Buffer overflows

Secure Programming
Lecture 6

Where are we?

- We have started looking at common classes of vulnerabilities
- Today, we begin looking at memory corruption vulnerabilities, focusing on stack-based overflows

Buffer overflow

- Program writes data to a buffer and overruns the buffer's boundary overwriting adjacent memory

- Mostly relevant in C and C++ programs

- Memory-safe languages are not affected
  - Dynamic bound checks (e.g., Java)
  - Automatic resizing (e.g., Perl)

Usual suspects

Buffer overflows are associated with a number of insecure library functions that make it easy to overwrite buffers

String manipulation functions without boundary checking

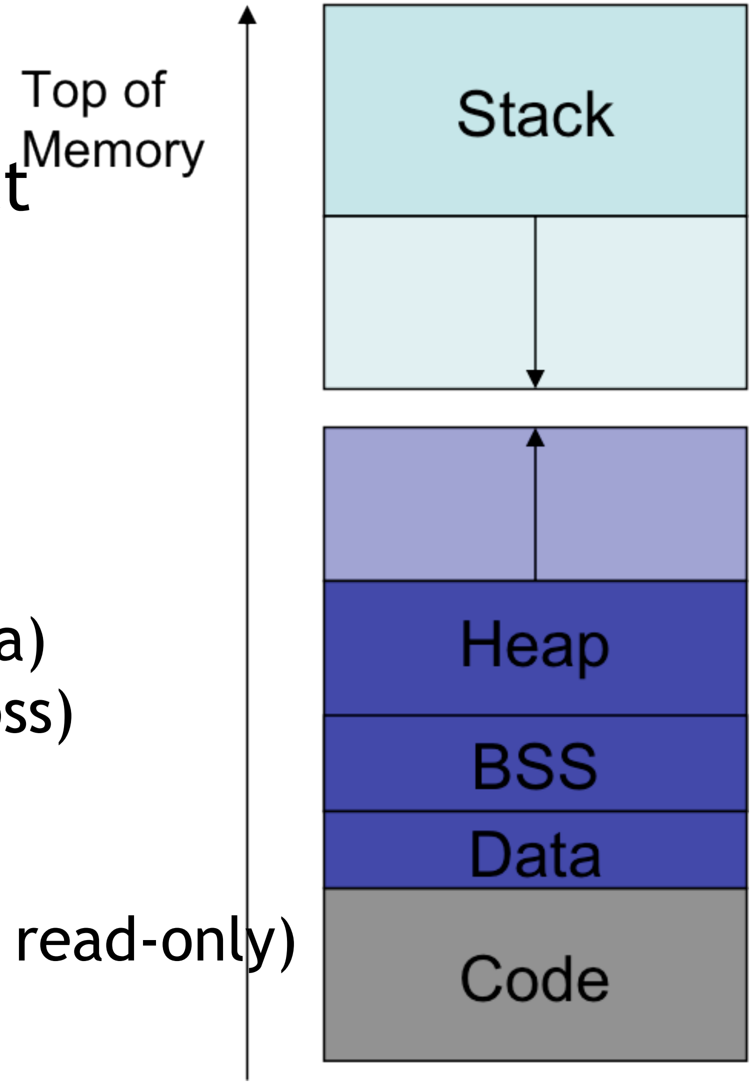
- `gets`
- `strcat`
- `strcpy`

String manipulation function with boundary checking that are used incorrectly

- `fgets`

Background: memory regions

- Each running program is allocated a number of distinct memory regions
- *Stack segment*
  - local variables
  - procedure calls
- *Data segment*
  - global initialized variables (data)
  - global uninitialized variables (bss)
  - dynamic variables (heap)
- *Code (text) segment*
  - program instructions (typically, read-only)



