David Scholberg
Master's Project Paper
12/7/2012

An Intrusion Prevention System to Mitigate the Apache Killer Vulnerability

Introduction

Versions 2.0.x through 2.0.64 and 2.2.x through 2.2.19 of the popular Apache web server contain a vulnerability (dubbed "The Apache Killer") involving the processing of HTTP Range headers that can lead to a Denial of Service (DoS) attack[1]. In an HTTP request, the HTTP Range header may contain multiple ranges that tell the HTTP server to only return those parts of the requested resource that are specified by the ranges, and those ranges may overlap each other[2]. When a vulnerable version of Apache attempts to process an HTTP Range header containing too many overlapping ranges, it starts consuming an excessive amount of CPU and RAM and effectively prevents legitimate client requests from being processed.

The goal of this project is to create an Intrusion Prevention System (IPS) that can detect, block, and log any attempt at exploiting the Apache Killer vulnerability while still allowing legitimate client requests to pass through. The approach taken here is to place a proxy server in front of Apache that inspects all incoming HTTP headers before they get sent to Apache. If a given HTTP header has more than certain number of ranges in the Range section, the proxy server does not send the HTTP header to Apache, and the IP address of the offending client is logged.

In the case of this vulnerability, an official patch that fixed the vulnerability was released very shortly after the vulnerability was disclosed[3], and there were several unofficial patches and workarounds available even sooner[4]. Using an official patch from the maintainer would generally be the best way to mitigate a vulnerability. If, however, the vendor is not forthcoming with a patch, and there's no other simple workaround, then the system administrator responsible for the vulnerable system has only three options; 1) Take the server offline until a patch is released. 2) Leave the server online and vulnerable to attack. 3) Deploy an IPS in front of the server to catch the attack and protect the vulnerability but still allow legitimate traffic through. The third option seems to be the safest, and while this approach may not have been the best for the Apache Killer in particular, it may be the only option for certain other vulnerabilities. Thus, the IPS for this project uses the Apache Killer vulnerability as an example to show how this approach can be applied to a certain class of vulnerabilities. Specifically, that class of vulnerabilities is network vulnerabilities involving application level (e.g. HTTP) exploits for which the maintainer does not release a patch in a timely manner.

---

[1] http://www.metasploit.com/modules/auxiliary/dos/http/apache_range_dos

[2] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.35

[3] http://httpd.apache.org/security/vulnerabilities_22.html#2.2.20

[4] http://httpd.apache.org/security/CVE-2011-3192.txt

IPS Design

An IPS is a specific type of Intrusion Detection System (IDS) that not only detects an intrusion or attack, but attempts to block it as well. IPSs, as well as IDSs, can operate on any layer of the TCP/IP stack, whether it's the link layer (Ethernet), internet layer (IP), transport layer (TCP or UDP), or application layer (HTTP). The layer that the IPS operates on depends on which layer the attack takes place, and usually the IPS operates on more than one layer. For example, the IPS for this project, from now on referred to as the Apache Killer IPS, needs access to the internet layer (to log the IP addresses of malicious clients), the transport layer (to specify ports to bind to), and the application layer (to examine the HTTP headers). This type of IPS is often referred to as an application firewall[5].

Because the Apache Killer IPS needs to examine HTTP headers, a good way to implement this IPS is as a reverse proxy using TCP sockets. A reverse proxy is a proxy server that takes requests from a set of clients and proxies the connection to a set of servers, unbeknownst to the clients[6] (see Figure 1).

