that, I typed the string `smith' OR '1' = '1` as the injection and, surprisingly, I succeeded to logged in as Neville. From this activity, I realized that SQL injection is one of the most dangerous attacks. Imagine all the wicked things (read, modify, and delete sensitive data from database) that I could do after I logged in as an admin.

2) Lab 2: The Great Host/Lexical URL Reputation Bake-off:

In the second lab, I had to write some 'rules' in the Python script to filter out the malicious URLs from the non-malicious (safe) ones. The instructors provided me with two different JSON file, one called 'train' and the other one called 'classify'. Essentially, I had to practice filtering out the malicious URLs in the 'train.json' file first, then apply it to 'classify.json' when I'm sure that my script could run properly. I noticed that the main difference between the two JSON files is in the flag called `malicious_url`. In 'train.json' file, the instructor set this flag value either to 0 (non-malicious) or 1 (malicious). On the other hand, this flag was deactivated on 'classify.json' file. That's why the assignment instructed to filter out this JSON file.

The first thing that I did was to investigate the 'train.json' file. I decided to inspect the characteristic of a malicious URL by creating a simple `if-statement`. The way I analyzed it was to create a conditional (if) whether a particular URL has a `malicious_url` flag value of 0 or 1. If it is 1, then store the URL `domain_age_days` flag in a list called 'maliciousContainer'. However, if it is 0, the store the URL `domain_age_days` flag in another list called 'cleanContainer'. To make it easier to understand, here's a snippet of my testing code:

```
# Empty list for experiment
malcontainer = []
cleancontainer = []

for record in urldata:

  if record["malicious_url"] == 1:

              # print a flag value here
              # I tried almost everything like domain_age_days,
              # file_extension, default_port, etc.

  elif record["malicious_url"] == 0:

      # print a flag value here
              # I tried almost everything like domain_age_days,
              # file_extension, default_port, etc.

print sorted(malcontainer)
print sorted(cleancontainer)
```

I ran the script for several times because I tested different flag such as `domain_age_days`, `file_extension`, etc. Also, noticed that at the end of the `for-loop` I sorted and printed the list to see the result of the filtration. To make this experiment more reliable, I recorded some of the results into a table like this:

Test Result From Train.json (M = Malicious, S = Safe)			
Testing	Lowest Value	Highest Value	Most Common Value
Alexa Rank (M)	130	950889	$>$ 100,000
Alexa Rank (S)	1	92922	$<$ 10,000
Domain Age (M)	449	16198	N/A
Domain Age (S)	1	365	N/A
Domain Name Length (M)	6	185	14
Domain Name Length (S)	6	65	16
Domain Token Length (M)	2	18	3
Domain Token Length (S)	2	7	3
URL Length (M)	16	1249	36
URL Length (S)	14	573	29
Path Token Length (M)	1	17	1
Path Token Length (S)	1	13	1

As can be seen, the most obvious difference between a malicious and a non-malicious URL is in the Alexa rank and the domain age. A clean (safe) URL usually will not reach above 10,000 ranks, and the domain age is older than 365 days (1 year). Meanwhile, a malicious URL usually either has no rank or above 100,000. Their domain age is also relatively small between 1 to 365 days.

Basing on this my data, I decided to create several combinations of these flags, as a rule, to classify the URL records in ‘train.json‘ file. Some of the combination that I tried were checking the URL’s length and Path Token length; then I also attempted to classify based on the domain age with the URL’s length. After a couple of trial and error, I was surprised to know that filtering based on the combination of Alexa rank and domain Age. How did I know that it was the best combo? Simple, I tried it on the ‘train.json‘ file and my script was able to conclude that 49 % of the URLs in the file were malicious. Here’s the code that I ended up using:

```
# Count the valid URLs contained in the JSON file
myCounter = 0

# Empty lists to contain malicious URLs and clean URLs
maliciousContainer = []
cleanContainer = []

for record in urldata:
  # Counter to determine whether a URL is malicious or not
  maliciousPoint = 0

  # Check the URL validity,
  # one of the requirement is to NOT have negative value.
  if int(record["domain_age_days"]) > 0:
    if record["alexa_rank"] > 10000 or record["alexa_rank"] is None:
      maliciousPoint += 1
    if int(record["domain_age_days"]) < 400:
      maliciousPoint += 1

    if maliciousPoint == 2:
      maliciousContainer.append((str(record["url"]), 1))
    else:
      cleanContainer.append((str(record["url"]), 0))

    myCounter += 1
```

Let me explain the algorithm of my script. First, I declared a variable called maliciousPoint that will keep track of the score. Noticed that this variable is inside the `for-loop` so that the value will always be reset to 0 for every new URL taken from the JSON file. Moving on to the rules, I need to make sure that all the domain age days are legit. Thus, I decided to create `if-condition` to eliminate all the URLs that have negative value in their age. Next, my first filter is to check whether a particular URL has an Alexa rank more than 10,000 (based on my research data) or null. If the current URL fulfill one of those criteria, then I increment the maliciousPoint. In my second filter, I checked whether the current URL has an age lower than 400 days (based on my research data)or not. Again, another point will be added to the maliciousPoint if the URL is included in that criteria. Lastly, I made another `if-condition` to check whether the particular URL has two maliciousPoint or not. If the answer is yes, then I append that URL into a list called 'maliciousContainer'. However, if the URL's malicious point is either 0 or 1, then I add it to a different list called 'cleanContainer'. The reason why I locate an URL with maliciousPoint = 1 into a cleanContainer is that I saw one case where the URL has an Alexa rank more than 10,000, but the domain age was 800 days.

To check the accuracy of my rules, I decided to compare print both of my lists. The results are: 1992 valid URLs that are being tested, 960 of them are malicious, and the rest of them are safe. In other words, my rules can prove that 48% of the URL lists are malicious. I believe this was pretty good because the instructor told the students that the correct answer is 50%. To see the detailed result from my test, please check the file named `output.txt`.

II. Conclusion

I learned a lot of new things about web application security this week. First, I learned some new concepts about user-level and browser-level attacks. Furthermore, I acquired a new skill to classify an URL efficiently. Last but not least, I had a chance to play with cross-site scripting and SQL injection attack. I need to admit that web security is as fun as network security.