
Secure Programming II

(SecProg 2)

Engin Kirda

kirda@eurecom.fr

Administrative Issues

- We have 23 registrations
 - Challenge 1 has started
 - 11 people have solved it so far. Good work
 - Registration is over
 - Challenge 2 will go online today at 16.00
 - Buffer overflow

Low-Level Essentials for Understanding Security Problems

Computer Architecture

- The modern computer architecture is based on “Von Neuman”
 - Two main parts: CPU (Central Processing Unit) and Memory
 - This architecture is used everywhere (e.g., even mobile phones)
 - This architecture is fundamental, has not changed yet
- What is memory used for?
 - E.g., location of cursor, size of windows, shape of each letter being displayed, graphics of icons, text, values, etc...
 - Von Neuman also says that not only data, but programs should also be in main memory

The CPU

- Storing data by itself, of course, is not enough
 - The CPU reads instructions from memory one by one
 - Executes them (*fetch-execute-cycle*)
- Following components make up the CPU
 - Program Counter
 - Instruction Decoder
 - Data bus
 - General-purpose registers
 - Arithmetic and logic unit

Program Counter and Instruction Decoder

- Is used to tell the CPU where to fetch next instruction
 - There is no difference between memory and data
 - Program counter holds memory address of next instruction
- The instruction decoder then makes sense of the instruction
 - Addition? Substraction? Multiplication? Move operation? Etc.
 - A typical instruction usually consists of memory locations as well
 - E.g., move this piece of data from memory address X to memory address Y

The Data Bus and Registers

- The data bus connects the memory and the CPU
 - i.e., it is the actual, physical wire
- In addition to the memory, the processor has high-speed, special memory location
 - Called *registers*
 - Special-purpose registers
 - General-purpose registers
 - Registers are used for computation

Arithmetic and Logic Unit

- Once instruction has been decoded, CPU passes data and decoded instruction to ALU
 - Now, the instruction is actually executed
 - Results of the computation are placed on the data bus and sent to memory locations given in the instruction
 - For example, we can tell ALU to add 1 to register A and place result in register B

Some basics (you know them ;))

- The number attached to each storage location
 - ... is an *address*
 - A single storage location is called a *byte*
 - On x86 processors, a byte is between 0..255
 - Obviously, two bytes can be used to represent any number between 0..65536
 - Four bytes can be used to represent numbers between 0..4294967295., Luckily, we do not have to worry about this. The architecture helps us to do math with 4 byte numbers
- Addresses sizes are in *words* (a word is 4 bytes)
 - Note that this means that a number and an address are dealt with the same way

Data Accessing Methods

- Processors have a number of different ways of accessing data
 - Known as *addressing modes*
 - The simplest method is known as *immediate mode*
 - Data access is enabled in instruction itself
 - In *register addressing mode* the instruction contains a register to access rather than memory location
 - In *direct addressing mode*, the instruction contains memory address
 - In *indexed addressing mode*, the instruction contains memory address to access and an index register to offset
 - E.g., Address 1000, register=4, address=1004

Data Accessing Methods

- Processors have a number of different ways of accessing data
 - *Indirect addressing mode*
 - *We take address that is stored in register*
 - Base point addressing mode
 - You take address in register and add an offset
 - Used a lot

A Simple Program in Assembler

```
# No INPUT  
# Returns a status code, you can view it by typing echo $?  
# %ebx holds the return code  
.section .data  
.section .text  
.globl _start  
  
_start:  
mov $1, %eax # This is the sys call for exiting program  
movl $0, %ebx # This value is returned as status  
int $0x80 # This interrupt calls the kernel, to execute sys call
```

So now we have the source code...

