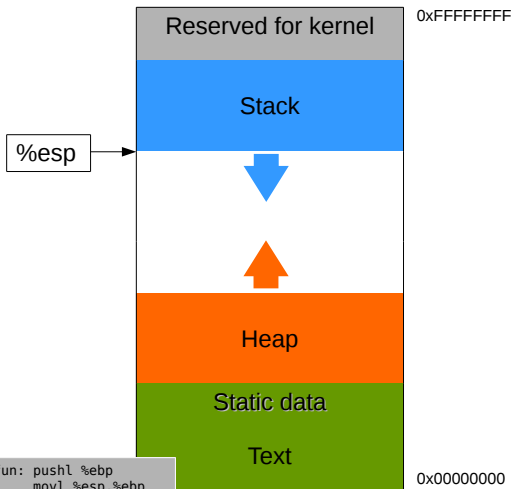


# Buffer overflows

Myrto Arapinis  
School of Informatics  
University of Edinburgh

November 06, 2017

# Linux (32-bit) process memory layout (simplified)

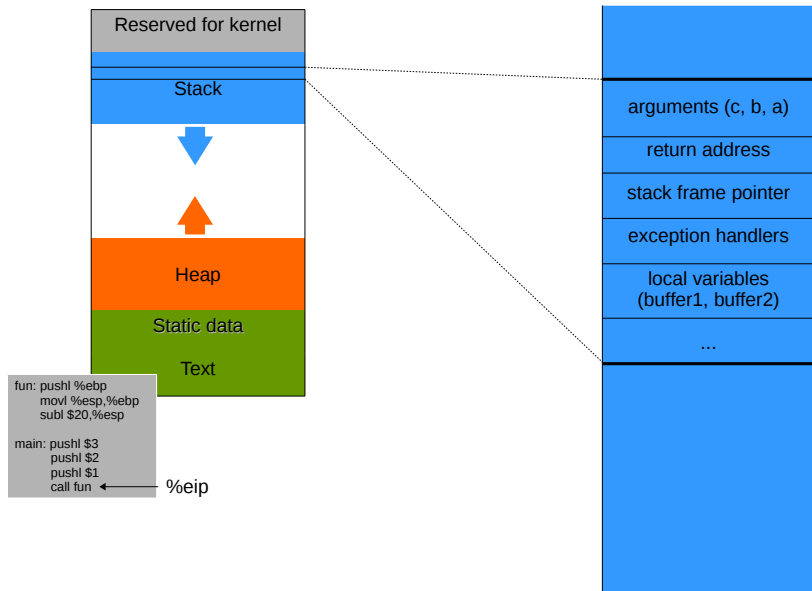


```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
fun: pushl %ebp
     movl %esp,%ebp
     subl $20,%esp

main: pushl $3
     pushl $2
     pushl $1
     call fun
```

# Stack frame



# Stack and functions: Summary

---

## Calling function

1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction to run after control returns
3. Jump to the function's address

## Called function

4. Push the old frame pointer onto the stack (`%ebp`)
5. Set frame pointer (`%ebp`) to where the end of the stack is right now (`%esp`)
6. Push local variables onto the stack

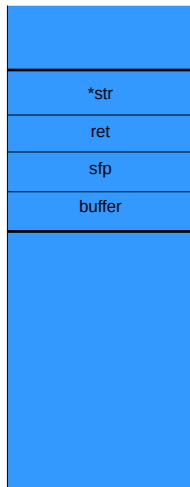
## Returning function

7. Reset the previous stack frame: `%esp = %ebp`, `%ebp = (%ebp)`
8. Jump back to return address: `%eip = 4(%esp)`

# Buffer overflows

---

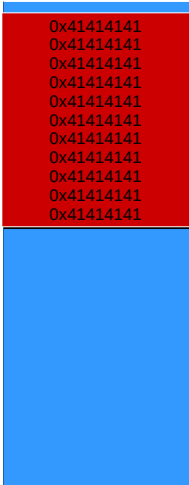
```
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}
void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```



# Buffer overflows

---

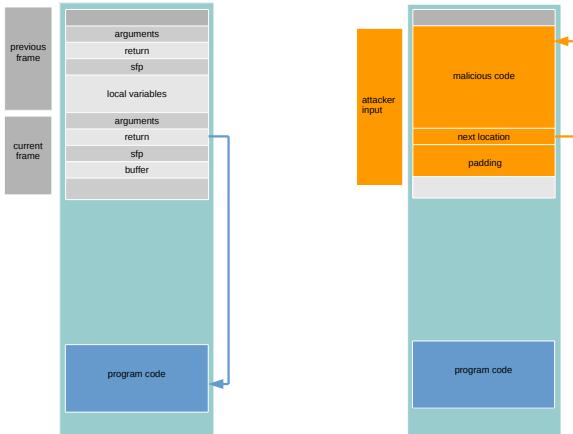
```
void function(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```