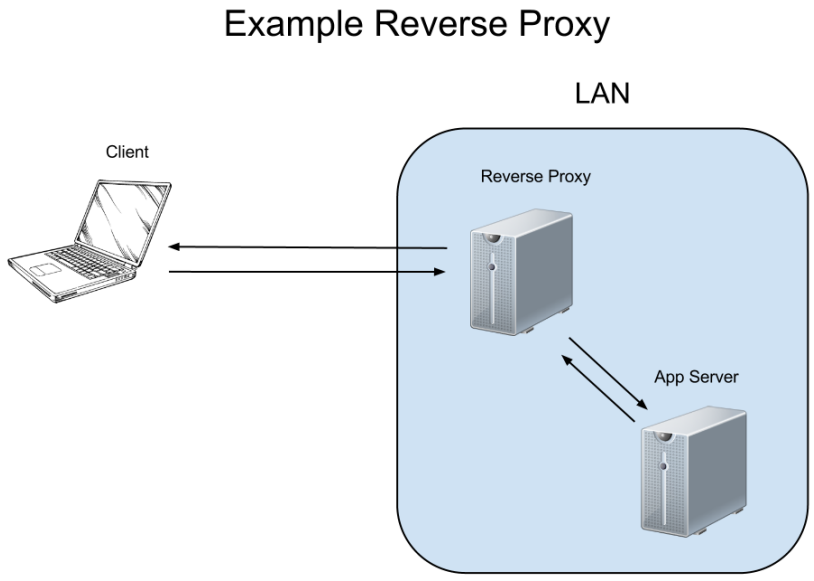

Figure 1 - Client communicating unwittingly through a reverse proxy

TCP sockets should be used for our IPS since the HTTP protocol is over TCP and since the sockets would handle any TCP fragmentation of the HTTP headers. The basic design of a TCP reverse proxy is given below. First, the reverse proxy creates a TCP listening socket which waits for clients to connect to it. Once a client connects, the reverse proxy then forks to a new process and creates a second TCP socket that connects to the server it is proxying to. At this

[5] https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/assembly/30-BSI.html

[6] http://stackoverflow.com/questions/224664/difference-between-proxy-server-and-reverse-proxy-server

point, the reverse proxy can read any data from the client socket, examine the data. potentially modify it, and then either buffer it, drop it (i.e. not pass it along), or send it to the server using the server socket. The reverse proxy can then do the same for the server (read data from the server, analyze it, and possibly send it to the client, modified or unmodified). This process of reading from the client, reading from the server, writing to the client, and writing to the server, is then repeated until either the client or the server closes their respective sockets, or until the reverse proxy sees fit to close the sockets. The original listening socket is kept open in the parent process to accept new client connections. See figure 2.

