Offensive Security
My First Buffer Overflow: Tutorial

César Bernardini

University of Trento
cesar.bernardini@unitn.it

October 12, 2015

Cesar Bernardini

- Postdoctoral Fellow at UNITN
- PhD Student at INRIA-LORIA
- Master in Computer Science at Universidad Nacional de Córdoba, Argentina
- Junior Security Research at Binamuse.com
  - **CVE-2010-3429**
- `http://www.loria.fr/~bernardc`

Bibliography

Bibliography & Links

- *Hacking, The Art of Exploitation* – Jon Erickson
- *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* – Chris Anley, John Headman, Felix Lindner and Gerardo Richarte
- BinAmuse.com – `http://www.binamuse.com`

Hacking

What is Hacking?

- Hacker is a term for both those who write code and those who exploit it.
- Hacking is really just the act of finding a clever and counterintuitive solution to a problem

If we want to find counterintuitive solutions...

We need to understand how technologies work **in-depth**

How to Hack?

The Hacking Steps

1. Understand the program execution
2. Understand the environment
   - OS (Linux 2.6.x), Programming language (C), Compiler (GCC), Processor (x86 32 bits)
3. Look for errors on the code
4. (when possible) Exploit

Assumptions on the course

- Proficiency on the C language and and the GCC compiler

Outline

Understanding Programs Execution

```c
#include <stdio.h>

int main()
{
	int i;
	for (i=0; i<10; i++)
	{
		puts("Hello, world!\n");
	}

	return 0;
}
```

How does Linux execute this program? (puts=printf)

Understanding Programs Execution
Assembla Version in Gentoo VM

```
reader@hacking:~/booksrc $ objdump -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   %ebp
 8048375:       89 e5                   mov    %esp,%ebp
 8048377:       83 ec 08                sub    $0x8,%esp
 804837a:       83 e4 f0                and    $0xfffffff0,%esp
 804837d:       b8 00 00 00 00          mov    $0x0,%eax
 8048382:       29 c4                   sub    %eax,%esp
 8048384:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 804838b:       83 7d fc 09             cmpl   $0x9,0xfffffffc(%ebp)
 804838f:       7e 02                   jle    8048393 <main+0x1f>
 8048391:       eb 13                   jmp    80483a6 <main+0x32>
 8048393:       c7 04 24 84 84 04 08    movl   $0x8048484,(%esp)
 804839a:       e8 01 ff ff ff          call   80482a0 <printf@plt>
 804839f:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 80483a2:       ff 00                   incl   (%eax)
 80483a4:       eb e5                   jmp    804838b <main+0x17>
 80483a6:       c9                      leave
 80483a7:       c3                      ret
 80483a8:       90                      nop
 80483a9:       90                      nop
 80483aa:       90                      nop
reader@hacking:~/booksrc $
```

Exercise: do the mapping between C++ and Assembly

Understanding Programs Execution

Program execution

1. As a program executes, the EIP is set to the first instruction in the code segment
2. Reads the instruction that EIP is pointing to.
3. Adds the byte length of the instruction to EIP.
4. Executes the instruction that was read in step 2.
5. Goes back to step 2

Memory Segmentation

Memory Segmentation

Memory

- Electronic components used to record/maintain data in a computer
- An Operating System is responsible for the administration of these components
- Memory unit is a word of certain number of bits (32 bits)
- Every word has usually an associated address to reference it (32 bits)
- To manage the memory, The OS commonly subdivide it in segments
  - Every segment holds certain information for the execution of our program

Memory Segmentation

Memory Segments in Linux OS

- Text/Code Segment
- Data Segment
- BSS Segment
- Heap Segment
- Stack Segment

Code Segment

- The (assembler) Code is stored in the code segment

Data and BSS Segment

Data Segment

- It is filled with initialized global and static variables.
- fixed size

BSS Segment

- It is filled with uninitialized global and static variables.
- fixed size

Heap Segment

- A segment that programmer can directly control.
- It has variable size.
- All this memory is managed with allocators/deallocators

Stack Segment

Stack Segment

- It has variable size.
- Temporary scratch pad to store local function variables and context during function calls.
- (i.e. GDB's backtrace)
- First-in, Last-out (FILO) data structure
- When an item is placed (pushing), when an item is removed (popping)

