Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Software Security (Part 1)

Hugo Pacheco
hpacheco@fc.up.pt

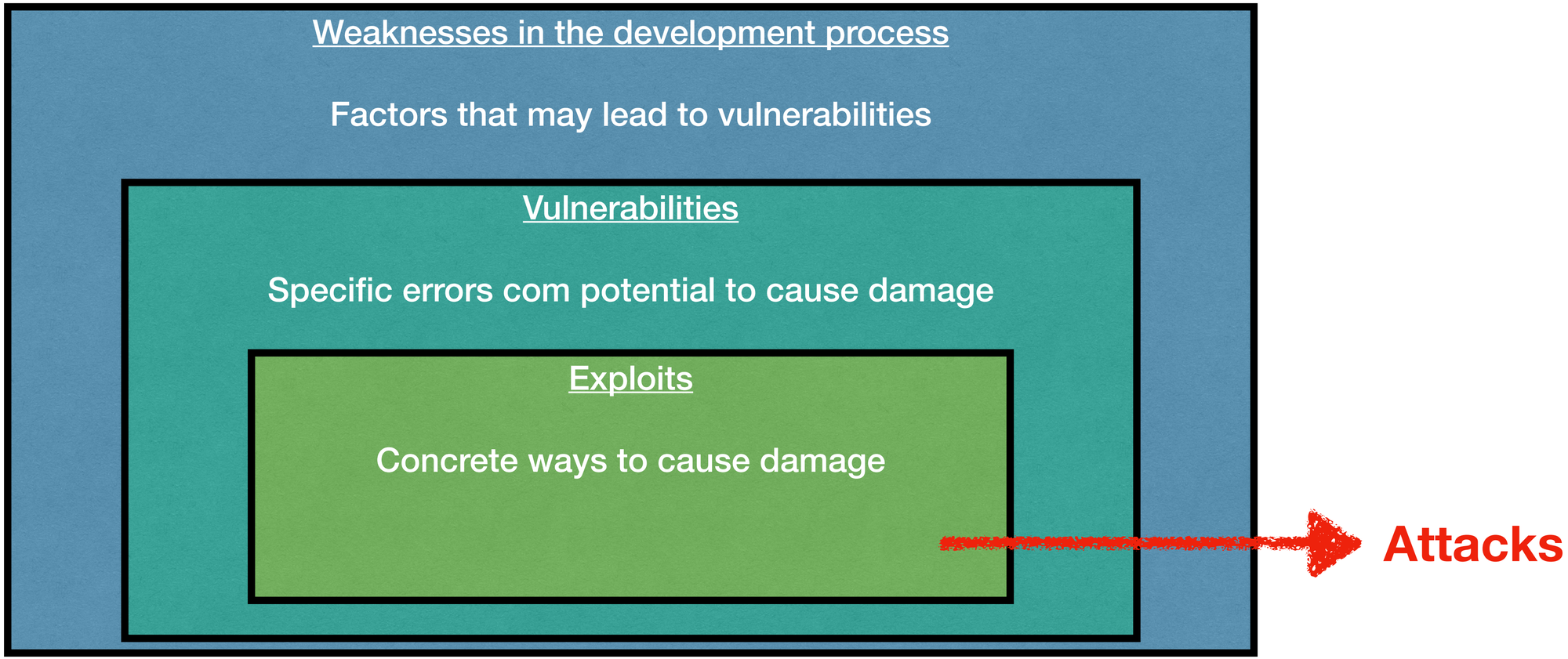
The Creature vs The Creator

Software

Us

The Creature vs The Creator

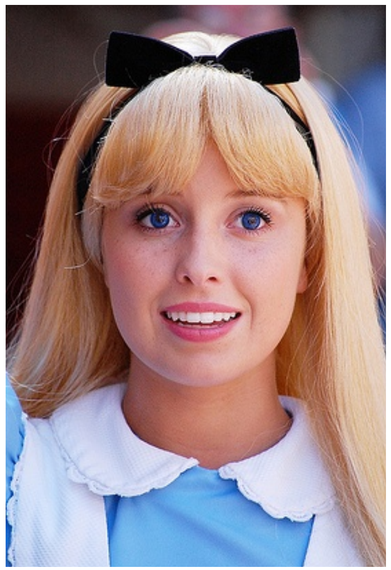
- Software can be manipulated to turn against us