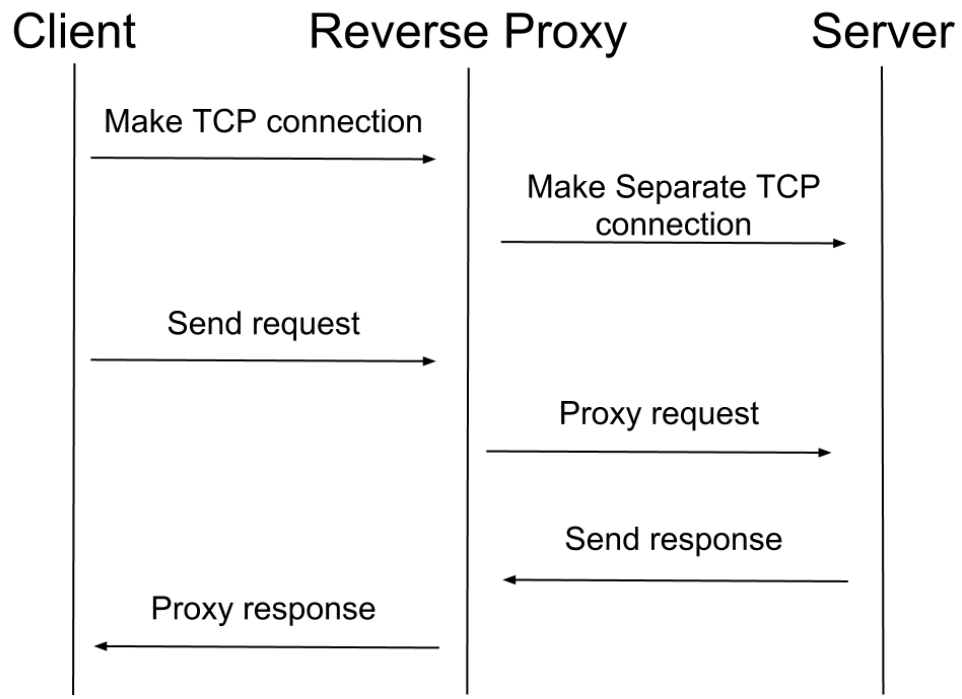

Figure 2 - Reverse proxy sequence of events

For the reverse proxy of the Apache Killer IPS, it is only necessary to examine data received from the client before proxying it; all data coming from the server is trusted and passed along to the client without inspection or modification. Because this IPS examines HTTP headers and HTTP is a text protocol, regular expressions could be used to analyze the headers. The basic logic for determining if client data should be sent along to the server or dropped is as follows. First, the IPS must determine if the whole HTTP header has been received yet, so it searches the client data for a "\r\n\r\n" character sequence, which signifies the end of an HTTP header[7]. If it doesn't find this sequence, then the IPS instructs the reverse proxy to hold the data in the buffer and get more data from the client. Otherwise, if the whole header is present, then the IPS tries to find the range section and parse out the ranges. If the number of ranges exceeds a certain value (which may vary depending on how much load the Apache server can take and how many ranges the web application needs to support), the IPS doesn't send the data to the server, logs the IP address of the client along with a warning about a potential attack,

[7] http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4

and closes both the client and server connection. If there is no range section in the header, or if the number of ranges is less than the threshold number, the data is sent along to the server unmodified.

Implementation

The Apache Killer IPS is written in C to avoid the overhead of higher level languages. The platform it is written for is GNU/Linux, since this is a common server platform. The Linux socket libraries are used to negotiate connections between the client and server. Below is an example of how a TCP listening socket is created using Linux sockets in C:

Example 1 - Creating a listening socket

```
// Create socket for incoming connections
int listening_socket =
      socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

// Construct address structure for listening address
struct sockaddr_in listening_address;
memset(listening_address, 0, sizeof(listening_address));
listening_address.sin_family = AF_INET;
listening_address.sin_addr.s_addr = htonl(INADDR_ANY);
listening_address.sin_port = htons(80);
// Bind to the local address
bind(
      listening_socket,
      (struct sockaddr *) &listening_address,
      sizeof(listening_address)
);

// Mark the socket so it will listen for incoming connections
listen(listening_socket, 5);
```

Example 1 above creates a TCP IPv4 stream socket that is bound to all IP addresses on port 80. Further, the socket is set to listen so that is can accept connections. (Note that any error checking code is left out for the sake of clarity).

To accept a connection, the following code is used.

Example 2 - Accepting connections from a listening socket

```
struct sockaddr_in client_addr;
int client_len = sizeof(client_addr);

// Wait for a client to connect
int client_socket = accept(
      listening_socket,
```

```
      (struct sockaddr *) &client_addr,
      &client_len
);
```

In Example 2 above, the accept function returns a new socket that is connected to the client. The client's IP address is placed in the `client_addr` variable. The `listening_socket` can be reused to accept new connections.

To handle the client connection, a child process is created using the `fork()` function, while the parent process continues listening for new connections. Example 3 shows the code for creating the child process.

Example 3 - Forking

```
// Create child process to handle client connection.
// The return value of fork tells the process whether it is the
// parent or the child.

int pid = fork();
if(pid == 0) {
    // this is the child process
    handle_client_socket(
        client_socket,
        inet_ntoa(client_addr.sin_addr)
    );
}
else if (pid < 0) {
    // fork() returned an error
    die("fork() failed");
}
else {
    // this is the parent process
    close(client_socket);
}
```

The `handle_client_socket()` function, which is passed both the connected client socket and the IP address of the client, is the function that performs the proxying between the client and the server, and it is only called by the child process. The parent process closes `client_socket` because the child process has its own copy of it and the parent process no longer needs it. The procedure of accepting client connections and passing them to child processes is in an infinite loop, so that it only exits by receiving an external signal.

In the `handle_client_socket()` function, a socket is created that connects to the server to be proxied to, which in the case of the Apache Killer IPS is the vulnerable Apache server that is to be protected. Then, the proxy loop is entered, in which data from the client is proxied to

the server and vice versa. The pseudocode for the proxy loop is given in Example 4 (the actual code for this is in the `handle_client_socket()` function in reverse_proxy.c).