- So how do we create the application?
 - Well, we need to assemble and link the code
 - This can be done by using the assembler as:
 - *as exit.s -o exit.o*
 - *ld -o exit exit.o*

Anatomy of the Code

- `.section` breaks up the program into sections
 - `.data`, obviously, is used for variables and data you might need
 - `.text` is the code of the program that you write
 - `.globl __start` indicates that `__start` is a symbol, required by the linker
 - `__start`, in fact, indicates that the loader will load and start the program from this location
 - The next instruction, `movl`, has two operands: *source* and *destination*
 - `movl, addl, subl`

Registers on the x86 Architecture

- General purpose registers:
 - `%eax, %ebx, %ecx, %edx, %edi, %esi`
- Special purpose registers:
 - `%ebp, %esp, %eip, %eflags`
- Some registers (e.g., `%eip, %eflags`) can only be accessed through special instructions
- The \$ sign before a number means that we are using *immediate addressing mode*

Let us write a more complicated Program

- Task: Find the maximum of a list of numbers
 - Questions to ask:
 - Where will the numbers be stored?
 - How do we find the maximum number?
 - How much storage do we need?
 - Will registers be enough or is memory needed?
 - Let us designate registers for the task at hand:
 - %edi holds position in list
 - %ebx will hold current highest
 - %eax will hold current element examined

“Algorithm” we use

- Check if %eax is zero (i.e., termination sign)
 - If yes, exit
 - If not, increase current position %edi
 - Load next value in the list to %eax
 - We need to think about what addressing mode to use here
 - Compare %eax (the current value) with highest value so far %ebx
 - If the current value is higher, replace %ebx
 - Repeat until termination

Let's get down to the code

```
.section .data  
    data_items:  
        .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0  
.section .text  
.globl _start  
_start:  
mov $0, %edi # Reset index  
movl data_items(,%edi,4), %eax  
movl %eax, %ebx #First item is the biggest so far  
start_loop:  
cmpl $0, %eax  
je loop_exit
```

Let's get down to the code

```
incl %edi # Increment edi  
movl data_items(,%edi,4), %eax #load the next value  
cmpl %ebx, %eax % Compare ebx with eax  
jle start_loop # if it is less, then just jump to the beginning  
movl %eax, %ebx #Otherwise, store the new largest number  
jmp start_loop  
  
loop_exit:  
movl $1, %eax # Remember the exit sys call? It is 1  
int $80
```

Important Instruction

- The compare instruction
 - `cmpl $0, %eax`
 - `je end_loop`
 - Other jump instructions *jg, jge, jl, jle, jmp*
- `mov` instruction
 - Used often. One of the most important and common instructions that you are going to see
 - ... and use, for example, when writing shell code

Addressing Modes in Assembler

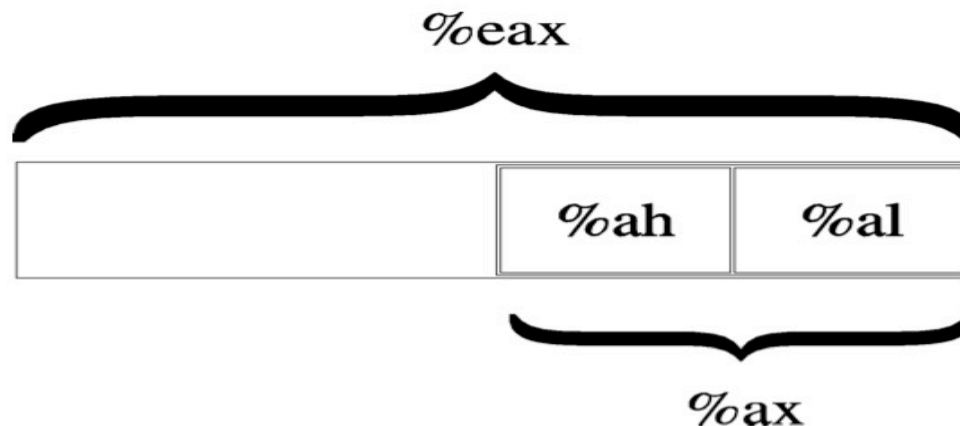
- The general form of memory address references is this:
 - ADDRESS_OR_OFFSET(%BASE_OR_OFFSET, %INDEX, MULTIPLIER)
- To calculate the address, simply perform the following calculation:
 - FINAL ADDRESS = ADDRESS_OR_OFFSET + %BASE_OR_OFFSET + MULTIPLIER * %INDEX
 - ADDRESS_OR_OFFSET and MULTIPLIER must both be constants, while the other two must be registers.