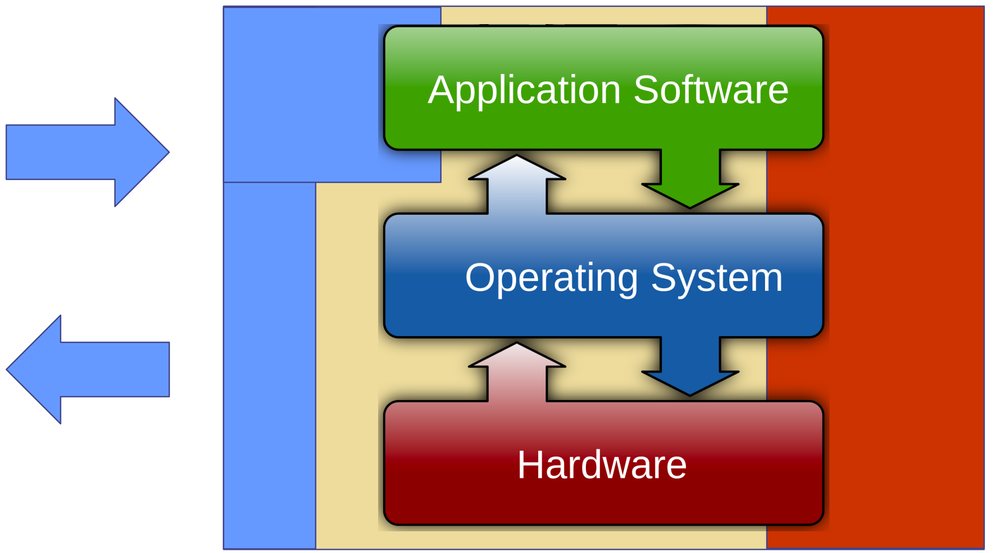


Learning with history

- It is important to study vulnerabilities and previous attacks
  - Identifying possible vulnerabilities and solutions
  - Incentive for fixing:
    - knowledge + flaws = negligence
  - Characterising threats:
    - Assessing the severity of vulnerabilities in a specific context
    - Help users assessing risk

Software Security





Alice

Mallory

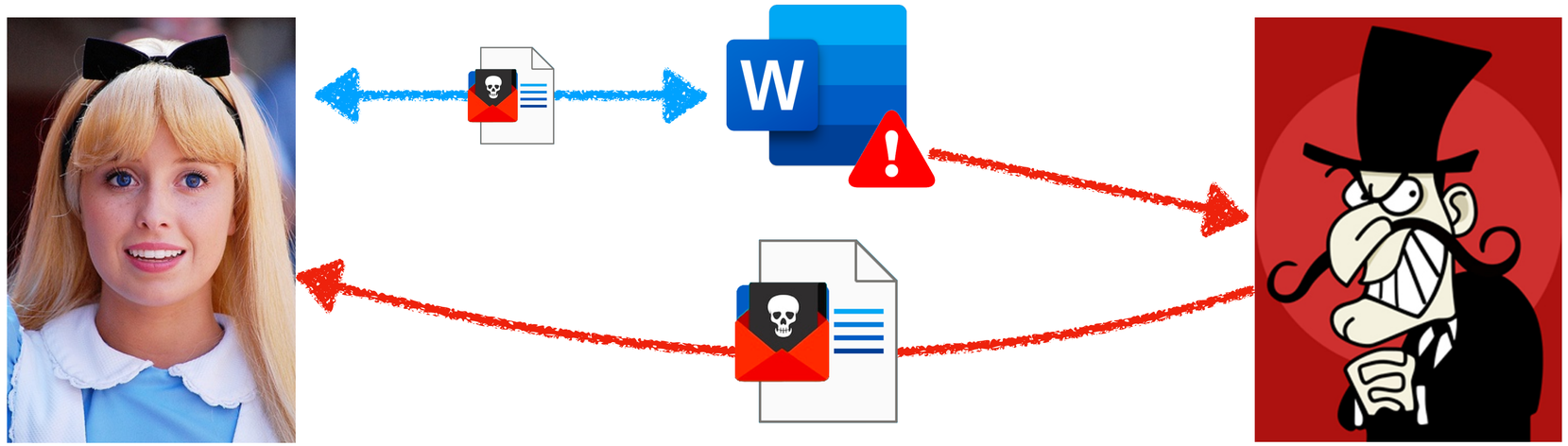
- Honest user (Alice) interacts with a software application
- Malicious attacker (Mallory) tries to exploit a software (or OS/HW) vulnerability
- Exploiting programming errors / language / compiler / execution environment

