Buffer and Stack Overflow

Security 1 (CM0475, CM0493) 2020-21
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Introduction

Buffer overflow is one of the **most common vulnerabilities**

- caused by “**careless**” programming
- known **since 1988** but still present

Introduction

Why still there ...

Can be avoided, in principle, by writing ***secure code***

- non-trivial in “**unsafe**” languages, e.g., C
- **legacy** application/systems might have overflows

⇒ **mitigation** mechanisms are important!

Brief history of some famous overflows

1988 The **Morris Internet Worm** used a buffer overflow exploit in **`fingerd`**

1995 A buffer overflow in **httpd 1.3** was discovered and published on the Bugtraq mailing list

1996 "*Smashing the Stack for Fun and Profit*" in Phrack magazine (a step by step **introduction**)

2001 Code Red worm exploited a buffer overflow in **Microsoft IIS 5.0**

2003 Slammer worm exploited a buffer overflow in **Microsoft SQL Server 2000**

2004 Sasser worm exploited an overflow in Microsoft Windows 2000/XP, **Local Security Authority Subsystem Service** (LSASS).

Definition

A buffer **overflow** (**overrun** or **overwrite**), is defined as follows [NISTIR 7298]:

> A condition at an interface under which **more input** can be placed into a buffer or data holding area **than the capacity allocated**, **overwriting** other information.
>
> Attackers **exploit** such a condition to
>
> - **crash** a system
> - insert specially crafted **data** that break integrity
> - insert specially crafted **code** to gain control of the system

Consequences

Example: a program storing data **beyond the limits** of a fixed-sized buffer

Buffer can be located

- on the **stack**
- in the **heap**
- in the **data section**

Effects:

- modify other variables (**corruption of data**)
- modify the program control flow data such as return addresses and pointers to previous stack frames (**corruption of control**)

⇒ arbitrary **code execution** with the **privileges** of the attacked process

Safe vs. unsafe languages (1)

Assembly is fast but does not provide any notion of type

- 👍 **full access** to resources
- 👍 high **performance**
- 👎 data can be **interpreted** and used in any way
- 👎 programmer's **responsibility** to enforce safe execution

Languages such as **Java**, **ADA**, **Python** are safer

- 👍 strong notion of **types**
- 👍 overflows are **not possible**
- 👎 usually limited/missing **direct access** to resources
- 👎 compile-time and run-time **overhead**

Safe vs. unsafe languages (2)

C is in between!

Like Assembly:

👍 **full access** to resources

👍 high **performance**

⇒ used to develop Unix. Still the preferred language for **low-level programming** (OS, device drivers, firmware, ...)

Differently from ADA, Java, Python, C has **weak types**

👎 low-level, **unsafe access** to data is possible

👎 programmer’s **responsibility** to enforce safe execution in many cases (e.g. overflows are possible)

👎 many **unsafe library functions**

Example: simple overflow

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

void checkpassword() {
      int valid = 0;
      char str1[8]; // 7 chars + NULL
      char str2[8]; // 7 chars + NULL
      strcpy(str1,"pwd1234"); // a secret pwd
      printf("Insert password: ");
      fflush(stdout);
      gets(str2); // reads the user password
      // compares 8 chars of str1 and str2
      if (strncmp(str1, str2, 8) == 0)
            valid = 1; // password is valid
      printf("buffer1: str1(%s), str2(%s),
            valid(%d)\n", str1, str2, valid);
}

int main(int argc, char *argv[]) {
      checkpassword();
}
```

`str1` and `str2` are two buffers of 8 bytes (1 byte for NULL termination)

`str1` contains, at run-time, the secret password `"pwd1234"`

`str2` is used to read user input

first 8 bytes of `str1` and `str2` are compared and if equal valid is set to `1` (true)

Example: simple overflow

```
# ./overflow
Insert password: AAAAAAA
buffer1: str1(pwd1234), str2(AAAAAAA), valid(0)             Password is wrong