You can address all methods with this approach

- direct addressing mode
 - `movl ADDRESS, %eax`
 - uses `ADDRESS_OR_OFFSET`
- indexed addressing mode
 - `movl string_start(,%ecx,1), %eax`
 - uses the `ADDRESS_OR_OFFSET` and the `%INDEX` portion
- indirect addressing mode
 - `movl (%eax), %ebx`
- base pointer addressing mode
 - `movl 4(%eax), %ebx`
- Immediate mode (already seen)

What if we do not want to move a word?

- Suppose you only need to move a *byte* at a time, not a word
 - You can use the *movb* instruction
 - In `%eax`, the least significant half is addressed by `%ax` (i.e., 2 bytes)
 - `%ax` is divided into `%ah` and `%al` (1 byte in size)



Functions

- A function is composed of several different pieces
 - function name
 - Symbol that represents where the function starts
 - function parameters
 - Data items passed to function for processing
 - Local variables
 - Temporary storage areas used in the function
 - Thrown away when the processing finishes
 - Static variables
 - Storage area that is reused over invocations
 - Global variables
 - Storage areas outside the function

The Return Address

- The return address is a parameter which tells the function where to resume executing after the function is completed
 - It is “invisible” – the programmer does not necessarily know the address
 - When a function is invoked, the calling point is saved
 - When the function completes, it returns to the initial calling point
 - In machine code, functions are called with *call* and they return when they execute *ret*
- Return value
 - Usually, a single value is returned to caller

The Calling Convention

- The way that the variables are stored and the parameters and return values are transferred by the computer varies from language to language as well
 - This variance is known as a language's *calling convention*
- The assembly language can use any calling convention
 - You can make one up yourself... however...
 - If you want to interoperate with other languages, you need to follow their calling conventions
 - E.g., suppose you want your code to be callable from a C program...

The Stack

- Each computer program that runs uses a region of memory called the stack to enable functions to work properly
 - You generally keep the things that you are working on toward the top, and you take things off as you are finished working with them
- The computer's stack lives at the very top addresses of memory
 - You can push values onto stack using *pushl*
 - You can retrieve items from the stack using *popl*

The Stack

- Where is the “top” of the stack
 - Because of architectural considerations, the computer stack grows from higher addresses to lower addresses
 - i.e., it grows downwards
 - How do we know where the “top” of the stack is?
 - The `%esp` register stores a pointer to stack location
 - If something is pushed onto stack, the stack decreases by `%esp - 4` if `pushl` is used
 - Yes, but if I only want to read? How can I do this without popping?
 - `movl (%esp), %eax`
 - `movl 4(%esp), %eax` (for addressing a higher value)

The C calling convention

- The stack is the key element for implementing a function's...
 - local variables, parameters, and return address
 - Before executing a function, a program pushes all of the parameters for the function onto the stack in the reverse order that they are documented
 - Program issues *call* instruction
 - *call* does two things
 - pushes address of next instruction (i.e., return)
 - Modifies `%eip` to point to function start

The C calling convention

- Parameter #N

...

Parameter 2

Parameter 1

Return Address <--- (%esp)

- Now, function has to do some thing
 - It saves the current base pointer register %ebp
 - Needed for local variables and parameters
 - pushl %ebp
 - movl %esp, %ebp

The C calling convention

- Copying the stack pointer into the base pointer at the beginning of a function allows you to always know where your parameters are

Parameter #N $\leftarrow N \cdot 4 + 4(\%ebp)$

...