Example 4 - Reverse proxy pseudocode

```
while true
    if there is client data to send and data is not to be buffered
        send client data to server
        if not all data was sent
            shift remaining client data to start of client buffer

    if there is server data to send and data is not to be buffered
        send server data to client
        if not all data was sent
            shift remaining server data to start of server buffer

    read from client socket into free space in client buffer
    if more client data was received
        determine fate of client data (allow, buffer, or block)
        if client data is blocked
            exit loop

    read from server socket into free space in server buffer
    if more server data was received
        determine fate of server data (allow, buffer, or block)
        if server data is blocked
            exit loop
```

As previously mentioned, data from the server is implicitly trusted for the Apache Killer IPS, so it is never buffered or blocked. What is not shown in the above pseudocode is that if any of the reads or writes over either socket fails because either the client or the server closed the connection, the proxy loop is exited. Once the proxy loop is exited, both sockets are closed if they haven't already been, and the child process is exited.

Now that the basic structure of the reverse proxy has been laid out, it remains to describe how the Apache Killer IPS determines whether or not a given set of data received from the client should be allowed through, buffered until more data comes in, or blocked from the server. The Perl Compatible Regular Expressions (PCRE) library is used to do pattern matching with the HTTP headers. The regular expressions (regexes) must initially be stored as C-style strings and then must be "compiled" by PCRE before they are ready to be used for pattern matching[8]. The "compilation" of the regexes involves creating data structures that speed up the pattern-matching process[9]. Once the regexes have been compiled, a given char buffer, in the form of a C-style string, may be searched with the regex by calling the pcre_exec function (see example

[8] http://linux.die.net/man/3/pcreapi

[9] *Ibid.*

5).

Example 5 - Searching a string with a regex using PCRE

```
int ovector[OVECCOUNT];
int subject_length = (int) strlen(subject);
int rc = pcre_exec(
    regex,          // the compiled pattern
    NULL,           // no extra data - we didn't study the pattern
    subject,        // the subject string
    subject_length, // the length of the subject
    0,              // start at offset 0 in the subject
    0,              // default options
    ovector,        // output vector for substring information
    OVECCOUNT);     // number of elements in the output vector
```

The value returned by `pcre_exec` is the number of matches found (which only includes a single match of the whole regex as well as any subexpressions that may have been in the regex). The byte offsets for each match are given in the `ovector` int array. Note that `pcre_exec` will only return one match of the given regex, and that a more complex procedure is required to find multiple matches. See the `match_regex_count()` function in apache_ips_regex.c for more details.

The following is the array of regexes used in the Apache Killer IPS.

```
const char *regex_strings[REGEX_NUM] = {
     "\r\n\r\n$",
     "HTTP\/[0-9]+?\.[0-9]+?[^0-9]",
     "Range:.+?=(.*?)\r",
     ",?[0-9]*?\-[0-9]*"
};
```

The first regex matches the end of an HTTP header, to ensure that we have the whole header. The second regex matches the HTTP version declaration in the header, to ensure that the data is HTTP and not some other protocol. The third regex matches the range section of the header and places the actual ranges into a subexpression (which is specified by the parentheses). The fourth regex matches a single range, and is used to count the number of ranges in the range section.

Syslog is used by the reverse proxy to do logging. Logging with syslog is initialized with the following command:

```
openlog("reverse_proxy", LOG_PID, LOG_USER);
```

The first argument specifies a name to associate with each log entry, which is usually the program name. The second argument tells syslog to also log the process ID of the calling process, and the third argument tells syslog what type of program is doing the logging, which in our case is a user level program.

Log entries for detected attacks in the IPS are written using the function call shown below.

```
syslog(LOG_WARNING, "data from %s was rejected\n", client_addr);
```

The first argument tells syslog that this log entry is at the warning level. The rest of the arguments are the same as what you might find in a printf call. A message that the data was rejected is given, along with the IP address of the client.

To compile the Apache Killer IPS on Linux, you must ensure that you have all of the following files in the same directory:

```
apache_ips_main.c
apache_ips_main.h
apache_ips_regex.c
apache_ips_regex.h
reverse_proxy.c
reverse_proxy.h
```

Further, you must ensure that the pcre library is installed on your system. The following command will compile the program and output the executable to a file called “apache_ips”.

```
gcc -Wall -lpcre apache_ips_main.c apache_ips_regex.c reverse_proxy.c
-o ~/apache_ips
```

The resulting executable can be run with “./apache_ips”.