Our focus in this tutorial

Stack Segment

Stack Segment

- It has variable size.
- Temporary scratch pad to store local function variables and context during function calls.
- (i.e. GDB's backtrace)
- First-in, Last-out (FILO) data structure
- When an item is placed (pushing), when an item is removed (popping)

Our focus in this tutorial

Now, let us focus on practical examples

Example

Indicate the segments where variables are stored in

```
int main(void)
{
    int a;
    int b;
    int c;
    int d;
    return 0;
}
```

Example

Indicate the segments where variables are stored in

```
int main(void)
{
    int a;
    int b;
    int c;
    int d;
    return 0;
}
```

Memory Segments

- Code is stored in the Code Segment
- Variables a, b, c, d in the Stack Segment

Example - Stack Segment

Indicate the segments where variables are stored in

```
void test_function(int a, int b, int c, int d){
    int flag; char buffer[4];
    flag = 31337;
    buffer[0] = 'A';
}
int main(void){
    test_function(1, 2, 3, 4)
}
```

Example - Stack Segment

Indicate the segments where variables are stored in

```
void test_function(int a, int b, int c, int d){
    int flag; char buffer[4];
    flag = 31337;
    buffer[0] = 'A';
}
int main(void){
    test_function(1, 2, 3, 4)
}
```

Memory Segments

- flag and buffer are stored in the Stack Segment

Example - Data Segment

Indicate the segments where variables are stored in

```
int main(void)
{
    global int x=2;
    static char y[3] = ['a', 'B', 'Z'];
    return 0;
}
```

Example - Data Segment

Indicate the segments where variables are stored in

```
int main(void)
{
    global int x=2;
    static char y[3] = ['a', 'B', 'Z'];
    return 0;
}
```

Memory Segments

- initialized static and global variables (i.e. x, y) are stored in the Data Segment

Example - BSS Segment

Indicate the segments where variables are stored in

```
int main(void)
{
    global int x;
    static char[3] y;
    return 0;
}
```

Example - BSS Segment

Indicate the segments where variables are stored in

```
int main(void)
{
    global int x;
    static char[3] y;
    return 0;
}
```

Memory Segments

- uninitialized static and global variables (i.e. *x*, *y*) are stored in the BSS Segment

Example - Heap Segment

Indicate the segments where variables are stored in

```
int main(void) {
    char *char_ptr;
    char_ptr = malloc(50);

    printf("pointer: %p", char_ptr);
    free(char_ptr);
}
```

Example - Heap Segment

Indicate the segments where variables are stored in

```
int main(void) {
    char *char_ptr;
    char_ptr = malloc(50);

    printf("pointer: %p", char_ptr);
    free(char_ptr);
}
```

Memory Segments

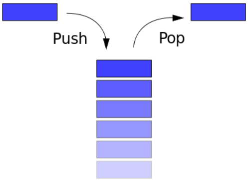
- char_ptr is stored in the stack
- char_ptr's content (*char_ptr is stored in the Heap seg.)

Outline

Stack

Stack Segment

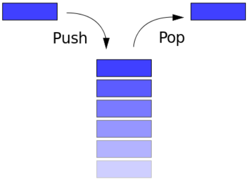
- Last-In First-Out stack
- Useful for context switching
- ebp (Stack Base Pointer): initial address of the stack
- esp (Stack Pointer): top address of the stack



Stack Operations

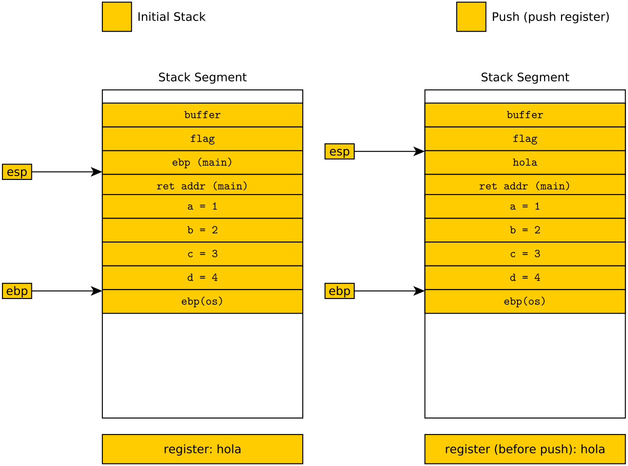
Operations

- *push* $<$ *register* $>$: decrements $esp - 4$ and places the content of *register* in the top of the stack (*esp*)
- *pop* $<$ *register* $>$: removes the content of *esp*, place it into the *register* and then increments $esp + 4$