Parameter 2 $\leftarrow 12(\%ebp)$

Parameter 1 $\leftarrow 8(\%ebp)$

Return Address $\leftarrow 4(\%ebp)$

Old $\%ebp \leftarrow (\%esp)$ and $(\%ebp)$

The C calling convention

- *stack frame* consists of all of the stack variables used within a function
- Next, the function reserves space on the stack for any local variables it needs
 - This is done by simply moving the stack pointer out of the way
 - E.g., if we need two words: `subl $8, %esp`
 - Variables are local because when function returns, the stack frame is reset and variables disappear

The C calling convention

- Our stack now looks like this:
 - Parameter #N $\leftarrow N \cdot 4 + 4(\%ebp)$
 - ...
 - Parameter 2 $\leftarrow 12(\%ebp)$
 - Parameter 1 $\leftarrow 8(\%ebp)$
 - Return Address $\leftarrow 4(\%ebp)$
 - Old $\%ebp \leftarrow (\%ebp)$
 - Local Variable 1 $\leftarrow -4(\%ebp)$
 - Local Variable 2 $\leftarrow -8(\%ebp)$ and $(\%esp)$

The C calling convention

- Now we can use the `%ebp` for accessing all variables
 - `%ebp` was specifically designed for this purpose
- When a function is done...
 - It stores its return value in `%eax`
 - It resets the stack to what it was when it was called
 - It gets rid of the current stack frame and puts the stack frame of the calling code back into effect
 - It returns by invoking the `ret` command
 - `movl %ebp, %esp`
`popl %ebp`
`ret`

Buffer Overflows

Buffer Overflows

- A buffer overflow occurs any time the program attempts to store data beyond the boundaries of a buffer, overwriting the adjacent memory locations
 - Results from mistakes while writing the code
 - Unfamiliarity with the language
 - Ignorance about security issues
 - Unwillingness to take any extra effort
- Vulnerable software
 - Mostly C/C++ programs
 - Not language with automated memory management
 - Java, Perl, Python, C#

Buffer Overflows

- Common targets
 - setuid/setgid programs
 - network servers
- Goals
 - Overwrite other “interesting” variables (file names, passwords, pointers...)
 - Force the program to execute operations it was not intended to do:
 - inject (or simply find) code into the process memory
 - change flow of control (flow of execution) to execute that code

Buffer Overflows

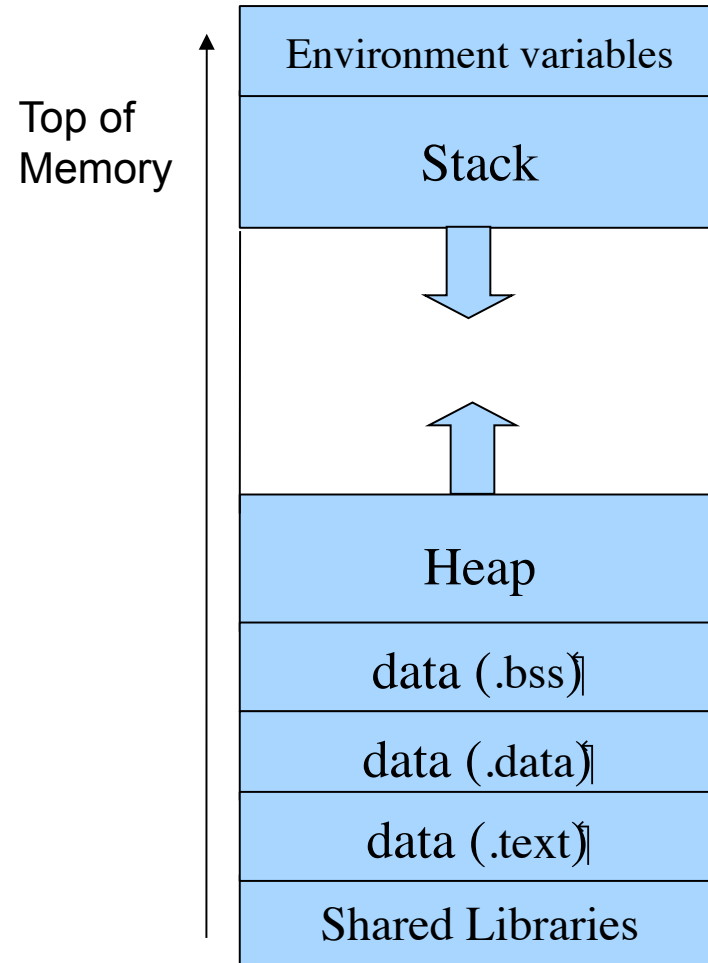
- Morris worm (1988): overflow in fingerd
 - 6,000 machines infected (10% of the Internet)
- CodeRed (2001): overflow in MS-IIS server
 - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
 - 75,000 machines infected in **10 minutes**
- In 2003, around 75% of the vulnerabilities were buffer overflows (CERT)