Testing Environment

To test the Apache Killer IPS, three virtual machines (VMs) were used, one representing the attacker, one the router, and one the web server. The VMs were all created and run using VirtualBox, and their specifications are given below.

VM 1	
Name:	Attacker
OS:	CentOS 6.3
RAM:	1024 MB
CPUs:	1
Internal Network:	ips_internet (IP: 7.0.0.2)
Software:	Metasploit, killapache.pl, telnet, w3m

VM 2

Name:	Router
OS:	CentOS 6.3
RAM:	512 MB
CPUs:	2
Internal Network:	ips_internet (IP: 7.0.0.1), ips_lan (IP: 10.0.0.1)
Software:	iptables, Apache Killer IPS

VM 3

Name:	App Server
OS:	CentOS 6.3
RAM:	3072 MB
CPUs:	2
Internal Network:	ips_lan (IP: 10.0.0.2)
Software:	Apache 2.2.18

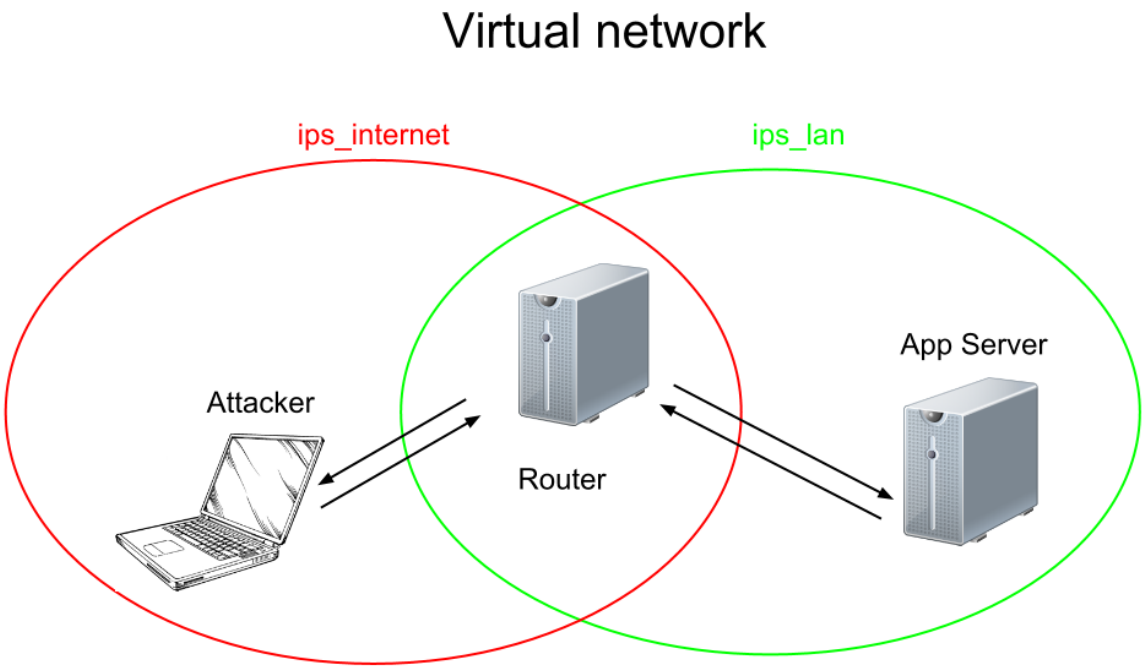

Figure 3 - Network setup of VMs

The Attacker and the Router are both on the VirtualBox internal network labeled “ips_internet,” while the Router and App Server are both on “ips_lan.” This is meant to simulate an actual

local area network (LAN) where a client (in this case, the attacker) is outside the LAN and must connect to the router in order to access machines on the LAN. Figure 3 gives a graphic representation of the virtual network layout.

