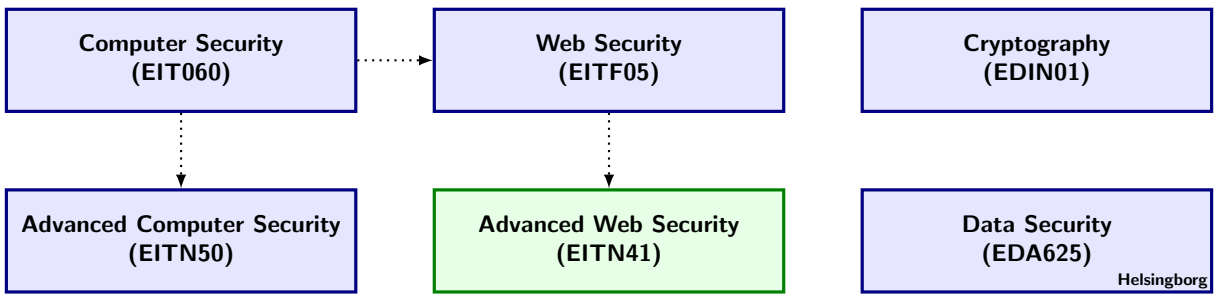
Advanced Web Security 2016

Department of Electrical and Information Technology
Lund University

Web Server Security: Attacking and Defending

Learning goals:

- Attack a web server using SQL injections.
- Get sensitive information using XSS attacks.
- Defend against attacks using ModSecurity.



Read this earlier than one day before the lab!

Note that you will not have any internet access during the lab, so come prepared. You may bring as many books and printed materials as you can carry. You may bring your computer although you may not be able to get a good wifi signal.

There are preparatory assignments for this lab, **write answers on paper, you will have to show your answers to be allowed to do the laboratory**. For most students, these assignments take more than a couple of minutes. Read through this lab guide carefully (Yes, the complete paper), and then prepare your assignments. During the lab, answer **all problems** on a separate sheet of paper, so your work can be approved.

IMPORTANT! The lab computers are completely shielded from the internet. As a consequence, any information you believe will be needed during the lab, has to be written down/printed out beforehand. Study the questions in this lab manual, consider what you will need to be able to solve them, and make sure you bring that information with you.

Introduction and Goal

In this laboratory lesson you will be provided with a webserver and a poor implementation of a webshop. Your objective is twofold: to attack the webserver in various ways, and to implement security measures to thwart your previous attacks. The attacks you will perform include SQL injections and XSS to steal cookies. As defense mechanisms you will write rules for Apaches web application firewall ModSecurity. Part of the lab is also to configure ModSecurity properly. To make sure you are well prepared for the lab you should be familiar with the concepts of SQL injections and XSS, especially persistent XSS as opposed to reflected XSS. The basics of ModSecurity should also be familiar to you. In the appendix, there is a short introduction and one of the preporatory assignments links you to a free beginner's manual.

1 Attack

The attack should be performed in the following way

- Use SQL injection vulnerabilities to gain administrative access to the website
- Use a persistent XSS vulnerability to inject a script that will steal another user's cookie
- Use the stolen cookie to assume the identity of the attacked user

1.1 Reconnaissance

In real life attacks, reconnaissance is always the first step. Reconnaissance includes gathering as much data about the target as possible and may involve activities like port scanning and vulnerability scanning. In this lab you will perform a lot of manual vulnerability scanning in order to find SQL injection and XSS vulnerabilities.

Start doing some basic reconnaissance on the website. You gain access to the website by using **localhost** as the hostname in the browser. You will not have access to any PHP code or the MySQL database, so you can see this as a black box scenario.

Visit the different pages that exist, sign up, log in, add items to cart, etc.

A vulnerability doesnt necessarily mean that there is only one way to take advantage of it, or any way for that matter. There can be multiple attack vectors on a single vulnerability. For instance, a specific SQL injection vulnerability may be used to both dump the contents of a database as well as delete them.