Part I

A deeper look into the STACK

Memory Layout

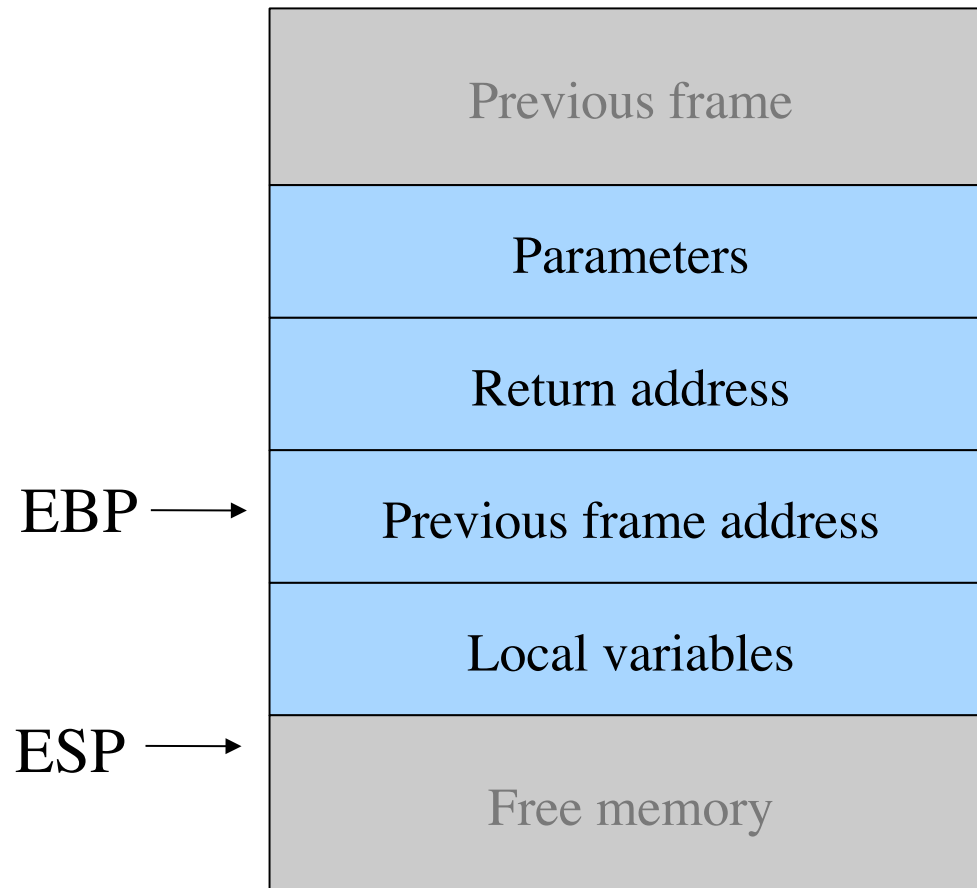
- Stack segment
 - local variables
 - procedure activation records
- Data segment
 - global uninitialized variables (.bss)
 - global initialized variables (.data)
 - dynamic variables (heap)
- Code (Text) segment
 - program instructions
 - usually read-only
- In linux, under the proc filesystem
 - >cat /proc/<pid>/maps



