KerneronSecurity

Saturday, October 24, 2015

Timing attack vulnerability in Zeus server-sides

Timing attacks has proven practical since 96' as shown in a paper by Paul C. Kocher. In his paper Paul demonstrate how, by effectively measuring the amount of time required for private key operation, one could completely uncover the private key. This attack was shown to be effective against widely known crypto-systems such as Diffie-Hellman, RSA and DSS.

Almost ten years later on 2004, another research paper was published by Dan Boneh and David Brumley, entitled "Remote Timing Attacks are Practical" claiming that timing attack as shown in Paul C. Kocher paper are also practical remotely. Their research shows a successful attack against a remote instance of Apache server using OpenSSL running on local network.

Then, in Crosby paper and also in Daniel Mayer & Joel Sandin paper they documented an extensive bench-marking work to determine what is actually the smallest processing time frame that can be measured across the different hardware and networking setups.

Now, to tell you the truth, I didn't know a thing about these publications or much of the existence of timing attacks when I found this vulnerability in Zeus botnet's server-side about three years ago. Even though i didn't use much of the mentioned knowledge in my research, I decided to give this intro for people who would like to expand their knowledge about these attacks.

The vulnerability I've discovered is basically a timing attack which enable a remote attacker to resolve the length in characters of the reports directory name by carefully measuring the response time of the server. While this vulnerability maybe considered as low risk, as well as found on fraudulent piece of software, I find its nature to be a very interesting and intriguing case-study which could be of a good use for future researchers.

What can you do with it?

Using this vulnerability, against Zeus server-side, might increase our chances of finding the reports directory. By knowing the exact length of the directory, we could launch a length-specific brute-force attack. If we were lucky in finding the reports directory we might be able to proceed to one of the following scenarios -

1. Access a shell script uploaded to the server. and take over!
2. It might enable us to harvest the content of this directory in cases of open directory listings

and of course there is always the chance that you can do absolutely nothing ;)

But from my experience I've had a lot of success exploiting this vulnerability in the search for the report folder.

The vulnerability

I shall focus only in the vulnerable section of the gate.php. This part is where the server accepts file uploads from the bot. So it begins where the server checks if the type of the request from the bot is of an uploaded file -

```
1  $type = toInt($list[SBCID_BOTLOG_TYPE]);
2
3  if($type == BLT_FILE)
4  {
```

Next, the server construct the local report folder path adding to the path the "files/" directory and the botname + botid.

```
5  //Расширения, которые представляют возможность удаленного запуска.
6  $bad_exts = array('.php3', '.php4', '.php5', '.php', '.asp', '.aspx', '.exe', '.pl', '.cgi', '.cmd'
7  $fd_hash  = 0;
8  $fd_size  = strlen($list[SBCID_BOTLOG]);
9
10 //Формируем имя файла.
11 if(isHackNameForPath($botId) || isHackNameForPath($botnet))die();
12 $file_root = $config['reports_path'].'/files/'.urlencode($botnet).'/'.urlencode($botId);
13 $file_path = $file_root;
14 $last_name = '';
```

so it may result in something like - "_reportz/files/VICTIM/VICTIM-PC/"
The next code section, it merges the above mentioned path with the remote path of the file -

```
15    $l = explode('/', (isset($list[SBCID_PATH_DEST]) &&
16 strlen($list[SBCID_PATH_DEST]) > 0 ? str_replace('\\', '/', $list[SBCID_PATH_DEST]) : 'unknown'));
17    foreach($l as &$k)
```

Pages

- Home
- About me

Blog Archive

- ▶ 2016 (1)
- ▼ 2015 (2)
  - ▼ October (2)
    - Timing attack vulnerability in Zeus server-sides
    - A Walkthrough of the "APT" Intelligence Gathering ...
- ▶ 2014 (1)
- ▶ 2012 (1)
- ▶ 2006 (1)

Labels

0Day APT Backoff C&C CCTV cipher Citadel cryptography decipher Embed Exploit GameOver Human Factor Intelligence Licat Malware Old Stuff OSINT Perl PoS RCE Retail Timing Attack tool Vulnerability Web XSS Zeus ZeusVM

```
    {
      if(isHackNameForPath($k))die();
      $file_path .= '/'.($last_name = urlencode($k));
    }
    if(strlen($last_name) === 0)$file_path .= '/unknown.dat';
    unset($l);

    //Проверяем расширении, и указываем маску файла.
    if(($ext = strrchr($last_name, '.')) === false || in_array(strtolower($ext), $bad_exts) !== fals
    $ext_pos = strrpos($file_path, '.');
```

So this next line is actually a core part of the this vulnerable code, the server now checks the length of the merged paths to see if its 180 chars or bigger, and if so it renames the remote path to `longname.dat`

```
//FIXME: Если имя слишком большое.
if(strlen($file_path) > 180)$file_path = $file_root.'/longname.dat';
```