1.1.1 SQL injections

Preparatory assignment 1

- Consider the following SQL statement:

  ```
  SELECT * FROM users WHERE username="<username>" AND password="<password>"
  ```

  Assume that you have full control over the input in <username>and <password>. How would you go about testing if this statement is vulnerable to SQL injections and what do you think you can accomplish if it is vulnerable?

  Hint: What input would make the statement an erroneous SQL query?

Problem 1

- *Can you find any SQL injection vulnerabilities on the site? How do you test for them?*
- *Write down all the SQL injection vulnerabilities you can find.*
- *What are some possible attack vectors on the SQL injections you found?*

To achieve the goal of this lab you are going to need administrative access to the website. The first step in order to achieve this is to find the username of the administrator's account. This can be done in many ways. One way is to try and sign up with usernames common to be associated with the administrator and see if they exist or not. That approach will not work in this lab.

1.1.2 Unauthorized access

Another common approach is to find discrepancies in the error output when trying to log in. For instance, if an account exists, but the password is incorrect, the error message could be "Incorrect password", whereas if the username doesn't exist the error message could be "No such user". It is important to not reveal any information like this, therefor an appropriate message for both cases should be displayed for the user, like "Incorrect username or password".

You should now try to enumerate the administrator's username with one of the SQL injection vulnerabilities you have discovered. Use the table below and include one TRUE statement (AND 1=1) and one false statement(AND 1=2) for each username and make a note of the error messages. When the first two rows are filled, you should be able to see what error messages an existing and a non-existing account generates.

Problem 2

- *Fill in the table with the different error messages for each case.*

  What is the administrator's username? Do your results actually verify that it is the administrator's username?

  Can you say something about why the error messages differs when a username exists compared to when a username doesn't exist?

	TRUE statement	FALSE statement
<your username>		
<non-existing username>		
root		
administrator		
admin		
superuser		

One common way of gaining access to an account, for which the password is unknown to you, is to insert the following into the username field or the login form:

```
username" AND 1=1#
```

This will authenticate *username* and skip the password check since 1=1 is always true. The 1=1 part may not even be necessary. It all depends on the implementation.

Problem 3

- *Try the above method to authenticate. Does it work? Why or why not?*
- *Can you find the administrator's password in some way?*
- *Is it possible to gain access to the administrator's account without knowing the password?*

The following snippet of code is part of the page that performs a password change. First it makes sure that the old password is correct and then it updates the database with the new password.

```
<?
    $query = 'SELECT password FROM users WHERE username="' . $_SESSION['username'] . '"';

    $result = mysql_query($query) or die("query: " . $query . " failed");
    $row = mysql_fetch_assoc($result);
    if($row['password'] != $_GET['old']) {
        echo "Incorrect password";
        exit;
    }

    $query = 'UPDATE users SET password="' . $_GET['new'] . '" WHERE username="' . $_SESSION['username'] . '"';

    mysql_query($query) or die("query: " . $query . " failed");
    echo "Password changed successfully";
?>
```

Preparatory assignment 2

- Make sure you understand what the code displayed above does.

Problem 4

- *Insert data in the change password form that would make the UPDATE query fail. When you succeed, the query will be displayed for you*
- *Create a SQL injection that changes the password of the administrator's account instead of your own.*

Once you are able to authenticate as the administrator you are finished with this part of the lab.

1.2 Cookie stealing with Cross Site Scripting

With access to the administrator's account you have abilities that no other user has.

Problem 5

- *What abilities are these? See what you can do with your normally registered account and compare.*
- *Can you (ab)use these to your advantage?*

Preparatory assignment 3

- Make sure you understand the basics of XSS attacks. Especially persistent XSS attacks.
- Write two javascripts. One which is appropriate for testing if a XSS vulnerability exist and one that can be used to steal a user's cookie. Test the scripts and make sure they work. For the cookie-stealing script, assume that you have control over another webserver that can read GET parameters.

The cookie stealing script you wrote in the preparatory assignment should now be used so that all users who are logged on to the site discloses their cookie for you.