# ./overflow
Insert password: pwd1234
buffer1: str1(pwd1234), str2(pwd1234), valid(1)             Password is correct

# ./overflow
Insert password: AAAAAAAA                                   (8 chars)
buffer1: str1(), str2(AAAAAAAA), valid(0)                   0x00 (NULL) overflows first byte of str1

# ./overflow
Insert password: AAAAAAAAAAAAAAA                            (15 chars)
buffer1: str1(AAAAAAA), str2(AAAAAAAAAAAAAAA), valid(0)     7 A's and 0x00 overflows str1

# ./overflow
Insert password: AAAAAAAAAAAAAAAA                           (16 chars)
buffer1: str1(AAAAAAAA), str2(AAAAAAAAAAAAAAAA), valid(1)   Password is correct! strncmp(str1,str2,8)
```

Unsafe C functions

```
# gcc overflow.c -o overflow
overflow.c: In function 'checkpassword':
overflow.c:17:2: warning: implicit declaration of function 'gets'; did you mean 'getw'?
[-Wimplicit-function-declaration]
  gets(str1); // reads the user password
  ^~~~
  getw
```

Function `gets` is **unsafe** and **<u>should never be used</u>** (cannot limit user input!)

Note 1: `gets` has been removed from stdio.h, so compiling gives a warning but program works anyway (**legacy** code needs to be supported)

Note 2: `strcpy` is unsafe too, but it is still in stdio.h (no warning). In this case, since`"pwd1234"` fits the 8 bytes we do not get any security warning.

Stack overflow

A buffer overflow **occurring on the stack,** also known as ***stack smashing***

Right after the local variables, the stack contains

- The old **frame pointer**
- The **return address**

A stack overflow can overwrite these control data to run **arbitrary code**

Function call

Calling function:

1. push **parameters**
2. call (pushes the **return address**)

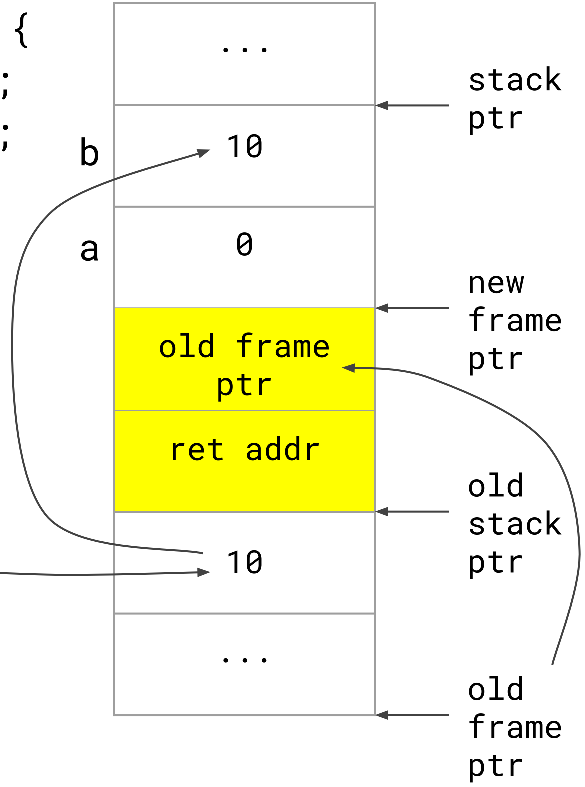
Called function:

3. push **old frame pointer**
4. **new frame pointer** is set where the stack pointer is
5. **stack pointer** is decreased so to allocate <u>local variables</u>
6. **parameters** are accessed

```
int f (int x) {
    int a = 0;
    int b = x;
    …
}
```

Invocation:

