CS915 Advanced Computer Security, Final Project

UID: 1690550

March 14, 2017

1 Part 1

1.1 Cross-site scripting

Cross-site scripting (XSS) occurs when a user injects a client-side code into a web page, through a field taking user input. This code is then executed in another's client browser (as is the case with DOM-based and reflected attacks) or on the server side (stored attacks). It occurs when the website fails to validate user's input. It is considered to be of high, although not critical severity [1]. It is the most common type of website attacks, which has been around since 1990s [2].

The aim of XSS attacks is most often to access a client's usernames and passwords, and other data such as banking information, which is often stored in the browser.

A real-life example of an XSS attack is a malicious script known as "Samy Worm", which in 2005 propagated across MySpace, making over 1 million users execute the payload within just 20 h. The development of the worm and the details of the implementation were described by the author himself after MySpace has fixed its issues, and is still available online: `http://samy.pl/popular/tech.html`.

In the case of our website, we can perform an XSS attack on the Guestbook page. Posting a comment with the content as follows:

```
<script>alert("Hello!")</script>
```

Will cause this comment to be saved to the database as is and inserted into the document whenever anyone opens the page. It is a valid DOM element, and it will simply cause a pop-up with a message that the author of the website did not intend to be there. Note the double quotation marks (so that they do not get mixed up with the single ones in the SQL statement) and the lack of semicolon, which makes it a valid input.

This vulnerability is found whenever we take user input and post it to the database, which is then displayed for other users.

To prevent this sort of attacks, we need to strip user input of tags. For example, PHP function `strip_tags()` is capable of that. So instead of `$row['name']` and `$row['comment]` we should have `strip_tags($row['name'])` and `[strip_tags($row['comment'].`

1.2 Directory traversal attacks

Directory traversal attacks aim to access the parent directory and traverse the directory tree on the server, by providing user input which includes multiple `../` that traverse back to the root directory. Note that repeating `../` more times than necessary after reaching the root directory does not have any effect.

It can be used to read or execute files stored on the server. The most common PoC of directory traversal is accessing `/etc/passwd`, where the passwords are stored on a Unix system. We can do it on the website by going to the following address:

`127.0.0.1:4567/cats.php?page=/etc/passwd`

We can also some files that we don't see on the website, for example:

`127.0.01:4567/cgi-bin/basic.cgi?file=basic.cgi`

A way to prevent it is to strip '../' or '%2e%2e%2f' or any combination of them and do not store any files that are not necessary for the website in the website's directory.

1.3 SQL code injection

SQL code injection - as the name suggests - relies on inserting SQL code devised by the attacker into the server's database. The code may involve for example dropping the tables. It is another example of an attack which can be prevented by sanitizing user input correctly.

Some of the most severe cases of SQL injection attacks involve an American hacker Albert Gonzalez, who in 2005-2007 used this technique on corporate systems to steal and later re-sell over 1.5 million credit card details [3].

This can be done easily on the guestbook page by adding the comment:

`first comment'),(now(), 'Eve', 'second comment')`

It results in adding two comments instead of one.

There are many methods of preventing SQL injections. It can be done by adjusting priviliges, or by simply sanitizing input, e.g. to escape all characters such as quotes, '%' or '*'.

1.4 Python code injection

This time we are interested in injecting Python script into the website. It is a much less well known attack than for example SQL injection, but nevertheless can be harmful. [**?**]

In our website, the function plotter is written in Python and at line 30 of the file 'domain.cgi' we see the following statement:

`v = eval( function )`

Where `function` is unsanitized user input. Now if we input a function such as:

`__import__('os').system('cp basic.cgi injected_copy.cgi')`

Python's `eval()` function is a known vulnerability, and we should avoid it whenever possible. Unfortunately, this is not easy in this case, so we have to make sure it can evaluate safely by using additional parameters. We need to define a dctionary of safe function, for example `safe_dict = {"x" : x, "y" : y, "tan" : tan}`, and then change line 30 of 'domain.cgi' to:

`v = eval(function, "__builtins__":None, safe_dict)`

1.5 Stack-based buffer overflow

C is especially susceptible to stack-based buffer overflows. These types of attack rely on no bound-checking, thus allowing an attacker to exceed buffer size. It usually involves overwriting the memory that was not intended to be accessed, but can also mean reading it, thus revealing sensitive information.