Software Security

- Software executes in a computational platform
  - Dynamic (memory) and persistent (disk) storage
  - Operating System
  - Hardware + Interfaces with external devices
- It is always a potential weapon against that platform:
  - It may allow to **gain control** over that platform
  - Can then be used for several purposes: DoS, SpyWare, Jailbreak, etc.

Why gaining control?

Is crashing MS Word a security vulnerability?



- Many software vulnerabilities are violations of “control flow integrity”: attacker alters the control flow of the program to execute his code in our machine

- Threat model: user code receives input from outside the security perimeter

[Microsoft Word CVEs: https://www.cvedetails.com/product/529/Microsoft-Word.html?vendor_id=26]

The infamous “buffer overflow” & friends

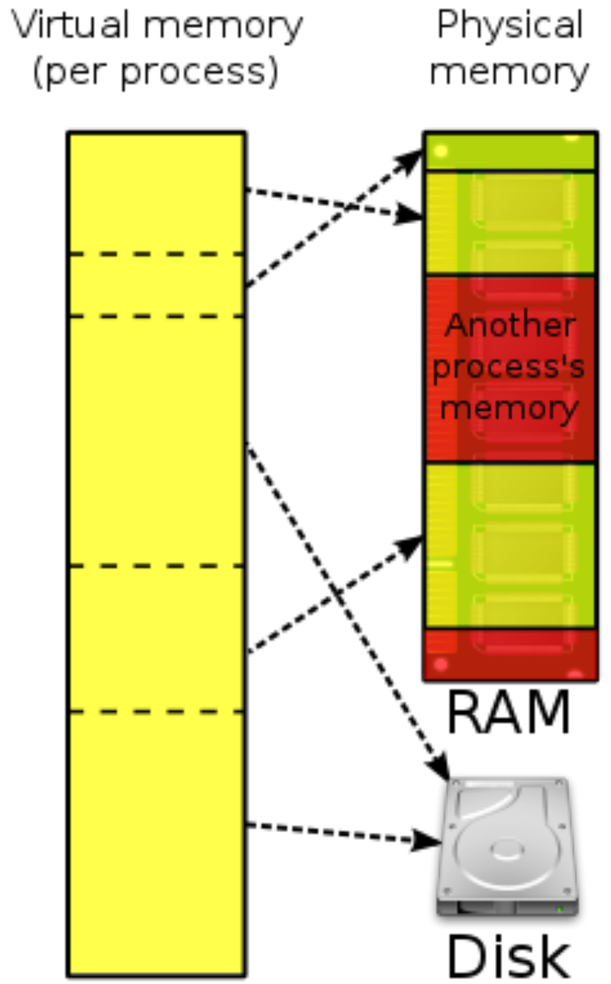
- One of the simplest (conceptually!) examples of gaining of control; belongs to a large class of memory management errors in (not only) C
- When the program “accesses outside” of its allocated memory

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2022
1	CWE-787	Out-of-bounds Write	63.72	70	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.54	4	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	34.27	6	0
4	CWE-416	Use After Free	16.71	44	+3
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	15.65	23	+1
6	CWE-20	Improper Input Validation	15.50	35	-2
7	CWE-125	Out-of-bounds Read	14.60	2	-2
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.11	16	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.73	0	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.41	5	0
11	CWE-862	Missing Authorization	6.90	0	+5
12	CWE-476	NULL Pointer Dereference	6.59	0	-1
13	CWE-287	Improper Authentication	6.39	10	+1
14	CWE-190	Integer Overflow or Wraparound	5.89	4	-1

[2023 CWE Top 25: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html]