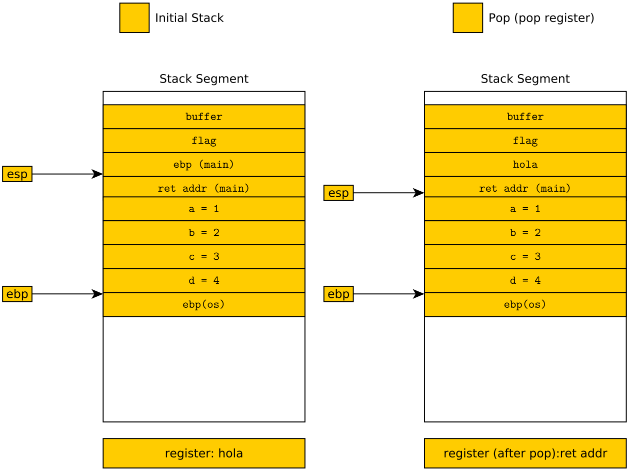
Stack Operations: Push

push < *register* >: decrements *esp* − 4 and places the content of *register* in the top of the stack (*esp*)



Stack Operations: Pop

pop < *register* >: removes the content of *esp*, place it into the *register* and then increments $esp + 4$



Context Switching

Definition

A context switching is the change from one process to another

Context Switching (execution of a function)

- Save Base Pointer (save ebp)
- Save parameters of the function in the stack
- Save return address

Remind...

- Every C application is composed of functions (i.e. int main)...

Content Switching: Example

```
void test_function(int a, int b, int c, int d){
    int flag; char buffer[4];
    flag = 31337;
    buffer[0] = 'A';
}
int main(void){
    test_function(1, 2, 3, 4)
}
```

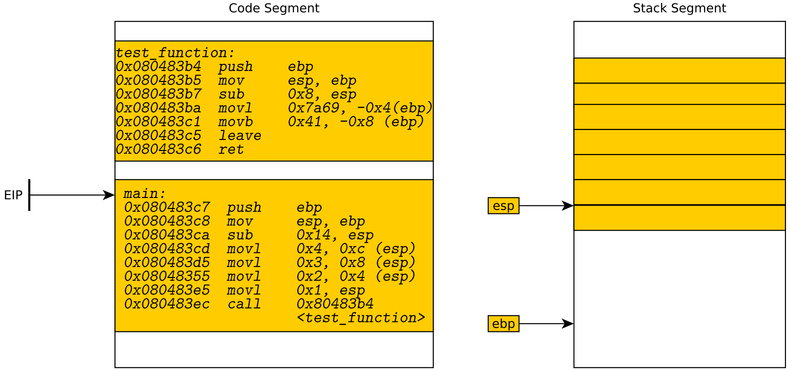
How do our computer execute this program?