You have a local webserver that is setup and under your control. The hostname for this host is **localhost2**. It is a virtual host and is part of the webserver as the site you are attacking. The index page of localhost2 will look for a GET parameter called cookie and save it's value to a file, /var/www/localhost2/log.txt. You can cat this file to show its contents.

Problem 6

- *Find a persistent XSS vulnerability on localhost. Use the first script to test your assumptions.*
- *Modify your cookie-stealing script so that it works appropriately in this context.*
- *How does the script work? Can the script be improved in any way?*
- *Is your exploit stealthy? Can the user who is subject to your attack notice anything out of the ordinary happening?*
- *Hint: If you are having problems injecting your script you can always right click on the website and click 'View Page Source' to see how your script was embedded on the page.*

A script for stealing cookies can be formed in many ways. In you home directory you will find three files, cookie_stealer{1,2,3}.js.

Problem 7

- *Look at the code in all files and try to understand how they work.*
- *Use each of the scripts in the XSS vulnerability on the site and try them out.*
- *Which one works best? Can you think of some circumstances where each script is the best alternative?*

Once you have successfully been able to steal a cookie you can move on to the next part.

1.3 Commit identity theft

Extract the stolen cookie from log.txt and have it ready.

To assume the new identity we need to assign our PHPSESSID cookie variable with the value of the stolen cookie. First of all, make sure you are logged in with the user you stole the cookie from. Close Firefox (without logging out!) and reopen it. Now you are not logged in as that user, and

the cookie has not been destroyed on the webserver, so it is still usable. Log on again with some other user (NOT the one you hacked).

> **Preparatory assignment 4**
>
> - Gain some basic knowledge of how the Firefox addon FireBug works.
> - Find out how to edit cookies with the help of FireBug.

The cookie is stored in your home directory under /.mozilla/default/cookies.sqlite. This means that we technically can use SQL queries to edit the cookies directly. FireBug, however, provides an easier and more flexible interface of doing this.

> **Problem 8**
>
> - *Edit the cookie to become the user you hacked. Use FireBug for this.*
> - *Can you browse the website as that user unhindered? What would happen if the real user is logged on and browsed the website at the same time as you?*

You have now completed your attack. From reconnaissance to exploit crafting to exploitation, these are common steps involved in real life attacks on web applications. To really understand how to protect yourself against these types of attacks, it is critical that you understand how to execute them. **It goes without saying that any unauthorized attempts to perform these techniques on other websites may very well be illegal and can get you into trouble.**

The attacking part of the lab is now finished, and you will move on to defending it against these attacks.

2 Defend

Now you will take on the role as the security administrator of the website and defend yourself against the vulnerabilities you previously exploited. The main tool you will use is ModSecurity. ModSecurity is an open source web application firewall for the Apache webserver. With it you can write rules in many different ways to satisfy your needs.

2.1 ModSecurity

> **Preparatory assignment 5**
>
> - Read up on the basics of ModSecurity. Important topics include configuration and how rules are written. The appendix provides what is necessary for this lab. The following link points to a free ModSecurity handbook and provides good documentation on setting up and configuring the firewall. Make sure you glance through this as well.
>
> **https://www.feistyduck.com/books/modsecurity-handbook/modsecurity-handbook-getting-started-may-2012.pdf**

Enable ModSecurity by adding and editing the proper directives in its configuration file. The configuration file can be found at /etc/apache2/conf.d/modsec.conf.

For a rule to be loaded by Apache, you need to restart the server. So whenever you want to try out a newly written rule, issue the following command:

```
service apache2 restart
```

If there is an error in the config file the server will fail to start.

Start focusing on stopping the XSS attack.