The Attacker VM launches the attack targeting host 7.0.0.1 (the Router VM) on port 80 using the Metasploit module designed to exploit the Apache Killer vulnerability. Note that the Metasploit module was modified slightly from the original so that it each HTTP header contained exactly 1300 ranges of the form: 0-1,0-2,0-3,...,0-1299,0-1300. On the Router VM, the Apache Killer IPS is listening on port 80 and, as it enters the proxy loop to handle the client as described above, makes a TCP connection to Apache on the App Server VM, which is at 10.0.0.2 on port 80.

Results

The first test is to attempt to exploit Apache without the IPS in place to see how Apache would behave. To do this, the Apache Killer IPS is compiled with a NULL passed to the `client_callback` parameter of the `reverse_proxy()` function. If there's no callback function for the client, the data from the client gets passed along to the server without scrutiny.

The screenshot in figure 4 shows the output of the top command on the App Server VM just before the exploit takes place.

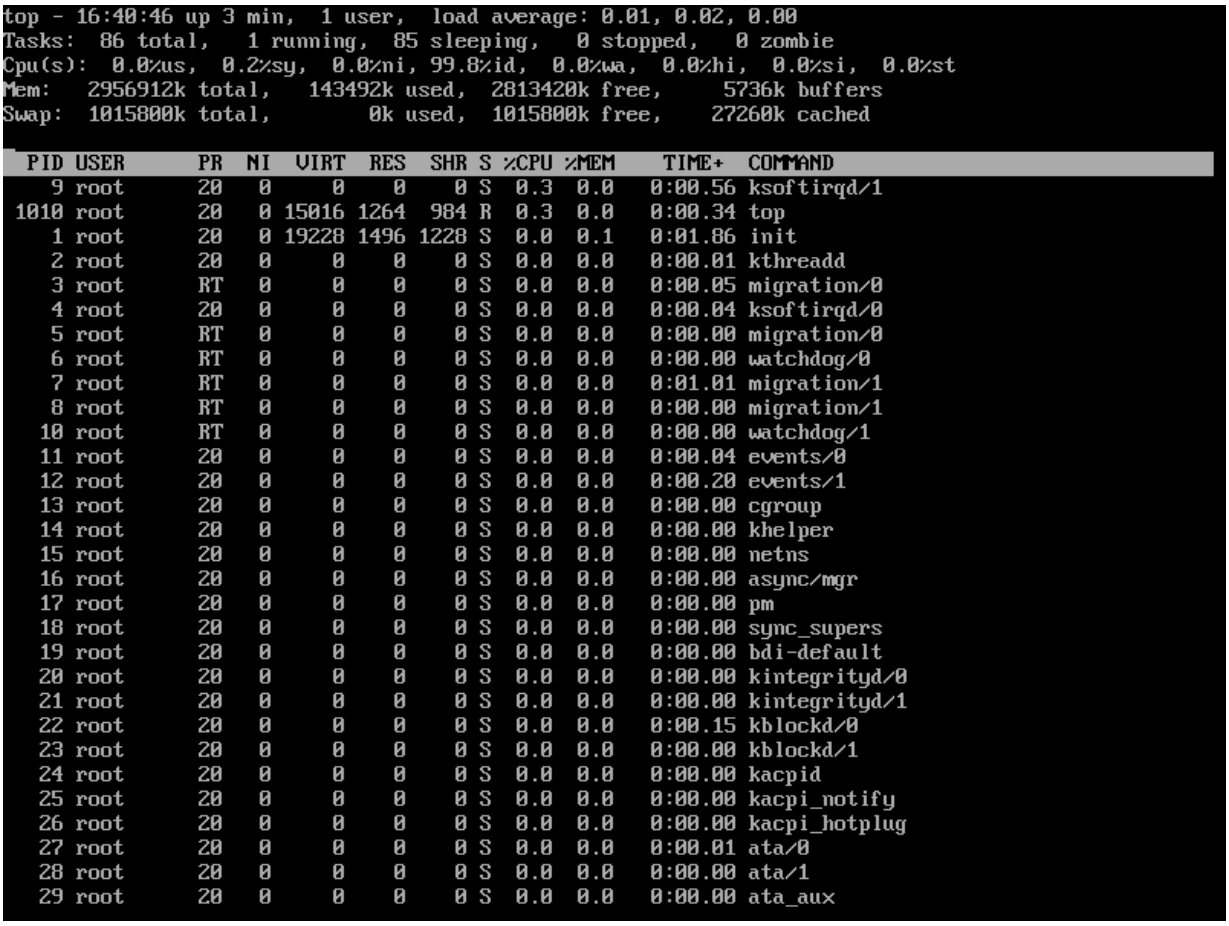

Figure 4 - Output of top command on App Server just before an attack

As the output in figure 4 shows, the machine has plenty of memory and the CPU is almost 100% idle.