Content Switching: Example

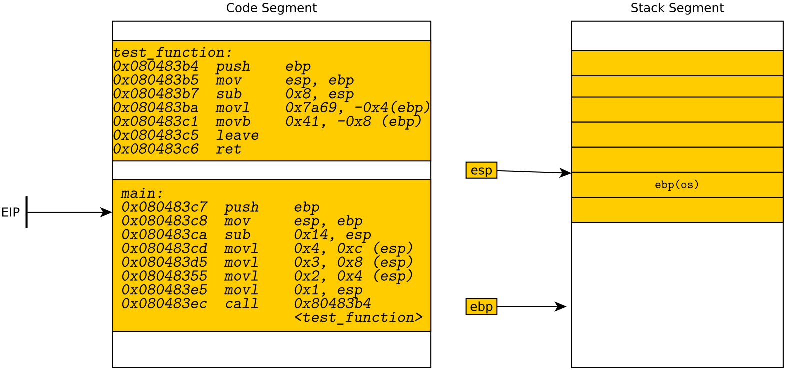
```
0xbffff5c4:     inc    %eax
(gdb) disass main
Dump of assembler code for function main:
0x080483c7 <main+0>:    push   %ebp
0x080483c8 <main+1>:    mov    %esp,%ebp
0x080483ca <main+3>:    sub    $0x14,%esp
0x080483cd <main+6>:    movl   $0x4,0xc(%esp)
0x080483d5 <main+14>:   movl   $0x3,0x8(%esp)
0x080483dd <main+22>:   movl   $0x2,0x4(%esp)
0x080483e5 <main+30>:   movl   $0x1,(%esp)
0x080483ec <main+37>:   call   0x80483b4 <test_function>
0x080483f1 <main+42>:   leave
0x080483f2 <main+43>:   ret
End of assembler dump.
(gdb) disass test_function
Dump of assembler code for function test_function:
0x080483b4 <test_function+0>:   push   %ebp
0x080483b5 <test_function+1>:   mov    %esp,%ebp
0x080483b7 <test_function+3>:   sub    $0x8,%esp
0x080483ba <test_function+6>:   movl   $0x7a69,-0x4(%ebp)
0x080483c1 <test_function+13>:  movb   $0x41,-0x8(%ebp)
0x080483c5 <test_function+17>:  leave
0x080483c6 <test_function+18>:  ret
End of assembler dump.
(gdb) _
```

Let us build the stack for this program

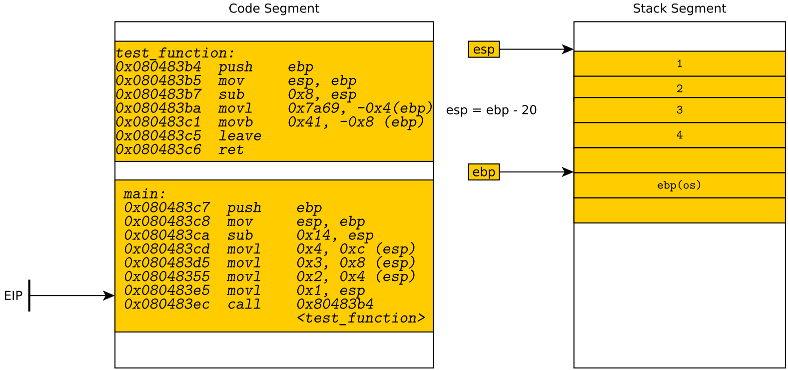
Content Switching: Building the Stack 1/7



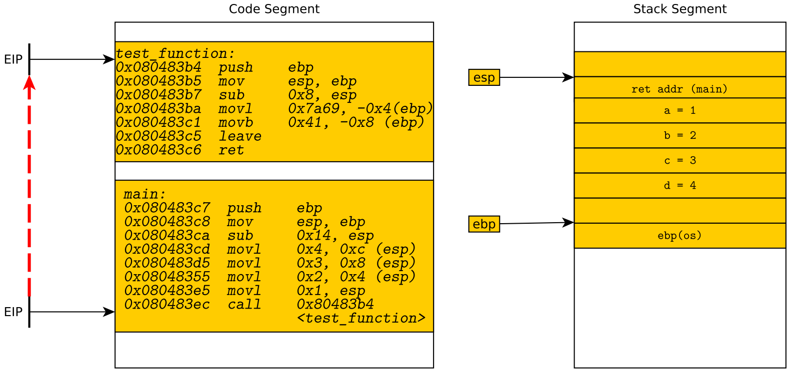
Content Switching: Building the Stack 2/7