```
    f(10);
```



Function return

When f returns, it

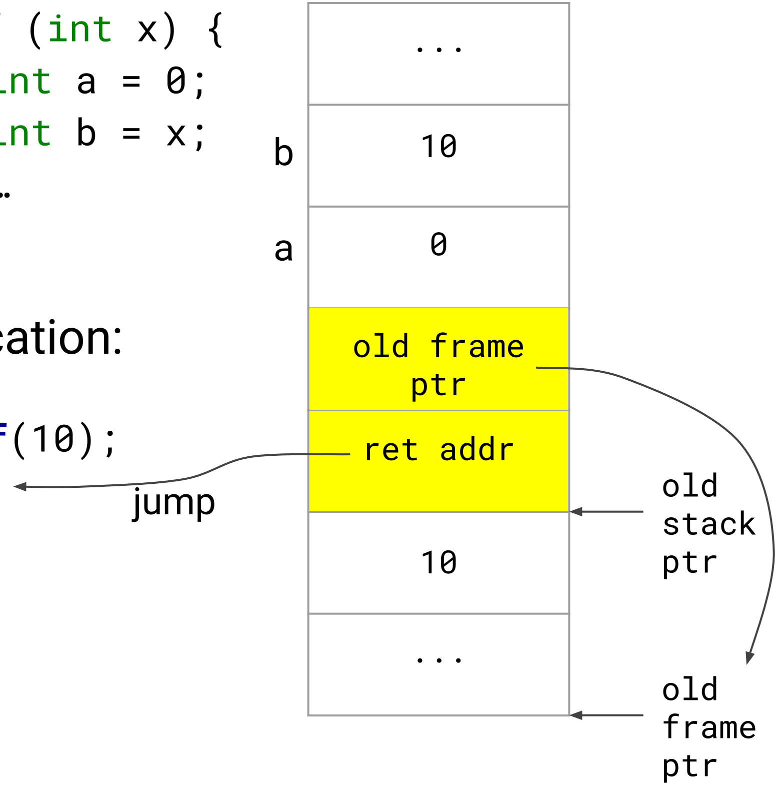
1. sets the **stack pointer** to the old frame pointer position
2. pops and **restores** the **old frame pointer**
3. return (pops and **jumps** to the **return address**)

⇒ If an overflow overwrites the return address, the **control** goes to the new address (possibly malicious)

```
int f (int x) {
    int a = 0;
    int b = x;
    …
}
```

Invocation:

```
    f(10);
```



Example: stack overflow

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

void hiddenfunction() {
     printf("This will never be reached!\n");
}

void checkpassword() {
     ...
     // same code as before with overflow
     ...
}

int main(int argc, char *argv[]) {
     checkpassword();
}
```

Suppose our previous example program contains a function that is **not even invoked** (`hiddenfunction`)

Assume that `hiddenfunction` is **located** at `0x00005555555551da`

(because of **little-endianness** we will need to pass the address in reverse order as bytes `0xda 0x51 0x55 0x55 0x55 0x55 0x00 0x00`)

Example: stack overflow

```
# ./stack
Insert password: AAAAAAAAAAAAAAAAAAA
buffer1: str1(AAAAAAAA), str2(AAAAAAAAAAAAAAAA), valid(1)
```

(19 chars)

str2 is 16 chars! valid is overwriting it

```
# ./stack
Insert password: AAAAAAAAAAAAAAAAAAAA
buffer1: str1(AAAAAAAA), str2(AAAAAAAAAAAAAAAA), valid(1)
Segmentation fault
```

(20 chars)

Segfault!

```
# echo -e 'AAAAAAAAAAAAAAAAAAAA\xda\x51\x55\x55\x55\x55\x00\x00' | ./stack
Insert password: buffer1: str1(AAAAAAAA), str2(AAAAAAAAAAAAAAAA), valid(1)
Segmentation fault
```

Old frame pointer but not return address!

```
# echo -e 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA\xda\x51\x55\x55\x55\x55\x00\x00' | ./stack
Insert password: buffer1: str1(AAAAAAAA), str2(AAAAAAAAAAAAAAAA), valid(1)
This will never be reached!
Segmentation fault
```

Old frame pointer and **return address**!