Memory Model

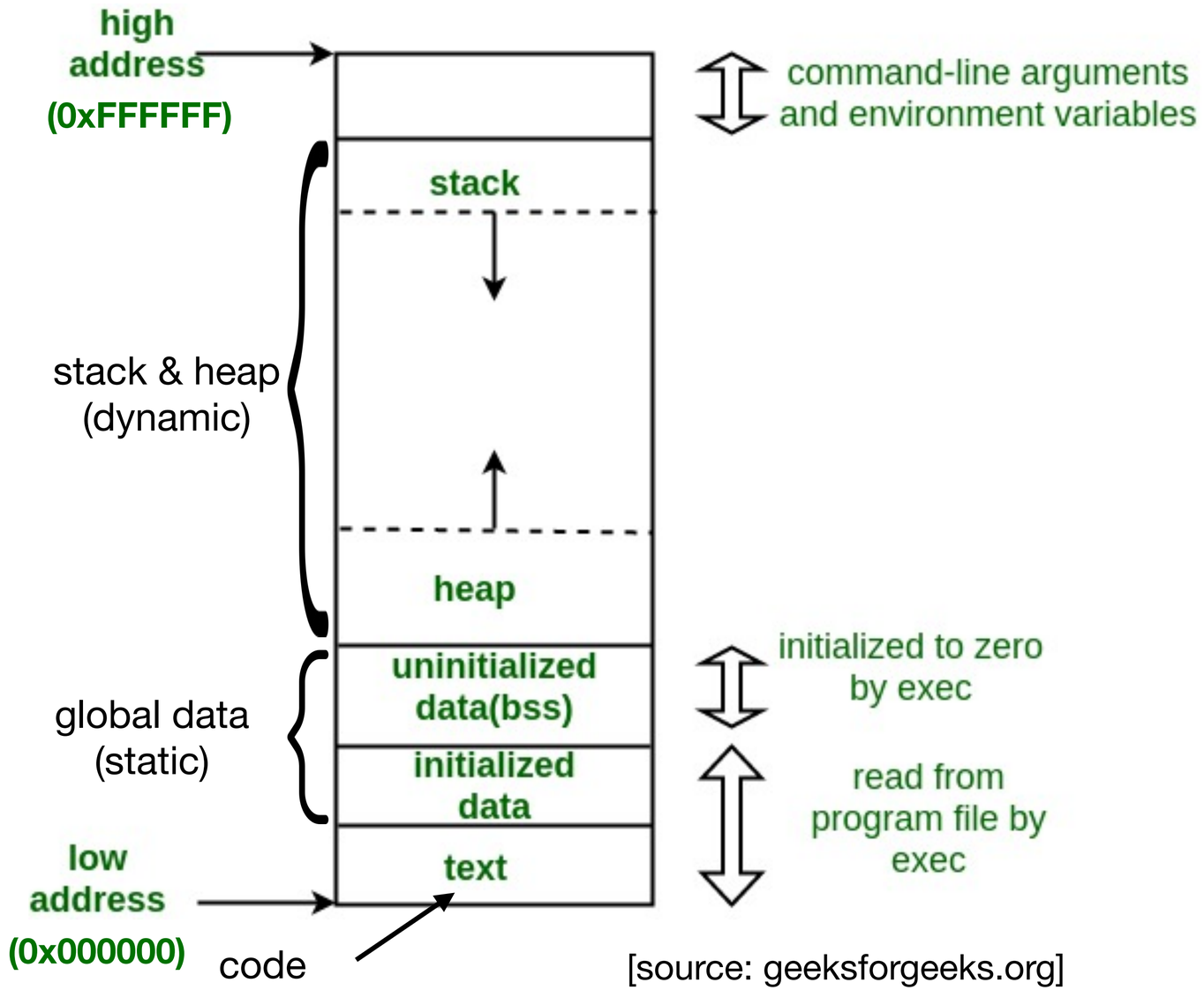
- Our examples will assume a Linux system
- In other OSes there may be small inessential differences
- Each process has access to virtual memory, that is “simulated” by the OS
- OS manages the virtual memory (e.g., paging)



[source: Wikipedia]