Stack

Usually grows towards smaller memory addresses

- Intel, Motorola, SPARC, MIPS

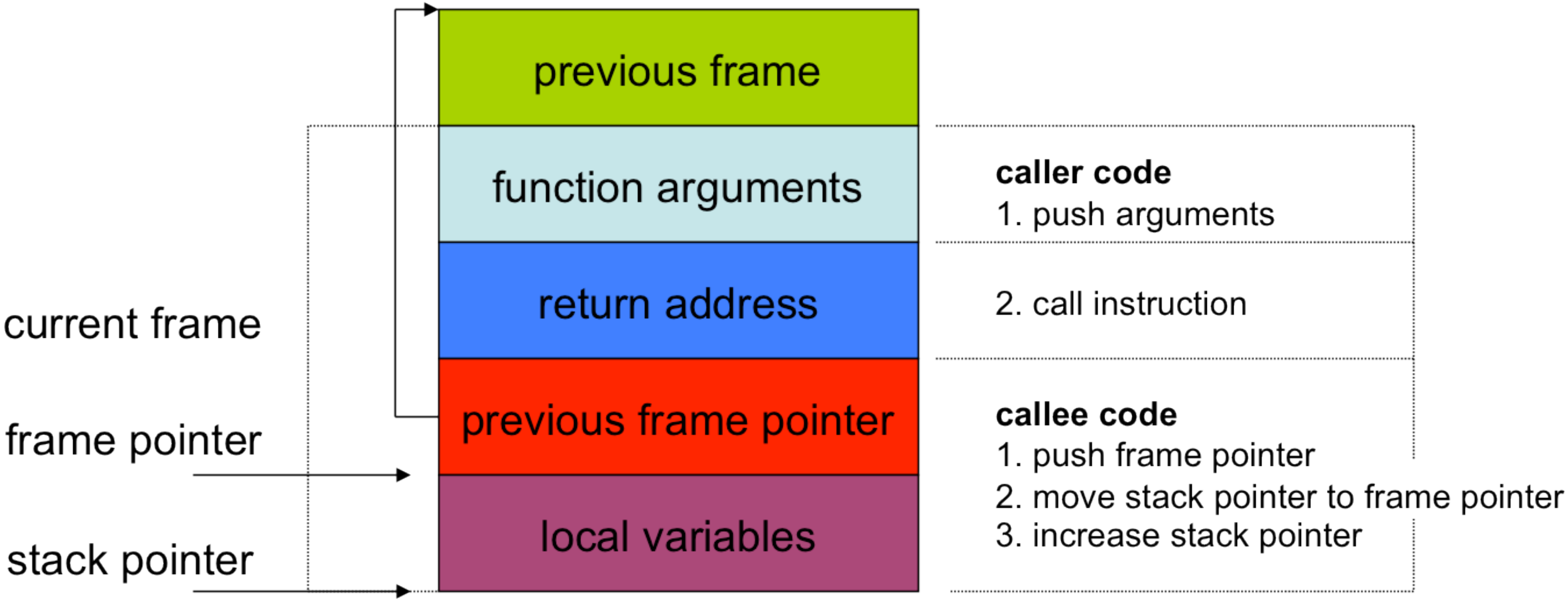
Register points to top of stack

- stack pointer – `sp`
- points to last stack element or first free slot

Composed of *frames*

- pushed on top of stack as consequence of function calls
- address of current frame stored in processor register
- frame/base pointer – `fp`
- used to reference local variables

Stack


previous frame
function arguments
return address
previous frame pointer
local variables
current frame
frame pointer
stack pointer
caller code
1. push arguments
2. call instruction
callee code
1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

Procedure call

```c
#include <stdio.h>

int foo(int a, int b) {
    int c = 42;

    return a * b + c;
}

int main(int argc, char **argv) {
    int c = 0;

    c = foo(1, 2);

    printf("%d\n", c);

    return 0;
}
```

2
1
Saved IP
Saved BP
42

Under the hood

```
(gdb) disass main
Dump of assembler code for function main:
   0x080483fd <+0>:     push   %ebp
   0x080483fe <+1>:     mov    %esp,%ebp
   0x08048400 <+3>:     and    $0xfffffff0,%esp
   0x08048403 <+6>:     sub    $0x20,%esp
   0x08048406 <+9>:     movl   $0x0,0x1c(%esp)
   0x0804840e <+17>:    movl   $0x2,0x4(%esp)
   0x08048416 <+25>:    movl   $0x1,(%esp)
   0x0804841d <+32>:    call   0x80483e4 <foo>
   0x08048422 <+37>:    mov    %eax,0x1c(%esp)
   0x08048426 <+41>:    mov    $0x8048520,%eax
   0x0804842b <+46>:    mov    0x1c(%esp),%edx
   0x0804842f <+50>:    mov    %edx,0x4(%esp)
   0x08048433 <+54>:    mov    %eax,(%esp)
   0x08048436 <+57>:    call   0x8048300
<printf@plt>
   0x0804843b <+62>:    mov    $0x0,%eax
   0x08048440 <+67>:    leave
   0x08048441 <+68>:    ret
End of assembler dump.
```

0x2
0x1
0x8048422

Under the hood

```
(gdb) disass foo
Dump of assembler code for function foo:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x10,%esp
   0x080483ea <+6>:     movl   $0x2a,-0x4(%ebp)
   0x080483f1 <+13>:    mov    0x8(%ebp),%eax
   0x080483f4 <+16>:    imul   0xc(%ebp),%eax
   0x080483f8 <+20>:    add    -0x4(%ebp),%eax
   0x080483fb <+23>:    leave
   0x080483fc <+24>:    ret
End of assembler dump.
```