Next, we enter a loop which iterate 9999 times at most. Each iteration it perform three checks to determine if the file uploaded already exists on the server. if this is not the first iteration then alter the original file-name adding parenthesis containing the iteration number. This could result in somthing like `_reportz/files/VICTIM/VICTIM-PC/filename(1).ext`

```
//Добавляем файл.
for($i = 0; $i < 9999; $i++)
{
  if($i == 0)$f = $file_path;
  else $f = substr_replace($file_path, '('.$i.').', $ext_pos, 1);
```

if the file exists by name then, perfrom some more checks. if not, then save it.

```
if(file_exists($f))
{
```

checks if they share the same size by any chance

```
if($fd_size == filesize($f))
{
```

do they have the same MD5 ?

```
    if($fd_hash === 0)$fd_hash = md5($list[SBCID_BOTLOG], true);
    if(strcmp(md5_file($f, true), $fd_hash) === 0)break;
  }
}
else
{
  if(!createDir(dirname($file_path)) || !($h = fopen($f, 'wb')))die();
```

save the file!

```
      flock($h, LOCK_EX);
      fwrite($h, $list[SBCID_BOTLOG]);
      flock($h, LOCK_UN);
      fclose($h);

      break;
    }
  }
}
```

Sweet! now after we've went through all the relevant logic, its time to understand whats so vulnerable in this pile of code. To explain this vulnerability lets assume the next scenario; Lets say we've submitted about a twenty files with path bigger than 180 characters, size of 10KB and containing random data. if we check the serverside folder containing our file it will look something like this -

```
exodus@x /var/www/zeus/_reportz/files/VICTIM/VICTIM-PC/ $ ls -lash
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname.dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(1).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(2).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(3).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(4).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(5).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(6).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(7).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(8).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(9).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(10).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(11).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(12).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(13).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(14).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(15).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(16).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(17).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(18).dat
 12K -rw-r--r-- 1 www-data www-data 9.8K Aug 17 16:35 longname(19).dat

 ...
```

The next time we send a file like that, the server is gonna have to compare it against all the other existing longname(x).dat. first by name, then size and finally the most time consuming process of it all is the MD5 comparison. Since the files content is random, their MD5 will always differ. The loop will continue until its 21th iteration where the file longname(20).dat wont exist so it will create it and exit. This condition creates a time consuming loophole that result in a significant delay in the server response time and will get worse with every file we add.

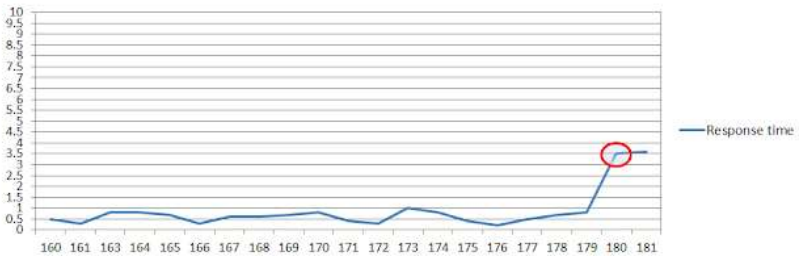

Figure 1: In theory

Now here is the catch - if we are able to spot this spike in response time, as shown in Figure 1, knowing how long our path was, we should be able to the determine the length of the reports directory.

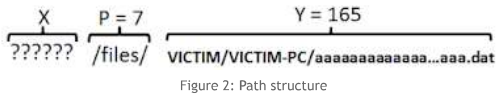

Figure 2: Path structure

As shown in the figure above, we are after ***X*** which is the length of the report directory. subtracting it by ***P*** which is the constant sub-directory and finally, subtracting again with our input path represented by ***Y*** this should result in the following -

$$180 - P - Y = X$$

$$180 - 7 - 165 = 8$$

figure 3: Reports length calculation

So in this example the directory length is 8. Awesome! so we understand the vulnerability. But to produce a reliable exploit, there are more difficulties we have to overcome.

The exploitation

Measuring the server response time is getting trickier due to natural inconsistency of the internet speed.

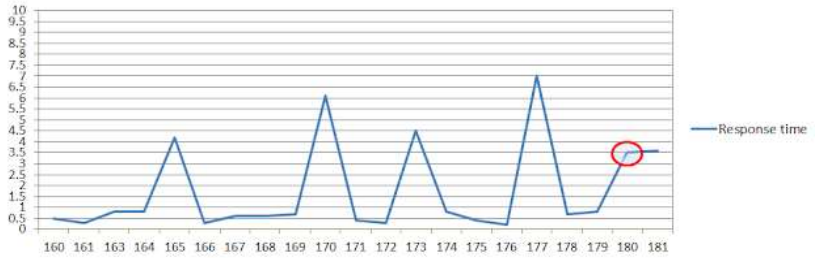

Figure 4: The internet reality

So we need to develop a reliable strategy that would enable us to spot that specific time difference we are after and then do the math.