Memory Model

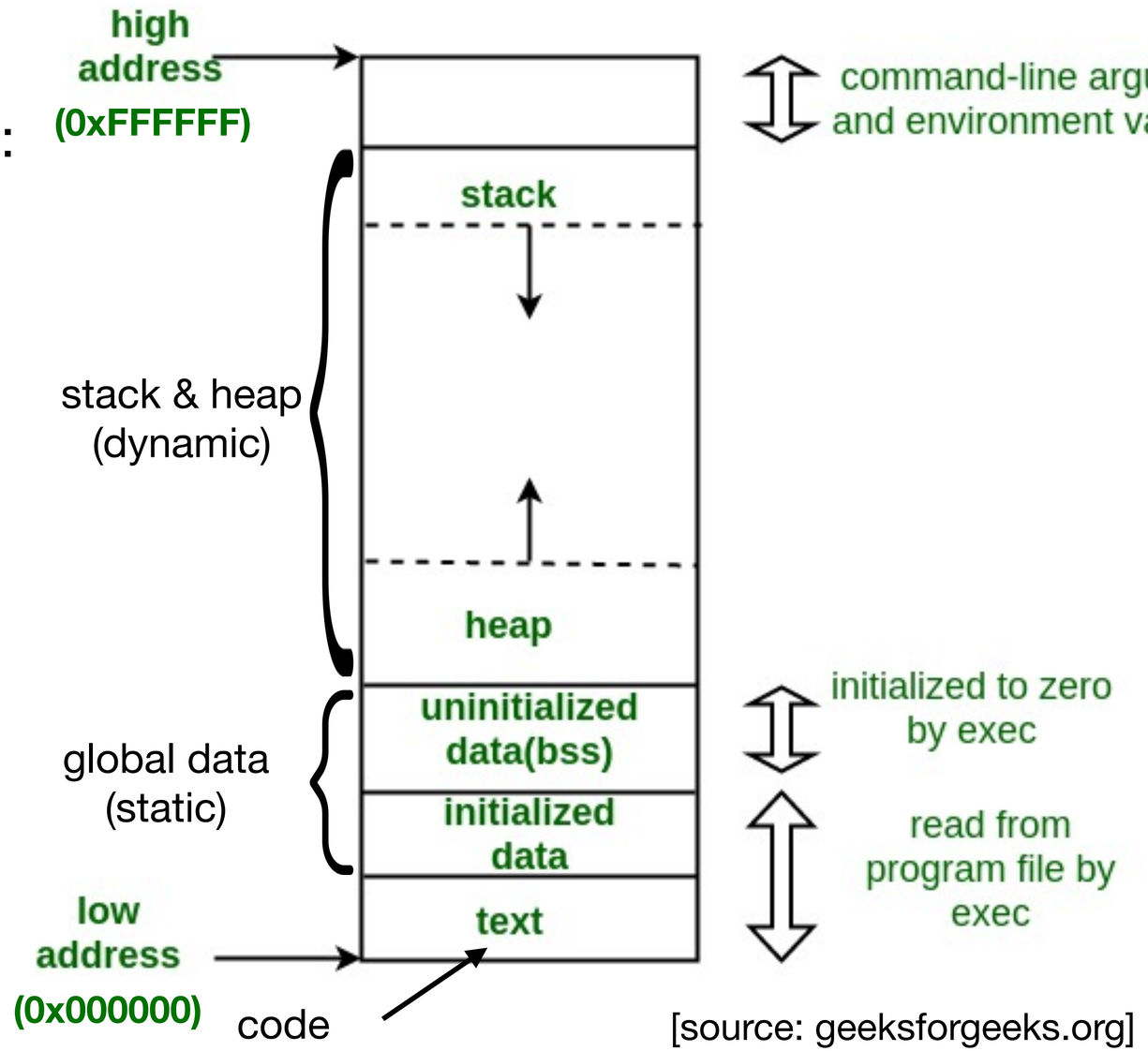
- When we program in a low-level language (e.g., C)
- There is a memory management convention
- That convention is established by the HW architecture and the OS
  - We will assume Intel x86



[source: geeksforgeeks.org]

Memory Model

- Memory allocated automatically (**stack**):
  - return address
  - global variables
  - function/procedure arguments
- Memory allocated “on demand” (**heap**)
- Memory w/ global/static data (**data**)
- Memory w/ code (**text**)



[source: geeksforgeeks.org]

