Security Review Postmortem

This document will outline a Postmortem documentation of our monitoring and pen-testing of the `U buntu?` service, developed by *Group C*, as well as our security review of our own service.

Review of Group C

We started by creating a monitoring tool collecting data about the `U buntu?` service, and then started pen-testing their service. After approximately a week of gathering monitoring data we had enough information to confidently evaluate whether Group C were violating their Service-Level Agreement(SLA). The following order is therefore not a reflection of the order in which the tasks were done, but merely with the intent of readability.

Creating a monitoring tool

To investigate whether or not Group C lives up to their service-level agreement, which had the following promises:

- *Up-time:* 99%
- *Average Response Time:* 200ms
- *Time To Recover:* 2 hours

To validate this we created two scripts, one for collecting data, done by monitoring their service, and one for evaluating the collected data.

We chose to develop these tools in python3.6, as we would be able to run the script from anywhere, while still developing a usable program in a short timeframe.

Monitoring script

The monitoring script continuously appended data to a `csv`-file. This is facilitated by the requests library that helps us send `GET` requests to their webpage. We then logged the response-code from their service. A 200 status code, we log that the webpage is running as intended as well as a timestamp and the elapsed time since the request was sent, if no response was returned or a non-200 was returned, we would mark it as a failure. We pushed this script to our droplet so we would be able to run it without interruption for several days.

The script can be foundhere.

Evaluation script

Our secondary script takes the `csv`-file as input, and then calculates the up-time, average response time, and the time it had taken to recover if any downtime had existed. Running this script at any point in time would reveal this information.

This script is simpler in nature, and can be foundhere.

Evaluating the monitored data

After running the monitoring script for a couple of days, we were able to extract the following results:

Uptime: 100%

Average Response Time: 1.87 seconds

Time to Recover: 0 seconds

We see that the uptime of the webpage and therefore also the recovery time, is flawless and therefore lives up to the SLA. The response time, however, is considerably slower than the 200ms specified by the SLA. We initially feared that it was simply our script that had an overhead, or the network time to our droplet, however after cross-validation with 3rd-party tools such as dotcom-tools.com, we could confirm that their response time was in fact slower than promised.

Therefore we could prove that they had violated their SLA.

Pen testing `U Buntu?`

Information gathering

We used Nmap for information gathering:

Target: www.minitwit.dk

IP address: 142.93.162.43

Ports:

Open port 80/tcp on 142.93.162.43 tcpwrapped (http)

Open port 22/tcp on 142.93.162.43 tcpwrapped (ssh)

Open ports 2, Closed ports 0, Filtered ports 998

Operating System:

2N Helios IP VoIP doorbell (Accuracy 98%) – 1 st try

British Gas GS-Z3 data logger (Accuracy 92%) – 2 nd try

Tcpwrapped results for port scanning and unreliable operating system detection implies that group C is probably using firewall or some other filtering technique, thus we are unable to detect which operating system they are using.

Vulnerability assessment

Unfortunately, we didn’t gather much useful information in the 1st step.

We proceeded with the basic scenario for Web Application Pen Testing. Using techniques such as SQL Injection, Cross Site Scripting and Broken authentication and session management we will check if their application is exposed to any security vulnerabilities.

Exploitations and Results

1. *Cross Site Scripting*

A few basic attempts but nothing worked. Looks like they are sanitizing all the inputs.

2. *SQL Injection*

Since XSS didn't work and inputs are sanitized properly, chances that SQL Injection will were very low but we gave it a try. We didn't have success here doing it manually. Afterwards we used tool called SqlMap in order to see if some of the inputs are injectable. Conclusion is that group C uses Web Application Firewall or Intrusion Prevention System.

```
[CRITICAL] heuristics detected that the target is protected by some kind of WAF/IPS
```

3. *Broken authentication and session management*

- URL rewriting is not possible - Application's timeout is set properly - No predictable login credentials such as admin:admin or admin:12345678

4. *Security Misconfigurations*

* There are no unnecessary ports left open * There are no sufficient services or pages * Default accounts and their passwords are changed (except for the Kibana) * Error handling doesn't reveal any new information about the system

5. *OWASP-ZAP scan* results

```
 X-Frame-Options Header Not Set
Risk: Medium
X-Frame-Options header is not included in the HTTP response to protect
against 'ClickJacking' attacks.
```

```
 Password Autocomplete in Browser
Risk: Low
The AUTOCOMPLETE attribute is not disabled on an HTML FORM/INPUT element containing
password type input.  Passwords may be stored in browsers and retrieved.
```

```
 Web Browser XSS Protection Not Enabled
Risk: Low
Web Browser XSS Protection is not enabled, or is disabled by the configuration of
the 'X-XSS-Protection' HTTP response header on the web server
```

```
 X-Content-Type-Options Header Missing
Risk: Low
The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'.
This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing
on the response body, potentially causing the response body to be interpreted and
displayed as a content type other than the declared content type. Current (early 2014)
and legacy versions of Firefox will use the declared content type (if one is set),
rather than performing MIME-sniffing.
```