0x2
0x1
0x8048422
0xbffff1a8
0x2a

Let’s run it

```
(gdb) br foo
Breakpoint 1 at 0x80483ea: file procedure.c, line 4.
(gdb) r
Starting program: procedure foo

Breakpoint 1, foo (a=1, b=2) at procedure.c:4
4           int c = 42;
(gdb) stepi
6           return a * b + c;
(gdb) x/8wx $ebp -4
0xbffff174: 0x0000002a 0xbffff1a8 0x08048422 0x00000001
0xbffff184: 0x00000002 0x002d8ff4 0x00166225 0x0011f270
```

0x2
0x1
0x8048422
0xbffff1a8
0x2a

Returning

```
(gdb) x/i $eip
=> 0x80483fb <foo+23>:  leave
(gdb) info r ebp esp
ebp            0xbffff178       0xbffff178
esp            0xbffff168       0xbffff168
(gdb) si
(gdb) info r ebp esp
ebp            0xbffff1a8       0xbffff1a8
esp            0xbffff17c       0xbffff17c
(gdb) x/i $eip
=> 0x80483fc <foo+24>:  ret
(gdb) si
0x08048422 in main (argc=2, argv=0xbffff244) at
procedure.c:12
(gdb) info r ebp esp eip
ebp            0xbffff1a8       0xbffff1a8
esp            0xbffff180       0xbffff180
eip            0x8048422        0x8048422 <main+37>
```

0x2
0x1

Reference

- [GDB: The GNU Project Debugger]
- [Intel 64 and IA-32 Architectures Software Developer's Manual]

Vulnerable program

```
#include <stdio.h>
#include <string.h>

void vulnerable(char* param)
{
    char buffer[100];
    strcpy(buffer, param);
}

int main(int argc, char** argv)
{
    vulnerable(argv[1]);
    printf("OK\n");

    return 0;
}
```

Let’s crash

```
06-buffer-overflows$ ./vuln test
OK

marco@ubuntu:$ ./vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

Let's crash

```
$ gdb vuln
(gdb) r
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA
Starting program: vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info r esp ebp eip
esp            0xbffff0f0     0xbffff0f0
ebp            0x41414141     0x41414141
eip            0x41414141     0x41414141
(gdb) x/20wx $esp -32
0xbffff0d0:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff0e0:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff0f0:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff100:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff110:     0x41414141     0x41414141     0x41414141     0x41414141
```

To reproduce

Disable defense mechanisms!