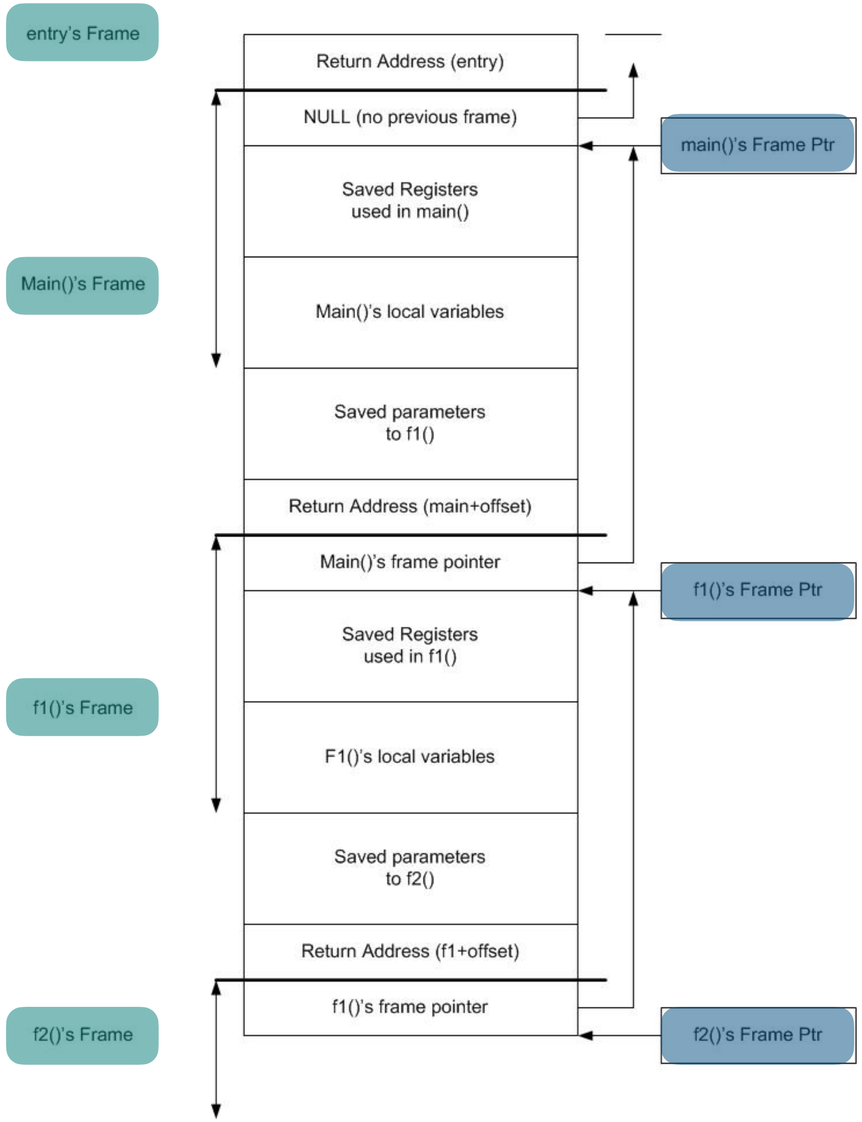
Stealing control

- Occurs when an external input leads the program to break the convention:
  - Replacing the expected sequence of instructions for a sequence with instruction controlled by the attacker
    1. Writing the malicious sequence of instructions to memory
    2. Executing the malicious sequence of instructions from memory
  - May occur in the stack, in the heap, during system calls, etc
  - Much harder than 30 years ago
  - But still very common

Smashing the Stack (buffer overflows)

Understanding the Stack

- Each function works under a local context called a **stack frame**:
  - Remember: stack grows downwards
  - **Frame pointer** stored in a registry (e.g., ebp) points to the start of that region (local variables)
  - **Stack pointer** stored in a registry (e.g., esp) points to the end of that region
  - Contains: parameters, data stored for the return, local variables, other temporary data.
- Main(…) → f1(…) → f2(…)