The Stack

- In most architectures (Intel, Motorola, Sparc), stack grows towards bottom
- The ESP (stack pointer) register points to the top of the stack
- The EBP (base pointer) points to the current frame
- Each frame contains:
 - Return address (where to jump at the end of the function)
 - Address of the previous frame
 - Parameters
 - Local variables of the function

Frame



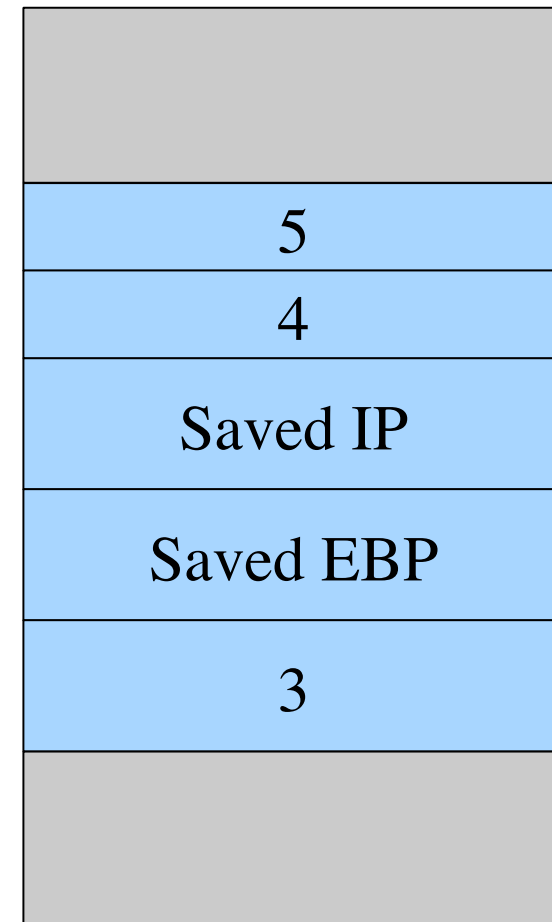
Procedure Call

```
int foo(int a, int b)
{
    int i = 3;

    return (a + b) * i;
}
```

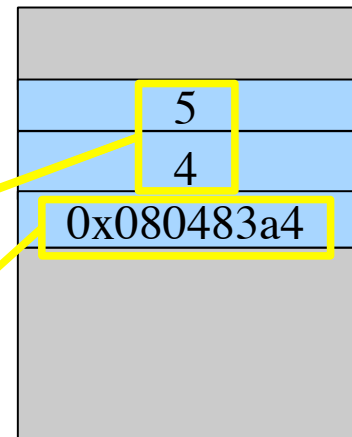


```
int main()
{
    int e = 0;
    e = foo(4, 5);
    printf("%d", e);
}
```