Figure 5 shows the output of the top command on the App Server VM about 10 seconds after the exploit was launched.

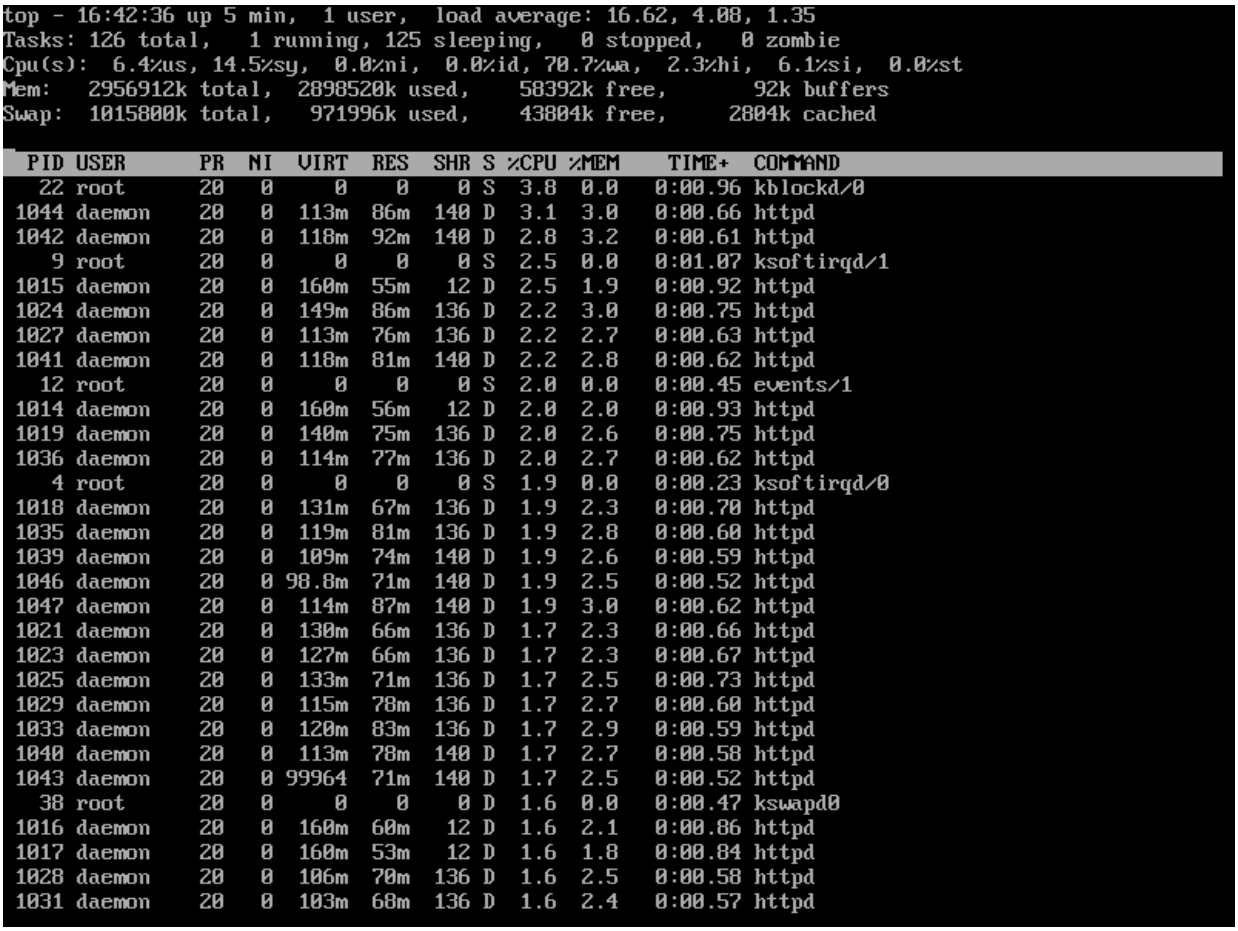

Figure 5 - Output of top command on App Server just after an attack

As can be seen in figure 5, there are now several httpd processes (httpd being the name of the Apache server binary) that are collectively taking up most of the memory and CPU. At this point, the App Server becomes intermittently responsive, and Apache takes a long time to respond to legitimate client requests (requests made by w3m, a console web browser, on the Attacker's machine). Further, it is very difficult to recover from this state cleanly; doing a hard shutdown of the App Server is often the only way to recover.

The second test is to attempt to exploit Apache with the fully functional IPS in place, to see if the IPS does indeed recognize and prevent the attack. For this test, the Apache Killer IPS is compiled with the `process_range_header()` function pointer passed to the `client_callback` parameter of the `reverse_proxy()` function. The `process_range_header()` function is where the attack is detected.

When this test is run, there is no noticeable increase in memory consumption or CPU load in the output of top on the App Server. Further, legitimate client requests, again made by w3m on the Attacker's machine, are allowed through and their responses are sent back without error. Below is an excerpt of log data generated by the IPS during an attempted attack.

```
Dec  6 02:39:21 ipsrouter reverse_proxy[1313]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1314]: data from 7.0.0.2 was buffered
Dec  6 02:39:21 ipsrouter reverse_proxy[1314]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1315]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1316]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1317]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1318]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1319]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1320]: data from 7.0.0.2 was rejected
Dec  6 02:39:21 ipsrouter reverse_proxy[1321]: data from 7.0.0.2 was rejected
```

The number in brackets represents the process ID (recall that each child process only handles a single client connection).

Conclusion

The goal of this project was to create an IPS for the Apache Killer vulnerability that could detect and block an attack while allowing legitimate client requests through, and that goal was achieved.