https://www.backerstreet.com/red/stack_frames.htm

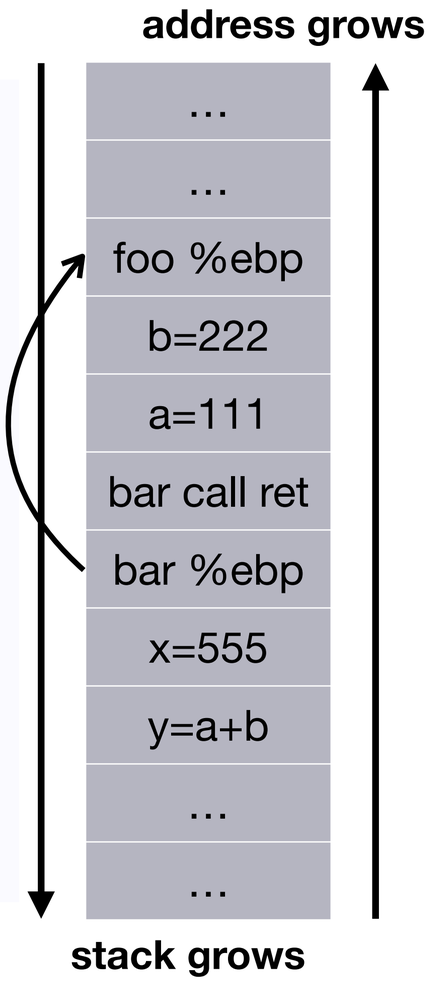
Understanding the Stack

```
void
bar(int a, int b)
{
    int x, y;

    x = 555;
    y = a+b;
}

void
foo(void) {
    bar(111,222);
}
```

```
bar:        # --------- start of the function bar()
    pushl   %ebp        # save the incoming frame pointer
    movl    %esp, %ebp  # set the frame pointer to the current top of stack
    subl    $16, %esp   # increase the stack by 16 bytes (stacks grow down)
    movl    $555, -4(%ebp)  # x=555 a is located at [ebp-4]
    movl    12(%ebp), %eax  # 12(%ebp) is [ebp+12], which is the second parameter
    movl    8(%ebp), %edx   # 8(%ebp) is [ebo+8], which is the first parameter
    addl    %edx, %eax  # add them
    movl    %eax, -8(%ebp)  # store the result in y
    leave           #
    ret         #
:       # --------- start of the function foo()
    pushl   %ebp        # save the current frame pointer
    movl    %esp, %ebp  # set the frame pointer to the current top of the stack
    subl    $8, %esp    # increase the stack by 8 bytes (stacks grow down)
    movl    $222, 4(%esp)   # this is effectively pushing 222 on the stack
    movl    $111, (%esp)    # this is effectively pushing 111 on the stack
    call    bar     # call = push the instruction pointer on the stack and branch to foo
    leave           # done
    ret         #
```



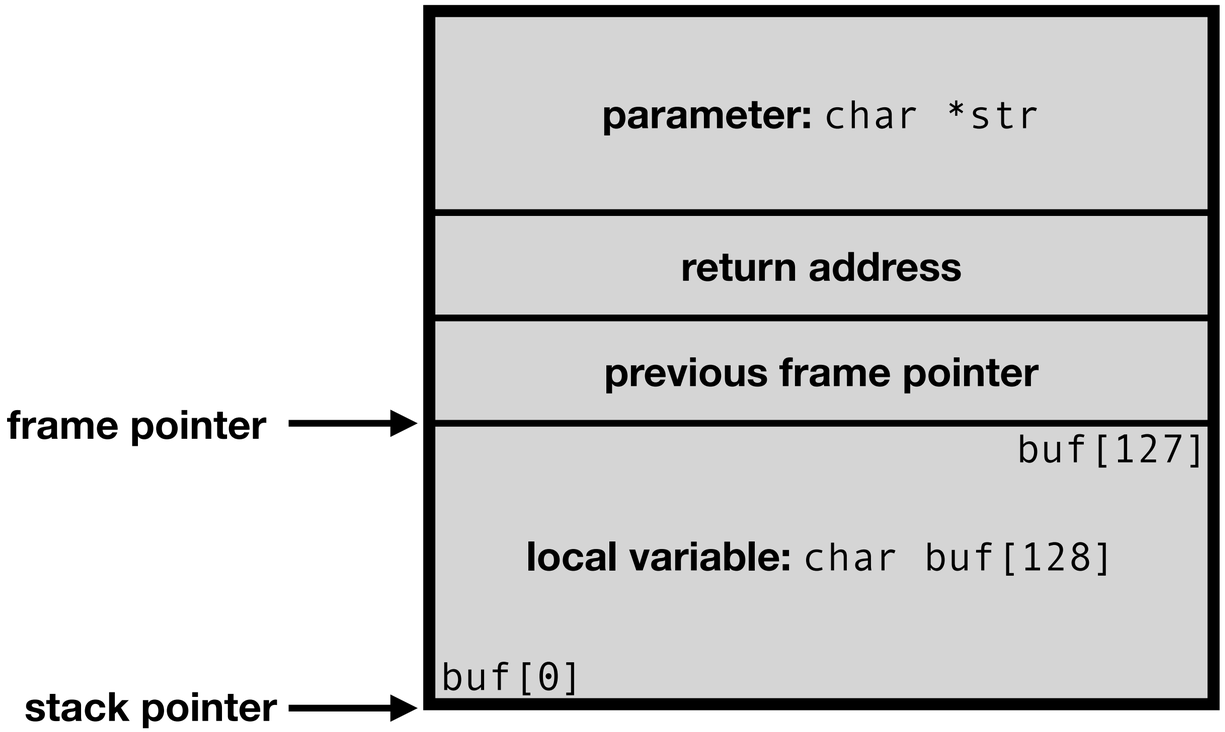
https://people.cs.rutgers.edu/~pxk/419/notes/frames.html
https://stackoverflow.com/questions/29790175/assembly-x86-leave-instruction/29790275