6. *Source code inspection*

Since we had access to source code, we investigated a bit further and found an error that could be fatal. `postgres.js` file contains all the information for database we need

```
    user: 'minitwit',
    host: 'db-postgresql-fra1-98386-do-user-3696963-0.db.ondigitalocean.com',
    database: 'minitwit',
    password: 'secret123shhhhhhhhhh',
    port: 25060,
    ssl: true
```

Next, we tried to use default credentials for some of the services and succeeded for Kibana. Afterwards, we also confirmed that by inspecting kibana.yml file.

Security review of own service

First draft

Define Assets

Our assets are:

Source Code: If source code contains any secrets such as passwords and the like. Information how the system is built and so forth.

Access to the Database: If access is gained to the database it is possible to administer the database, giving you full access to the database users and all data contained within the database.

Access to audit data: The audit data shows all auditable events that occurred within the application.

(Do not know if vulnerabilities and threats are required)

Vulnerabilities:

- Open ports

Threats:

- Denial of service
- ClickJacking

Risk Assessment Matrix

	Negligible	Marginal	Critical	Catastrophic
Certain		Denial of service		
Likely				
Possible				DB
Unlikely			ClickJacking	
Rare				

Penetration Testing of our own system

Our approach is to do both a penetration test, and a security audit.

Tools: - Kali Linux - Nmap - SqlMap - Metasploit - OWASP ZAP

Procedure:

In the security audit, we tried to follow OWASP guidelines for developing a secure applicationApplication Security Verification Standard. One of the points is to Verify that secrets, API keys, and passwords are not included in the source code, or online source code repositories. We realized that we are storing several passwords in our source code. The admin password to the database can be found in both our docker-compose file, as well as our sourcecode. Additionally passwords to our admin user is visible in the sourcecode, as well as passwords to our logging tools. This would in theory be acceptable if these can be changed in deployment, and regarding the admin user password, if it is possible to change via the user interface. This, however, is not the case. That is a huge security flaw and must be solved.

The solution to this would be to implement a change-password functionality as well as store all external authorization credentials via environmental contants.

Next is the penetration test in which we have divided into several steps.

- **Step 1** The first step is to make a port scan of the system. We use Nmap to get an overview of Enable OS detection and version detection and open ports.

```
nmap -v -A 46.101.119.181
```

The most worrying finding is the open port at 1433 because it is our database. Also, in continuation of our admin database password being available in source code makes it quite straightforward to access our database.

- Step 2

Firstly, we will try to test if we are vulnerable to Injection flaws, such as SQL injection. SQL injection is the highest security risk according to OWASP and in that case, it makes sense to try that on our system. we used SqlMap to figure out if any of our inputs are injectable. If not, then SqlMap will return - *[WARNING] heuristic (basic) test shows that POST parameter 'username' might not be injectable* That is, our system is not vulnerable regarding SQL injection. All tested parameters do not appear to be injectable.

- Step 3 Information about the DB.

Because the Database port is open it is possible to acquire information about the database. We will use a tool from Metasploit that will enumerate MSSQL configuration setting.

In essence, the module will perform a series of configuration audits and security checks against our Microsoft SQL Server database.

The password is required for the module to work and because the password is accessible in our source code it is not difficult to acquire.

Using auxiliary/admin/mssql/mssql_sql will allow for simple SQL statements to be executed against an MSSQL instance given appropriate credentials. That is, if someone knows our password then they had all they did need to delete our database. Also, because we have stated it would be catastrophic in our risk matrix if something happens to our database.

- Step 4

We have also tried to use another tool called OWASP ZAP.

It is an open-source web application security scanner and its main goal is to find vulnerabilities in web applications. It will run different attack scenarios against the web application and record the results.

The results that we got from OWASP ZAP were:

```
 X-Frame-Options Header Not Set

Risk: Medium

X-Frame-Options header is not included in the HTTP response to protect against 'ClickJac
```

and

```
X-Content-Type-Options Header Missing

Risk: Low

The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allo
potentially causing the response body to be interpreted
```

A comment on the first result is that it can be solved by setting the X-Frame-Options HTTP header and ensuring it is set on all web pages returned by our web application.

A comment on the second result is that it can be solved by ensuring that the web application sets the Content-Type header appropriately and that it sets the X-Content-Type-Options. Ensure that the application/web server sets the Content-Type header appropriately and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages. Overall, OWASP ZAP did not find any **critical** vulnerabilities and both vulnerabilities that OWASP ZAP reported can be solved with few lines of code in our application. It is all about setting headers correctly concerning security.