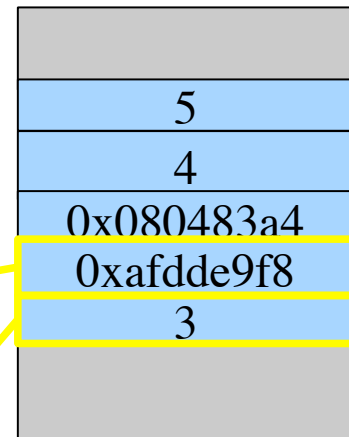
A closer look

```
(gdb) disas main
Dump of assembler code for function main:
0x0804836d <main+0>:   push   %ebp
0x0804836e <main+1>:   mov    %esp,%ebp
0x08048370 <main+3>:   sub    $0x18,%esp
0x08048373 <main+6>:   and    $0xffffffff0,%esp
0x08048376 <main+9>:   mov    $0x0,%eax
0x0804837b <main+14>:  add    $0xf,%eax
0x0804837e <main+17>:  add    $0xf,%eax
0x08048381 <main+20>:  shr    $0x4,%eax
0x08048384 <main+23>:  shl    $0x4,%eax
0x08048387 <main+26>:  sub    %eax,%esp
0
0x08048389 <main+28>:  movl   $0x0,0xffffffffc(%ebp) ↗
0x08048390 <main+35>:  movl   $0x5,0x4(%esp) ↗
0x08048398 <main+43>:  movl   $0x4,(%esp) ↗
0x0804839f <main+50>:  call  0x8048354 <foo>
0
```



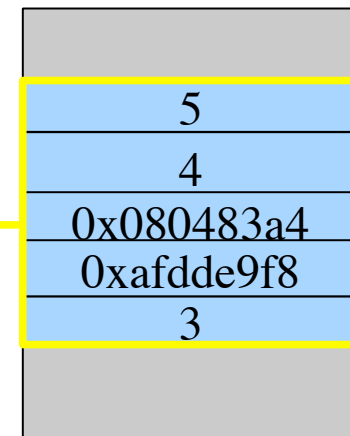
A closer look

```
(gdb) breakpoint foo
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: ./test1
Breakpoint 1, 0x0804835a in foo ()
(gdb) disas
Dump of assembler code for function foo:
0x08048354 <foo+0>:    push   %ebp
0x08048355 <foo+1>:    mov    %esp,%ebp
0x08048357 <foo+3>:    sub   $0x10,%esp
0x0804835a <foo+6>:    movl  $0x3,0xffffffff(%ebp)
0x08048361 <foo+13>:   mov   0xc(%ebp),%eax
0x08048364 <foo+16>:   add   0x8(%ebp),%eax
0x08048367 <foo+19>:   imul  0xffffffff(%ebp),%eax
0x0804836b <foo+23>:   leave
0x0804836c <foo+24>:   ret
End of assembler dump.
(gdb) ↑
```



The “foo” frame

```
(gdb) stepi
0x08048361 in foo ()
(gdb) x/12wx $ebp-16
0xaf9d3cc8: 0xaf9d3cd8 0x080482de 0xa7faf360 0x00000003
0xaf9d3cd8: 0xafdde9f8 0x080483a4 0x00000004 0x00000005
0xaf9d3ce8: 0xaf9d3d08 0x080483df 0xa7fadff4 0x08048430
```



Part II

Taking Control of the Program

The Idea

- Overwrite a pointer with the address of our code
- First, locate a pointer that will be copied to the EIP register, or that points to the data that will be copied to the EIP:
 - Function pointers
 - Saved EBP
 - Entry in the GOT (Global Offset Table)
- Second: Overwrite pointer, with *our* value
 - *Stack overflow* (Challenge 2 and 3)
 - Heap overflow
 - Format string

Smashing the Stack

- A procedure contains local variable allocated on the stack
- Procedure copies user controlled data (input) to the buffer without verifying that the data size is smaller than the buffer
 - Potentially dangerous functions: *strcpy*, *strcat*, *gets*, *scanf*
- The user data overwrites all other variables on the stack, up to the return address
- Procedure returns, program fetches the return address that has been modified and jumps to it

Example

```
// Test2.c
#include <stdio.h>
#include <string.h>

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

int main(int argc, char* argv[] )
{
    vulnerable(argv[1]);
    printf("Everything's fine\n");
}
```

Buffer that can contain 100 bytes

Copy an arbitrary number of characters from **param** to **buffer**

Let's Make it Crash

```
> ./test2 hello
Everything's fine

> ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault

>
```