We need a way to get rid of the random noise also known as Jitter. The way i approached this problem, is to collect response time samples of two separate groups.
The first group contain response times of files sent to the server with path length bigger than 180 bytes, we will refer to it as ***L group***. The second group contain response time of files sent with short path that wont exceed 180 bytes for sure, and we will refer to it as ***S group***.

So I sample those two groups in rounds of hundred iteration, at the end of each round i performed my Jitter exclusion calculation which is the following condition -

$$X_L > \overline{L} + \sigma_L$$

Figure 5: Jitter detection condition

This basically means, if ***X***, which is a time sample of group ***L***, is bigger than the sum of its average plus standard divination - exclude it!

Then i recalculate the average and move on to the observable interval test. To see if we have a sufficient interval to differ between the groups. The calculation i used for this was pretty simple as well - if the average of the ***S*** group plus 50% of its size is bigger than the average of the ***L*** group then we have a sufficient interval and we can move to the final stage of the exploit.

Note that with every submission of file for the ***L*** group, the servers processing time will be slightly longer, since after every submission there is one more files in the stack.

Once the exploit detect that it has a sufficient interval between the groups average, it moves to the final stage where we submit files with path of 180 bytes and decreasing its size by one each time. The first time we encounter a decrease in response time in which the response time was closer to the ***S group*** average

$$\overline{S} - T < \overline{L} - T$$

Figure 6: Closer to ***group S***

-is our sign for knowing that the total path length is now 180 characters and we can do the calculation as shown in figure 3.

The vulnerability survival journey

Now that we are through with all the technical stuff, we can move on to some fun stuff. One of the things i like about this vulnerability is its stealthy nature. It enable it to survive throughout almost all Zeus development cycles. So for this i prepared the following Zeus evolution graph that demonstrate the vulnerability survival journey

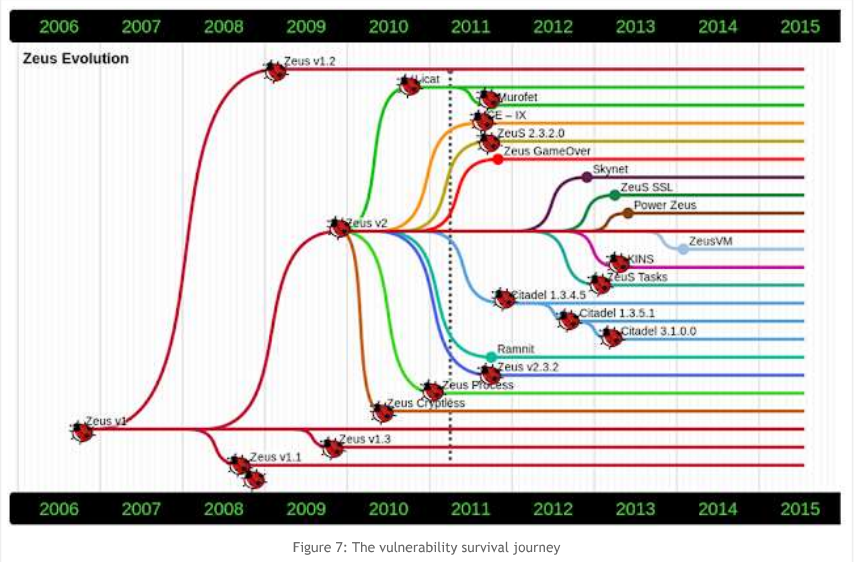

Figure 7: The vulnerability survival journey

Some of the variants on this map I haven't had the chance to check if they are vulnerable, but my guess that they are.

So i think all that's left to do is leave you here with the exploit code, and let you have fun with it.
The exploit, by the way, is written in PHP using Zend framework just because i was too lazy to convert it to python.

The exploit also contain a nice Zeus client library i wrote, which enables it to communicate with most Zeus variants, as long as you have the right encryption key.

Here is a the repository for the exploit on Github -
https://github.com/k1p0d/zeus_reports_len

You are all very welcome to comment here on the blog, or contact me by mail and i'll get back at you ASAP.

Peace :)

Posted by Exodus at 9:15 AM

Labels: 0Day, Citadel, Exploit, GameOver, Licat, Malware, Timing Attack, Vulnerability, Zeus, ZeusVM

13 comments:

Hai October 24, 2015 at 2:48 PM

我喜歡用三明治挺舉關閉

Reply

Exodus October 24, 2015 at 3:22 PM

Right this is exactly what I've been telling you all this time man... Just put some chilli on that sandwich you'd love it!

Reply