There are a few ways that this IPS can be improved. First, it may be useful to send an email to the system administrator alerting him/her of any attempted attack. This could be implemented as a script that runs periodically, examining the IPS logs and emailing a daily summary of anything noteworthy. Second, the log output is a bit too verbose as it's currently written, because a log entry is created for every single bad HTTP header detected. To fix this, the IPS could only log a bad HTTP header if another bad HTTP header from the same IP address hasn't been received in the past few minutes. This would greatly reduce the amount of log output while still keeping it useful. It may also be a good idea to start logging all traffic from a particular IP address from which an attack has been detected. To do this, the IPS could supply the offending IP address to a packet sniffer on the network, which would then log all packets to and from that address for a certain period of time. Another way of improving this IPS would be to add HTTPS support. One way to solve this, short of building HTTPS support into the IPS itself, would be to put another reverse proxy in front of the IPS, such as Nginx, and have Nginx handle the HTTPS authentication and proxy the client requests to the IPS over HTTP.

Despite the fact that the Apache Killer IPS was shown to be functional, there are a couple of caveats to using this particular approach for building an IPS. First of all, it should be understood that the custom-built IPS approach is only a temporary, last resort solution. If there's an official patch or workaround available, that's probably the safer bet. Additionally, writing a proxy server (or any sort of networking software) from scratch can be dangerous, especially if that software is to be deployed in a production environment and exposed to the ravages of the internet. The custom-built proxy server, while it may address the problem it was built for, may also end up introducing new vulnerabilities into the system! It is probably much safer to use a tried and tested piece of software that can be adapted to a particular need. To that end, there is an

application firewall called ModSecurity[10] that is intended to protect vulnerabilities such as the Apache Killer in a similar manner to the IPS written for this project. It is, however, a much more flexible and robust solution, and has community as well as commercial support[11].

This project increased my knowledge of the following topics: C programming in a Linux environment, network protocols (TCP/IP, HTTP), system and network administration, and protecting against application layer remote exploits.

[10] http://www.modsecurity.org/projects/modsecurity/

[11] http://www.modsecurity.org/contact/