Huh, what happened?

```
> gdb ./test2

(gdb) run hello
Starting program: ./test2
Everything's fine

(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: ./test2 AAAAAAAA...
Program received signal SIGSEGV,
Segmentation fault.
0x41414141 in ?? ()
```

	41 41 41 41 41
params	41 41 41
ret address	41 41 41 41
saved EBP	41 41 41 41
buffer	41 41 41 41
	41 41 41 41
	41 41 41 41

Choosing Where to Jump

- Address inside a buffer of which the attacker controls the content
 - PRO: works for remote attacks
 - CON: the attacker needs to know the address of the buffer, the memory page containing the buffer must be executable
- Address of an environment variable
 - PRO: easy to implement, works with tiny buffers
 - CON: only for local exploits, some programs clean the environment, the stack must be executable
- Address of a function inside the program
 - PRO: works for remote attacks, does not require an executable stack
 - CON: need to find the right code

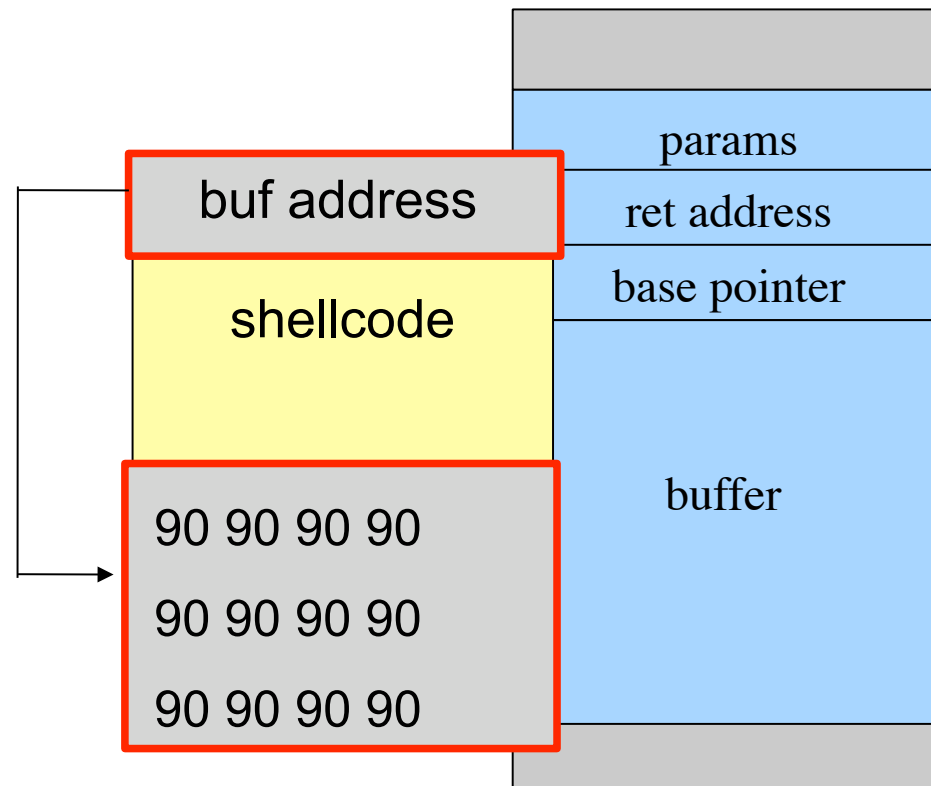
Jumping into the Buffer

- 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 affect the stack position

Solution: The NOP Sled

- A sled is a “landing area” that is put in front of the shellcode
- Must be created in a way such that wherever 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 no operation (NOP) instructions
 - Single byte instruction (0x90) that does not do anything
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area

Assembling the Malicious Buffer



Conclusion

- Today, we looked at low level programming and buffer overflows
- Next week, we continue to look at other types of overflows and how you can create your own shell code
- Read the buffer overflow tutorials on the Challenge 2 web page
- See you next week ;)