Problem 9

- *Create a rule that stops the XSS attack with the script from cookie_stealer2.js.*
- *Will the rule stop an XSS attack with your own script or the cookie_stealer3.js script? If not, write a rule that stops these as well.*

Sometimes we want to guard ourselves from attackers that may have an ace up their sleeve, i.e, they may know of some attack that we are not aware of, or which we for some reason are not protecting ourselves against. During the course of the attacker's reconnaisannce he accidently trigger's one of our rules. In this case, by denying the IP-address of the attacker access we could slow him down and raise our guard against a possible attack.

Problem 10

- *Create a rule that stops the IP-address of a user that tries to attack our website. It is up to you what the rule should trigger on. Consider that it may not be wise to trigger on something trivial that the user could enter as a mistake.*
- *The script /var/www/block.pl will block the IP address of the offending party. The script reads the attackers IP address through the USER_IP environment variable.*
- *Add a rule that allows you, the administrator, to perform attacks without adding your IP to the firewall.*
- *Ask a neighbour to attack your website and see if your rules are functioning properly.*

So far we only stop injection attempts on the database. If, however, the attacker has already managed to inject a script into the database before we utilized ModSecurity, or through some other means, a user requesting the infected page will not be protected with the rules we have written, i.e, XSS is still possible, it is just the injection of the script that has become troublesome to the attacker.

Problem 11

- *Write a rule that checks outgoing HTTP responses and verify that it stops the attack.*
- *Assume that the website filters input and output properly, e.g, in the PHP code. Is it a good idea having ModSecurity rules that check the same, or is the redundancy unnecessary?*

The SQL-injection flaws still exist on the website.

Problem 12

- *Write a rule that protects you from SQL-injections.*
- *Assume that the website is using the mysql_real_escape_string functions and/or prepared statements to break SQL injection attempts. Is it a good idea, or even necessary, to also have rules in ModSecurity that stops SQL injections?*

You are now finished with the lab. Report your answers to the lab assistant.

A ModSecurity

The main configuration file for ModSecurity can be found in */etc/apache2/conf.d/modsec.conf*. In the IfModule section you can insert a couple of directives.

SecRuleEngine specifies whether ModSecurity is enabled or disabled. Possible settings are on, off and DetectionOnly. DetectionOnly will only log when rules are triggered, but it will not stop them. This could be good for debugging purposes or if your rules are very aggressive and dont want to trigger on false positives.

SecDefaultAction specifies what the default action is when a rule is triggered. The following value will deny the offending user, log him and provide a status 403 response when a rule is triggered.

```
"phase:2,deny,log,status:403"
```

phase:2 means that the action should be triggered after the request body has been read. phase:1 would trigger the action after the request headers has been read by apache and phase:5 would never trigger. It is important to specify the correct phase in a rule depending on what it analyzes.

SecRequestBodyAccess is needed for ModSecurity to get access to the HTTP request body which includes, for instance, POST variables.

SecResponseBodyAccess is needed for ModSecurity to get access to the HTTP response body.

A rule has the following anatomy:

```
SecRule Target Operator [Actions]
```

The **target** specifies what part of the request, or response, that should be analyzed. Possible values include REQUEST_URI, REQUEST_PROTOCOL and REMOTE_ADDRESS. A special variable, called a collection, consists of several values and can also be specified as a target. ARGS is such a collection which contains all GET and POST values.

Operator specifies what to look for in the target. As default it is a regular expression.

The **action** part is optional, but could be used if a certain rule should deviate from the default action which was specified with SecDefaultAction.

The following rule will deny access and not do any logging when any part of the URI includes the phrase **passwd**.

```
SecRule REQUEST_URI "passwd" "deny,nolog"
```

An action can also include the execution of another application. If a rule is triggered and we want to send an email to the administrator we could specify a rule in the following way.

```
SecRule REQUEST\_URI "passwd" "deny,exec:send_email.out"
```

This rule executes the *send_email.out* program when the rule is triggered. Sometimes we may want to pass more data to the program to be executed. The data can also be included in the actions like this:

```
SecRule REQUEST_URI "passwd" "deny,setenv:URI=%{REQUEST_URI},exec:send_email.out"
```

The entire URI is now included as the environment variable URI which can be used in the script *send_email.out*.