Content Switching: Building the Stack 3/7

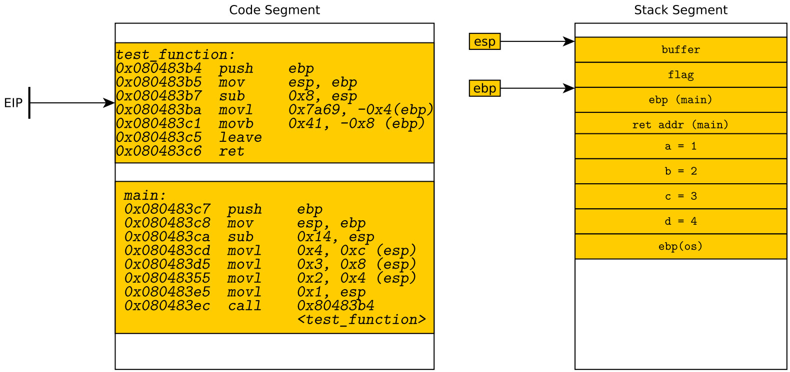


Content Switching: Building the Stack 4/7

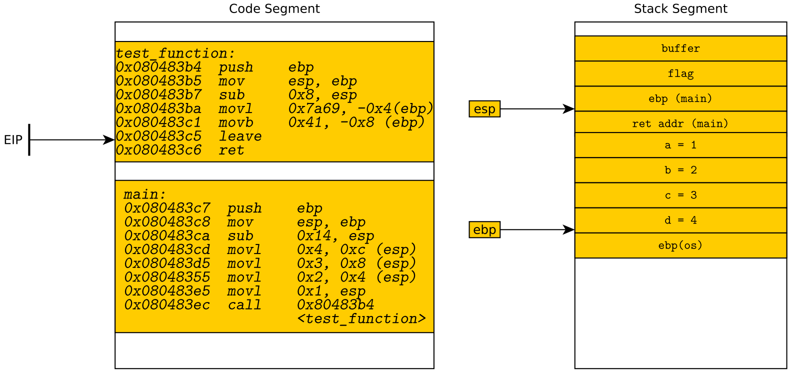


call $<$ *addr* $>$: stores return address into the stack and move EIP into the beginning pointed by the address.

Content Switching: Building the Stack 5/7

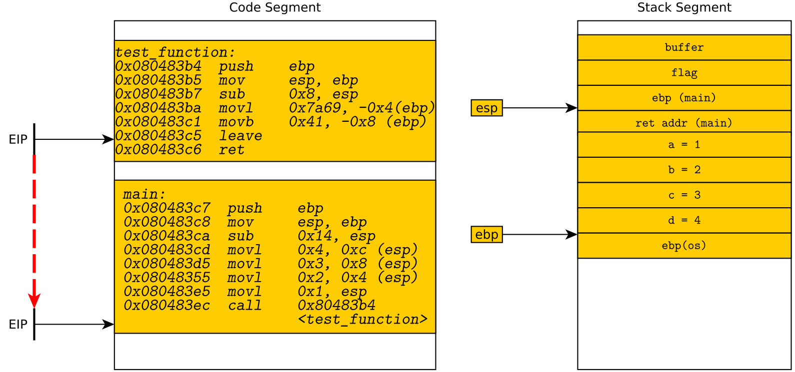


Content Switching: Building the Stack 6/7



leave: move ebp, esp; pop ebp (prepare stack for the return)

Content Switching: Building the Stack 7/7



ret: pop instruction pointer from the stack and make an inconditional jump to code segment.

Content Switching: Get your Hands dirty!

1. Open the virtual machine with VirtualBox (Gentoo 32 bits)
2. Get into InsecureProgramming folder
3. Type make to compile all the programs
4. Analyze and build the stack for *stack1.c* **on paper**
   - Check the source code of the program (stack1.c)
   - Run it on a debugger: *gdb stack1*; *disass main*
   - Make the mapping between the C and assembly code
   - Set a breakpoint into the main (b * [mem])
   - Check the registers (info reg $eip)
   - Using the *ni* instruction, follow the program step by step and build the stack (x/x [mem])

Outline

Common Programming Errors

Common Programming Errors

- Incorrect handling of buffer boundaries
  - Examples: gets() and strcpy() do not check buffer length.
- Do not sanitize end-users input data
  - Weird characters, characters instead of numbers, ...
- Do not sanitize filenames
  - Filenames could be used as program parameters
- Do not consider empty case

All these errors are commonly found in the Internet as ready-to-use code snipets

Common Programming Errors

Multipliers

- Quick modification to expand capabilities of a program
- Market Rules: as soon as possible
- Example of Microsoft ISS webserver
- Example Adobe Reader (PDF – 3D functionality)

Common Programming Errors

Check the following code

```
int main(void)
{
        int foo=0;
        foo = 1<<31;
        printf("%i ;", foo);
        foo--;
        printf("%i\n", foo);
        return 0;
}
```

Output: −2147483648; 2147483647. Why?