0x41414141  
0x41414141  
0x41414141  
0x41414141  
0x41414141  
0x41414141  
0x41414141  
0x41414141  
0x41414141  
0x41414141

strcpy(src, dest) does not check that dest is bigger than src  
The return address is now 0x41414141

# Control hijacking



A buffer overflow can change the flow of execution of the program:

- ▶ load malicious code into memory
- ▶ make `%eip` point to it

# Shellcode injection

---

**Goal:** “spawn a shell” - will give the attacker general access to the system

```
#include <stdio.h>
void main() {
char *name[2];
name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

C code

```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

Machine code

(part of attacker's input)

- ▶ must inject the machine code instructions (code ready to run)
- ▶ the code cannot contain any zero bytes (printf, gets, strcpy will stop copying)
- ▶ can't use the loader (we're injecting)

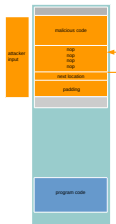


# The return address

---

**Challenge:** find the address of the injected malicious code?

- ▶ If code accessible: we know how far is the overflowed variable from the saved %ebp
- ▶ If code not accessible: try different possibilities!  
In a 32 bits memory space, there are  $2^{32}$  possibilities
- ▶ NOP sled
  - ▶ guess approximate stack state when the function is called
  - ▶ insert many NOPs before Shell Code



# Reference

---

Aleph One. Smashing The Stack For Fun And Profit.

<http://phrack.org/issues/49/14.html#article>

## Buffer overflow opportunities

# Unsafe libc functions

---

```
strcpy (char *dest, const char *src)
```

```
strcat (char *dest, const char *src)
```

```
gets (char *s)
```

```
scanf (const char *format, ...)
```

```
...
```

Do not check bounds of buffers they manipulate!!

# Integer overflows

---

[Ref] Blexim. Basic Integer Overflows

<http://phrack.org/issues/60/10.html#article>

Attempt to store a value in an integer which is greater than the maximum value the integer can hold

→ the value will be truncated

Example

```
# include <stdio.h>
int main(void){
    unsigned int num = 0xffffffff;
    printf('num + 1 = 0x%x\n', num + 1);
    return 0;
}
```

The output of this program is:  $\text{num} + 1 = 0x0$

# Integer overflow exploit (1)

---

```
int catvars(char *buf1, char *buf2,
            unsigned int len1, unsigned int len2){
    char mybuf[256];
    if((len1 + len2) > 256){
        return -1;
    }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);
    do_some_stuff(mybuf);
    return 0;
}
```

# Integer overflow exploit (1)

---

```
int catvars(char *buf1, char *buf2,
            unsigned int len1, unsigned int len2){
    char mybuf[256];
    if((len1 + len2) > 256){
        return -1;
    }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);
    do_some_stuff(mybuf);
    return 0;
}
```

**Check can be bypassed by using suitable values for len1 and len2: len1 = 0x104, len2 = 0xffffffc, len1+len2 = 0x100 (decimal 256)**

## Integer overflow exploit (2)

---

```
int myfunction(int *array, int len){
    int *myarray, i;
    myarray = malloc(len * sizeof(int));
    if(myarray == NULL){
        return -1;
    }
    for(i = 0; i < len; i++){
        myarray[i] = array[i];
    }
    return myarray;
}
```



## Integer overflow exploit (2)

---

```
int myfunction(int *array, int len){
    int *myarray, i;
    myarray = malloc(len * sizeof(int));
    if(myarray == NULL){
        return -1;
    }
    for(i = 0; i < len; i++){
        myarray[i] = array[i];
    }
    return myarray;
}
```

**Can allocate a size 0 buffer for myarray by using suitable value for len: len = 1073741824 , sizeof(int) = 4, len\*sizeof(int) = 0**

# Format strings (1)

---

[Ref] scut/team teso. Exploiting Format String Vulnerabilities

- ▶ A format function takes a variable number of arguments, from which one is the so called format string

Examples: `fprintf`, `printf`, ..., `syslog`, ...