```
$ gcc -fno-stack-protector \
      -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 \
      vuln.c -o vuln
```

(We'll see these in detail in a few lectures.)

Smashing the stack

Key idea: Overwrite a pointer with the address of code we want to execute

1. locate a pointer that (eventually) will be copied to the EIP register or that points to data that will be copied to the EIP
   - function pointers (on the stack, heap, BSS, …)
   - saved EBP
   - procedure return address
   - entry in the GOT
   - jmp_buf
2. overwrite the pointer with our value

Smashing the stack

- A procedure contains a local buffer variable allocated on the stack
- The procedure copies user-controlled data to the buffer without verifying that the data size is smaller than the buffer
- The user data overwrites the other variables on the stack, up to the *return address* saved in the function frame
- When the procedure returns, the program fetches the return address from the stack and copies it to the EIP register
- Since we can control the return address, we can jump to an address of our choice

Where do we jump to?

Address inside a buffer whose content is under user control

- PRO: works for remote attacks
- CON: the attacker needs to know the address of the buffer
- CON: the memory page containing the buffer must be executable

Where do we jump to?

Address of an environment variable

- PRO: easy to implement, works with tiny buffers
- CON: only for local exploits
- CON: some programs clean the environment
- CON: the environment must be executable

Where do we jump to?

- Address of a function inside the program
- PRO: works for remote attacks
- PRO: does not require an executable stack
- CON: need to find the right code
- CON: one or more fake frames must be put on the stack

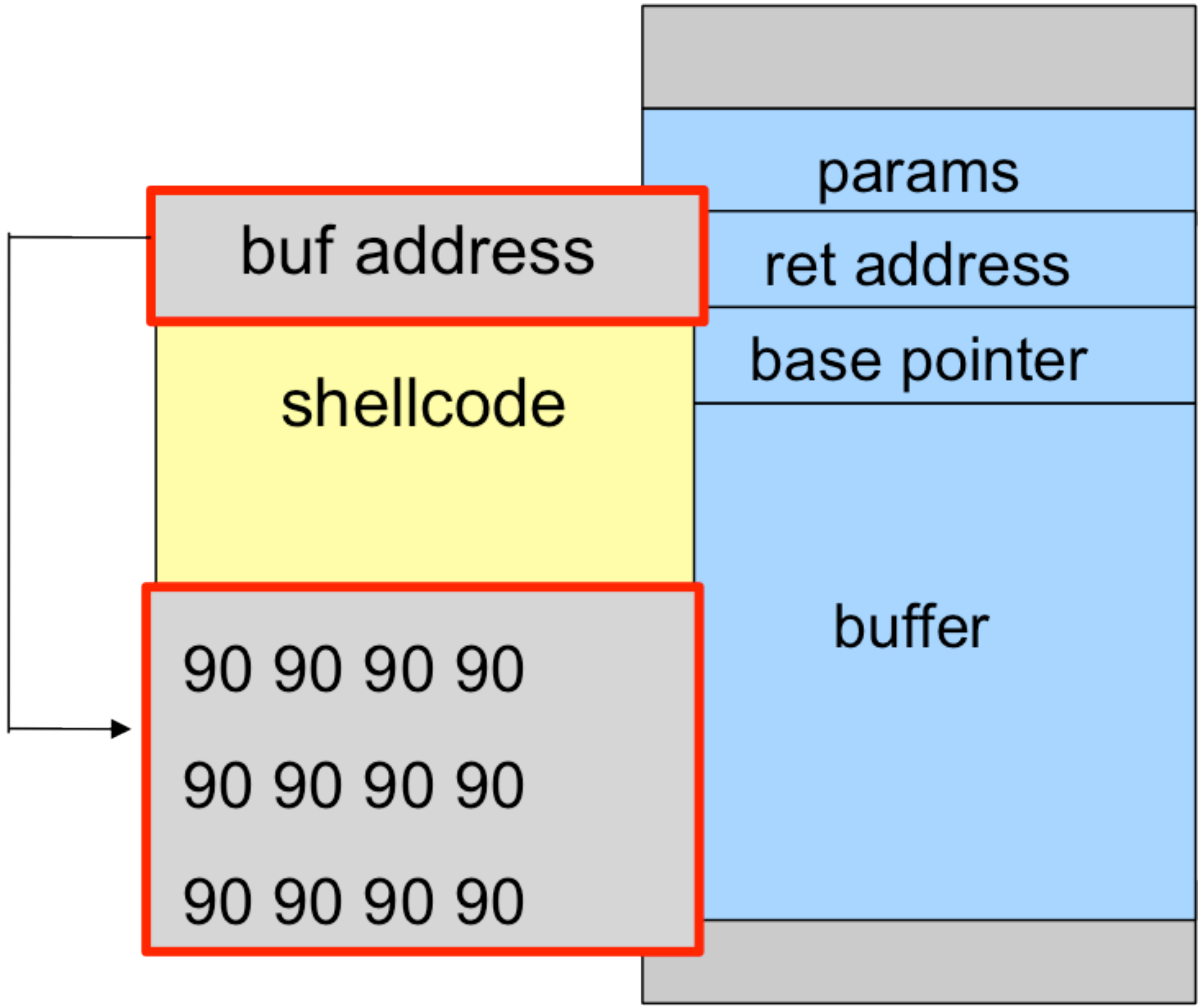
Jumping into the buffer

- The buffer that we are overflowing is usually a good place to put the code (shellcode) that we want to execute
- The buffer is somewhere on the stack, but in most cases the exact address is unknown