Outline

Exploitation

What is exploitation?

Exploiting a program is simply a clever way of getting the computer to do anything you want it to do.

Procedure

- Finding programmer errors
- Understand the code
- Alter normal program flow

Generalized Exploit Techniques

Motivation

- Same types of mistakes repeated over and over
- And when I mean over and over, it is millions of times

Exploit Techniques

- Most exploits related to memory corruption
- target is to take control of the target program's execution flow by running a piece of malicious code that has been smuggled into memory
- Search for unexpected cases that cause the program to crash
- We aim always at executing *arbitrary code*

Generalized Exploit Techniques

Exploit Techniques

- Buffer Overflow
- Buffer Stack Overflow
- Integer Overflow
- Format String
- and many more...

We focus on Buffer Overflow

Outline

Buffer Overflow

Precondition

- C assummes the programmer is responsible for *Data Integrity*
- Two-edges sword: no integrity check in exchange for velocity

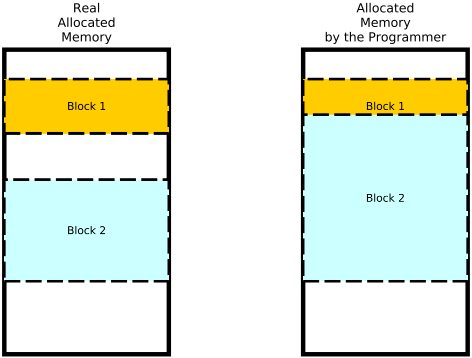
Target

- Allocate more data into a buffer that allocated previously less space
- If a critical piece of data is overwritten, the program will crash.

Buffer Overflow

Principle

- Developers forget to check variable's boundaries
- An Attacker overwrites memory in adjacent locations



Buffer Overflow: Our first Hack, step by step

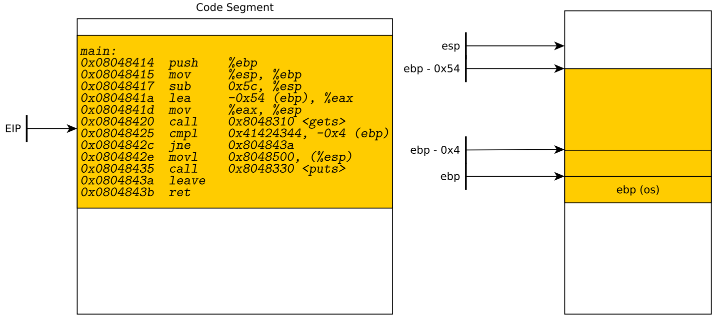
```
int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n",
        &buf, &cookie);
        gets(buf);

        if (cookie == 0x41424344)
                printf("you win!\n");
}
```

Buffer Overflow

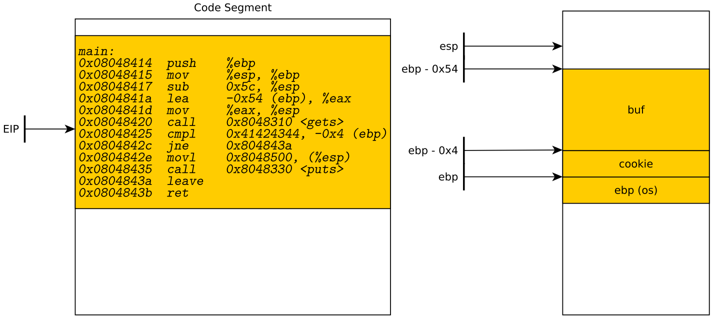
- How can we hack this program to print **you win!**?