# Format strings (1)

---

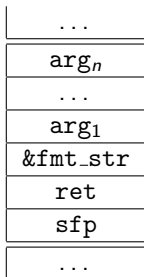
[Ref] scut/team teso. Exploiting Format String Vulnerabilities

- ▶ A format function takes a variable number of arguments, from which one is the so called format string

Examples: `fprintf`, `printf`, ..., `syslog`, ...

- ▶ The behaviour of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack

Example: `printf(fmt_str, arg1, ..., argn);`



## Format strings (2)

---

- ▶ If an attacker is able to provide the format string to a format function, a format string vulnerability is present

```
int vulnerable(char *user) {  
    printf(user);  
}
```

```
int safe(char *user){  
    printf ("%s", user);  
}
```

## Example: printf

---

```
printf('Num %d has no address, num %d has:%08x\n', i, a,&a);
```

...	
<&a>	address of variable a
<a>	value of variable a
<i>	value of variable i
&fmt_str	address of the format string
ret	
sfp	
...	

# Format strings exploits

---

- ▶ We can view the stack memory at any location
  - ▶ walk up stack until target pointer found
  - ▶ `printf (' '%08x.%08x.%08x.%08x.%08x|s|'');`
- ▶ We can write to any memory location
  - ▶ `printf('hello %n', &temp) - writes '6' into temp`
  - ▶ `printf('hello%08x.%08x.%08x.%08x.%n')`

# More buffer overflow opportunities

---

- ▶ Exception handlers
- ▶ Function pointers
- ▶ Double free
- ▶ ...

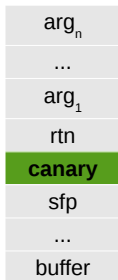
Defenses against buffer overflows:  
making exploitation hard



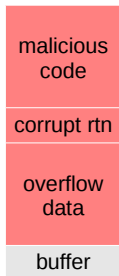
# Stack canaries

---

- ▶ detect a stack buffer overflow before execution of malicious code
- ▶ place a small integer (canary) just before the stack return pointer
- ▶ to overwrite the return pointer the canary value must also be overwritten
- ▶ the canary is checked to make sure it has not changed before a routine uses the return pointer on the stack



safe stack



corrupted  
stack

# Canary values

---

[Ref] Cowan & al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Symposium, 1998

1. **Terminator canaries** (CR, LF, NUL (i.e., 0), -1): `scanf` etc. do not allow these values
2. **Random canaries**
  - ▶ Write a new random value at each process start
  - ▶ Save the real value somewhere in memory
  - ▶ Must write-protect the stored value
3. **Random XOR canaries**
  - ▶ Same as random canaries
  - ▶ But store canary XOR some control info, instead

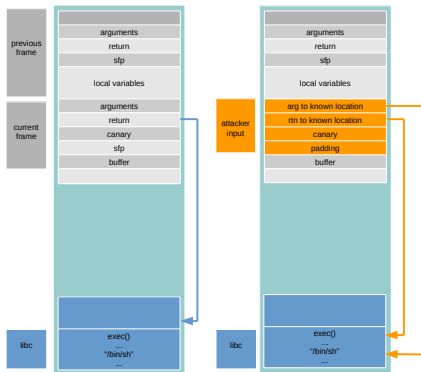
# Make stack and heap non executable

---

- ▶ **Goal:** even if the canary is bypassed, the malicious code loaded cannot be executed

# Make stack and heap non executable

- ▶ **Goal:** even if the canary is bypassed, the malicious code loaded cannot be executed
- ▶ **But:** vulnerable to return-to-libc attack!!
  - ▶ the libc library is linked to most C programs
  - ▶ libc provides useful calls for an attacker



# Address space layout randomization

---

- ▶ **Idea:** place standard libraries to random locations in memory  
—→ for each program, `exec()` is situated at a different location  
—→ the attacker cannot directly point to `exec()`
- ▶ Supported by most operating systems (Linux, Windows, MAC OS, Android, iOS, ...)