More C unsafe functions

```
sprintf(char *str,char *format,...)
```

create `str` according to supplied `format` and variables `...`

```
vsprintf(char *str, char *format,
         va_list vars)
```

create `str` according to supplied `format` and variables `vars`

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

append contents of string `src` to string `dest`

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

copy contents of string `src` to `dest`

… and their “safe” counterpart

```
snprintf(char *str, size_t size,
         char *format,...)
```

same as `sprintf` but writes at most `size` chars (including `0x00`)

```
vsnprintf(char *str, size_t size,
         char *format, va_list vars)
```

same as `vsprintf` but writes at most `size` chars (including `0x00`)

```
strncat(char *dest, char *src,
        size_t size)
```

same as `strcat` but appends at most `size` chars (excluding `0x00`)

`dest` size at least: `strlen(dest)+size+1`

```
strncpy(char *dest, char *src,
        size_t size)
```

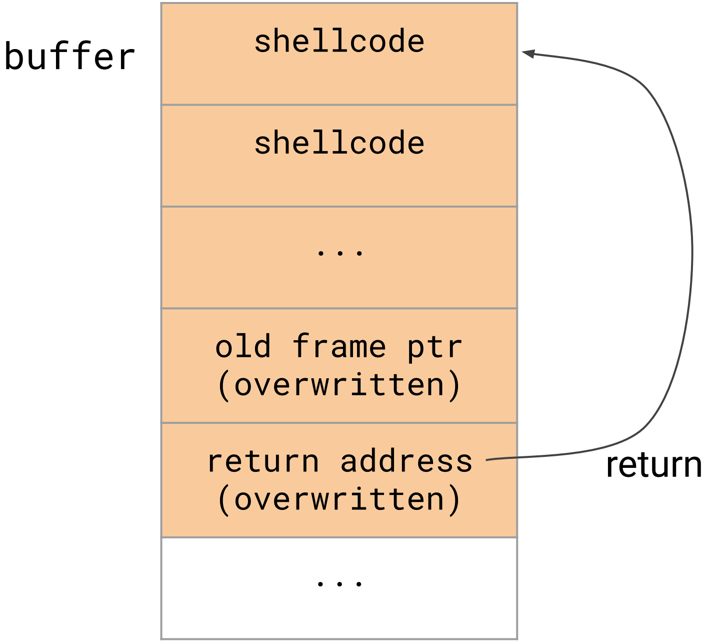
same as above but it does not add `0x00` if `src` is cut to n!

Shellcodes

Definition: small binary program that executes a **shell** (or arbitrary code)

- **small** so to fit the buffer
- **position independent**
- **null byte** (0x00) free (in case overflow is over string operations)
- **library** independent

⇒ inject on the stack and **return to it**

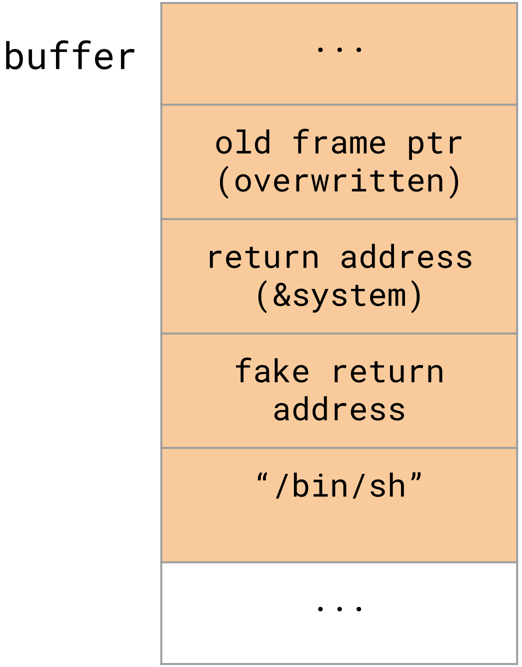


Return to syscall / libc

Idea: return to existing syscalls or library functions

- overwrite a “reasonable” old frame pointer
- write function address over **return address**
- write a **fake return address**
- write function **parameters**

⇒ function will read **parameters** from the stack and **execute** (cf. function call slide)