Buffer Overflow: Our first Hack, step by step

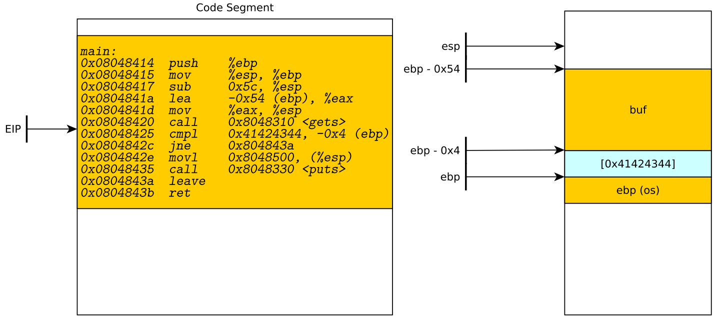
lea $< mem >< reg >$: places the address specified by first operand into the register specified into the second operand.



Buffer Overflow: Our first Hack, step by step



Buffer Overflow: Our first Hack, step by step



Outline

Buffer Stack Overflow: Principles

Now, we go for the stack

Precondition

- C assummes the programmer is responsible for *Data Integrity*
- Two-edges sword: no integrity check in exchange for velocity
- User has not control of the stack!

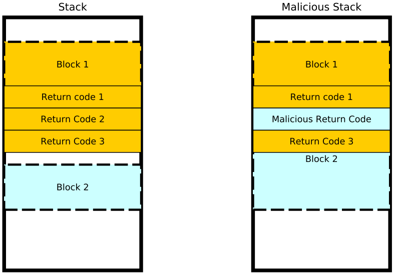
Target

- Allocate more data into a buffer that allocated previously less space
- We overwrite a critical pointer of the stack
- Full-knowledge of the memory organization

Buffer Stack Overflow: Principles

Principle

- Developers forget to check variable's boundaries
- An Attacker overwrites memory in adjacent locations
- The Attacker corrupts the stack to control the execution flow



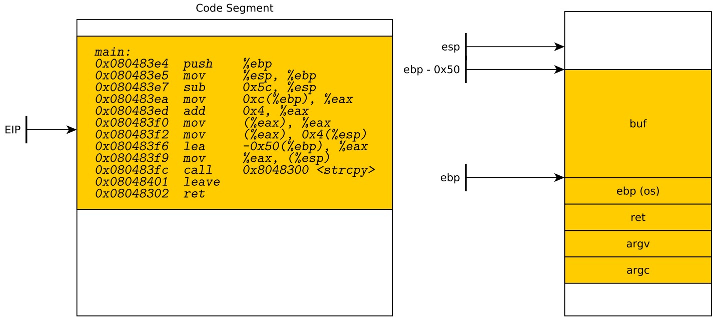
Buffer Stack Overflow: Abo #1

Advanced Buffer Overflow (Abo) #1

- How can we hack this program to print **you win!**?

```
int main(int argc,char **argv) {
        char buf[80];

        strcpy(buf,argv[1]);
}
```

Buffer Stack Overflow: Abo #1




```
main:
0x080483e4  push    %ebp
0x080483e5  mov     %esp, %ebp
0x080483e7  sub     0x5c, %esp
0x080483ea  mov     0xc(%ebp), %eax
0x080483ed  add     0x4, %eax
0x080483f0  mov     (%eax), %eax
0x080483f2  mov     (%eax), 0x4(%esp)
0x080483f6  lea     -0x50(%ebp), %eax
0x080483f9  mov     %eax, (%esp)
0x080483fc  call    0x8048300 <strcpy>
0x08048401  leave
0x08048302  ret
```

Buffer Stack Overflow: Abo #1

- Set a breakpoint at line 11
- Insert 4A into the memory and check the stack
  - x/x $ebp-0x54
- Insert AAAABBBB and check the stack
  - x/10i $ebp-0x54
- Insert 80*A + 4B and check the stack and cookie's value
  - x/x $ebp-0x4
- Put 80 $\times$ *A* and *ABCD* and check value of cookie
- Put 80 $\times$ *A* and *DCBA* and check value of cookie (endianess)
- Test without gdb

Suggested Literature

- http://phrack.org/issues/49/14.html
- http://phrack.org/issues/55/8.html
- https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Buffer-Overflows-Linux-wh pdf

Questions

THANK YOU