- The address must be precise: jumping one byte before or after would just make the application crash
- On the local system, it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine: any change to the environment variables affects the stack position

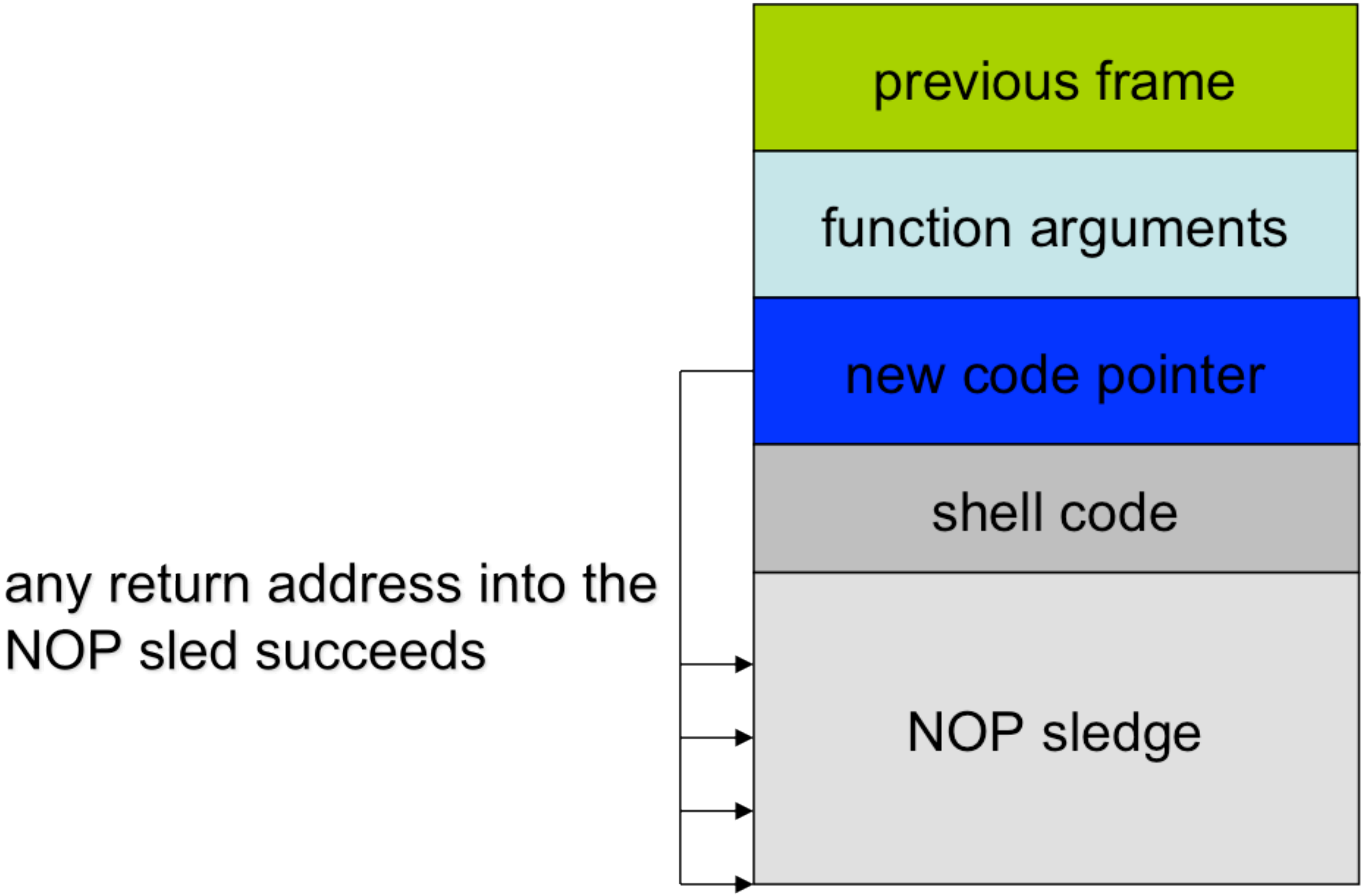
Solution #1: NOP sled

- A sled is a “landing area” in front of the shellcode
  - work arounds the problem of finding the right shellcode address
- Must be created in a way such that whenever the program jumps into it:
  - it always finds a valid instruction
  - it always reaches the end of the sled and the beginning of the shellcode
- The simplest sled is a sequence of NOP instructions
  - single byte instruction (0x90) that does not do anything
  - more complex sleds possible

Assembling the malicious input



Assembling the malicious input



Solution #2: register-based jump

- Find a register that points to the buffer (or somewhere into it)
  - ESP
  - EAX (return value of a function call)

- Locate an instruction that jumps/calls using that register
  - can also be in one of the libraries
  - does not need to be a real instruction, just look for the right sequence of bytes (jmp *$esp = 0xFF 0xE4)

- Overwrite the return address with the address of that instruction

(We will talk about address space randomization)

Solution #3: heap spraying

- We will talk about this later

Some history

Morris worm (1988): overflow in fingerd

- 6,000 machines infected (10% of the Internet)
- Internet had to be switched off
- CERT is created

CodeRed (2001): overflow in MS IIS server

- 300,000 machines infected in 14 hours

SQL Slammer (2003): overflow in MS SQL server

- attack: 1 UDP packet
- 75,000+ machines infected in 10 minutes

In 2003, around 75% of the vulnerabilities were buffer overflows

References

- Aleph One, Smashing The Stack For Fun And Profit, Phrack 49, 1996
- spoonm, Recent Shellcode Developments, ReCon 2005

Next time

Introduction to assembly

Read J. Mason et al., English Shellcode, CCS 2009