“invokes” `system(“/bin/sh”)`

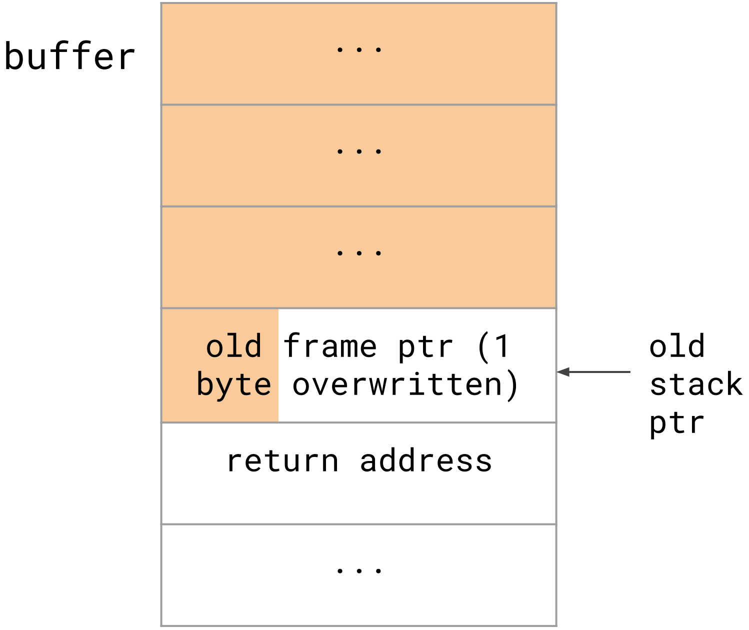
Replacement stack frame

off-by-one: a subtle overflow of a single byte (<= instead of <)

- **too short** to overwrite return address

IDEA: overwrite **a single byte** of old frame pointer

- stack frame is moved to an area controlled by the attacker so that **next** return address is malicious

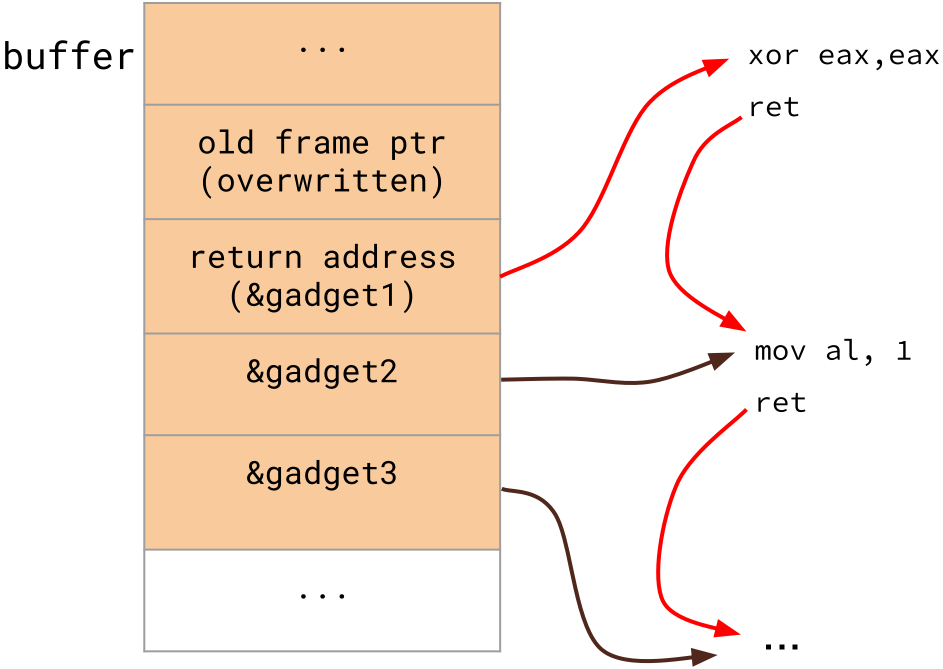


Return Oriented Programming (ROP)

Idea: return to fragment of codes close to return commands (***gadgets***)

- overwrite return address with a **sequence** of gadget addresses
- when function returns it will **activate** the first gadget that will activate the second, and so on...