The best-known example of buffer overflow attacks is the Heartbleed bug present in the OpenSSL library. It was exploited in 2014, when hundreds of social insurance numbers were stolen from the Canada Revenue Agency [4].

Note that in the RPN calculator, we are allocated a buffer of size 2048 for terms in the calculation. Therefore we can exceed the buffer size by writing '2 ' 1026 times followed by '- ' 1025 times, which causes stack overflow.

To prevent this, we can change the condition of the while loop in the main function of the RPN to be `while (pptr != NULL && !finished && stack_top < MAX_TERMS)`

1.6 Heap-based buffer overflow

Heap-based buffer overflow differs from a stack-based one in that it relies on dynamic memory allocation, and usually aims to corrupt the application's data, as reading or precise overwriting of data is much more difficult than in a stack overflow.

One of the more severe cases of such an attack was exploiting a bug in Internet Explorer's XML parser in 2014. It allowed for execution of an attacker's code in the browser. It is not the first issue of this kind with Internet Explorer. Another similar one has been discovered in December 2008 in IE7 [6].

1.7 Format string vulnerability

In C, printing formatted strings involves format specifiers, for example `%s, %d`. A format string attack occurs when a user-submitted string contains such specifiers. It may allow the attacker to read the content of the stack or perform a denial of service attack.

A notable case of an exploit of this vulnerability occurred in 2000, when hackers gained root access to a host running an FTP daemon [7].

In our website this vulnerability is present in the RPN calculator. More precisely, the `interpret()` function is very lenient about comments. As soon as it sees '//', it does not read past it into the content of the comment. It simply returns the value of that input line as a string. That string is then used by `sprintf(...)`, which accepts format specifiers. Thus we can read the stack by adding them.

For example, changing the first line of the default content of the calculator (the one we see when we first visit the page) to:

```
// RPN Calculator Syntax (This is a comment)%s %f %s %f
```

Results in a successful compilation and the following output:

```
[0.000000] // RPN Calculator Syntax (This is a comment)[%f] %s 0.000000 //
RPN Calculator Syntax (This is a comment)%s %f %s %f 0.000000
```

(I suppose on the bright side we did get our original comment out at some point after all...)

To prevent this attack, we need to inspect comment strings closer and strip it of format string specifiers. There are various ways to do that, for example replacing each '%' with '% '- i.e. '%'followed by a white space.

2 Part 2

First to prove that we can create and remove files from another machine, we present the following script

References

[1] I. Hydara et al., "Current state of research on cross-site scripting (XSS) - A systematic literature review" in *Information and Software Technology*, vol. 58, 2015, pp 170-186

[2] B. Gupta et al., "Cross-Site Scripting (XSS) Abuse and Defense: Exploitation on Several Testing Bed Environments and Its Defense" in *Journal on Information Privacy and Security*, vol. 11, 2015

[3] S. Gaudin, "Government informant is called kingpin of largest U.S. data breaches" in *ComputerWorld*, available:
`http://www.computerworld.com/article/2527161/government-it/government-informant-is-called-kingpin-of-largest-u-s--data-breaches.html`

[4] P. Evans, "Heartbleed bug: RCMP asked Revenue Canada to delay news of SIN thefts", CBC News, April 2014, available: http://www.cbc.ca/news/business/heartbleed-bug-rcmp-asked-revenue-canada-to-delay-news-of-sin-thefts-1.2609192

[5] "Microsoft Internet Explorer Heap Buffer Overflow Remote Code Execution Vulnerability", Zero-Day Initiative, April 2014, available: http://www.zerodayinitiative.com/advisories/ZDI-14-034/

[6] K. Dubey "Microsoft announces emergency patch to fix Internet Explorer", *Techshout.com*, Dec. 2008, available: http://www.techshout.com/internet/2008/18/microsoft-announces-emergency-patch-to-fix-internet-explorer/

[**?**] K. London, "Dangerous Python Functions, Part 1" in *Kevin London's blog*, 26 June 2015, available: *https://www.kevinlondon.com/2015/07/26/dangerous-python-functions.html*

[7] AJ Kumar, "Format String Bug Exploration", InfoSec Institute, May 2015, available: http://resources.infosecinstitute.com/format-string-bug-exploration/#gref