⇒ malicious code as the **composition of gadgets** (e.g., starting a shell)



Defences

Compile-time: **harden** programs to resist to overflow attacks (important for new programs)

Run-time: **detect** and **block** attacks on existing programs

Compile-time defences

Use safe programming languages: use **unsafe** languages only if **strictly necessary** (access to hardware, extreme performance). **Low-level libraries** might be vulnerable though

Safe coding techniques: always check buffer boundaries, use safe library functions; ***graceful failure*** when unexpected occurs. (more detail in next class)

Stack protection: Compiler

- **adds extra code** to look for stack corruption. StackGuard uses a random ***canary*** value that is pushed after old frame pointer and **checked** before return

- **rearranges** variable position so that buffers are the **last ones** on the stack (mitigates overflows)

Canary (1)

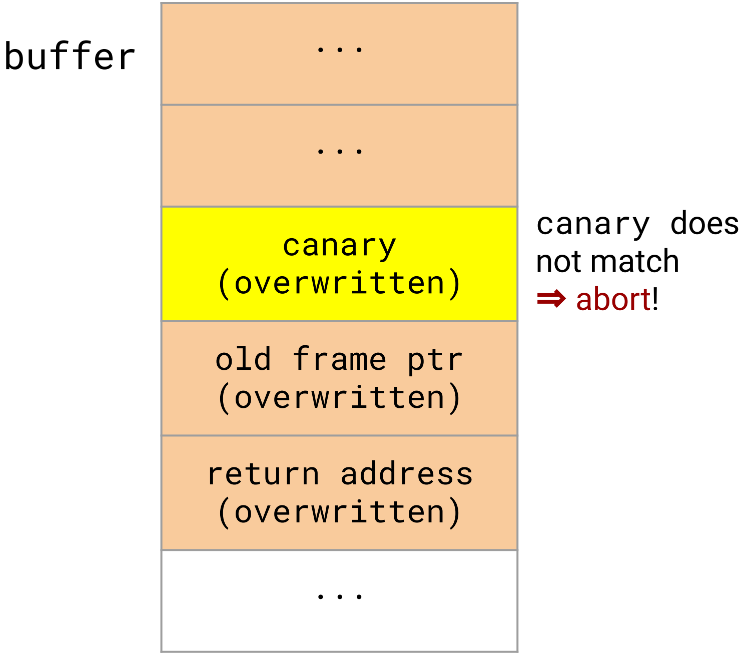
Requires operating system support

When function **starts**:

- random canary is **copied** to the stack from the process table

Before function **returns**:

- original canary is **compared** with the one of the stack and, if different, the function **aborts**



Canary (2)

```
# Read random canary from the process
mov     rax,QWORD PTR fs:0x28

# Copy canary on the stack
mov     QWORD PTR [rbp-0x8],rax

(function code)

# Reads canary  from the stack
mov     rax,QWORD PTR [rbp-0x8]

# Compares with process canary
xor     rax,QWORD PTR fs:0x28

# If OK go to return else fail
je      0x12b1 <checkpassword+180>
call    0x1060 <__stack_chk_fail@plt>
```

👍 **very effective** prevention of overflows but requires re-compiling programs

👎 void if canary is **leaked** (for example due to another vulnerability)

👎 void in case of **random access** to the stack (eg. overflowing a buffer index)

Run-time defences

Executable address space protection: prevent **execution** of code in particular segments (e.g. stack, heap, …). Requires **hardware** support.

👍 prevents **shellcodes**

👎 does not prevent return to **syscall, libc, ROP**

👎 some programs need to **disable** it (they execute code on the stack)

Address space randomization: randomize address space in order to make it harder to discover addresses

👍 make **overflow attacks** much harder (what return address??)

👎 bypassed if attacker can **brute-force**

👎 bypassed if addresses are **leaked** (e.g. recent side-channels attacks)

Run-time defences

Guard pages: memory pages that are not accessible, placed in between regular pages

👍 prevents cross-page overflows

👍 isolate memory regions

👎 cannot prevent an intra-page stack overflow

Control flow integrity: control program control flow, preventing jumps to arbitrary functions

Example: Microsoft Control Flow Guard (**CFG**). When a function is called the system checks if it belongs to a permitted set.

👎 **performance** issues

👎 many **bypasses** so far