%ebp positive offsets = params + ret
%ebp negative offsets = local variables

Stack Smashing

- When entering `func`, the stack frame looks like:

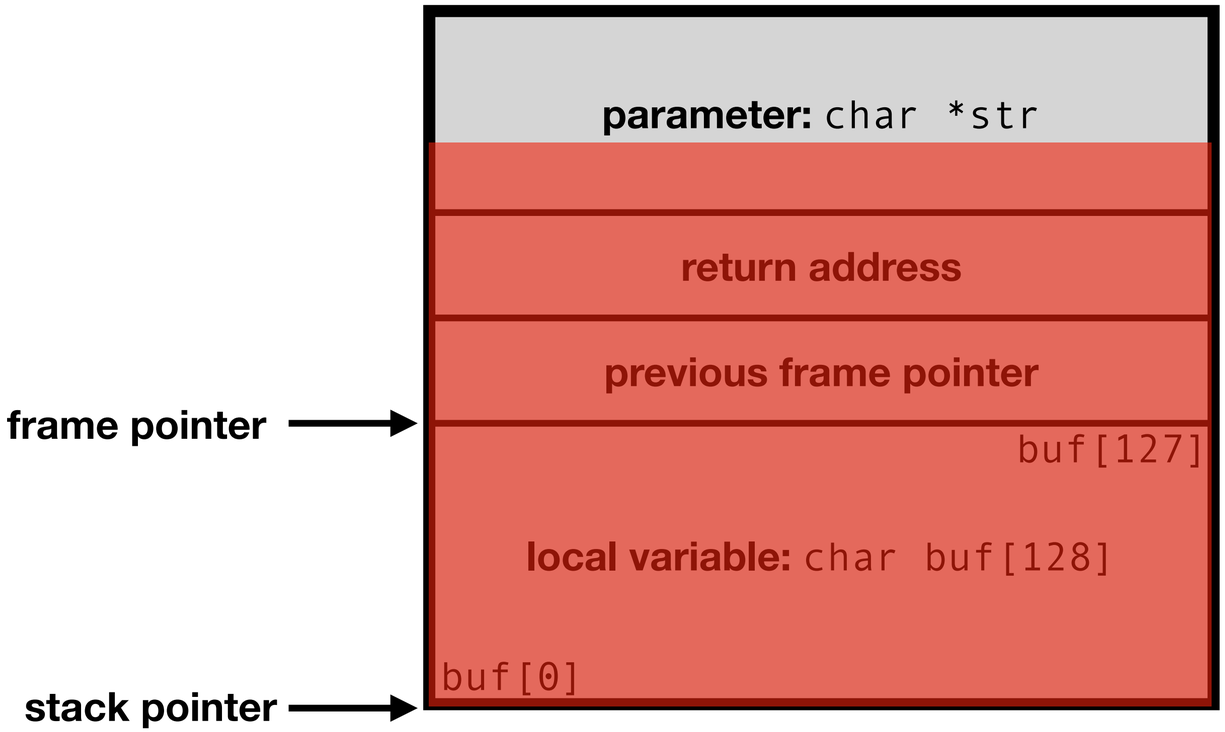
```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    usar(buf);
}
```



- What does `strcpy` do?
- What if `str` has size > 128?
- Crash? Undefined?

Stack Smashing

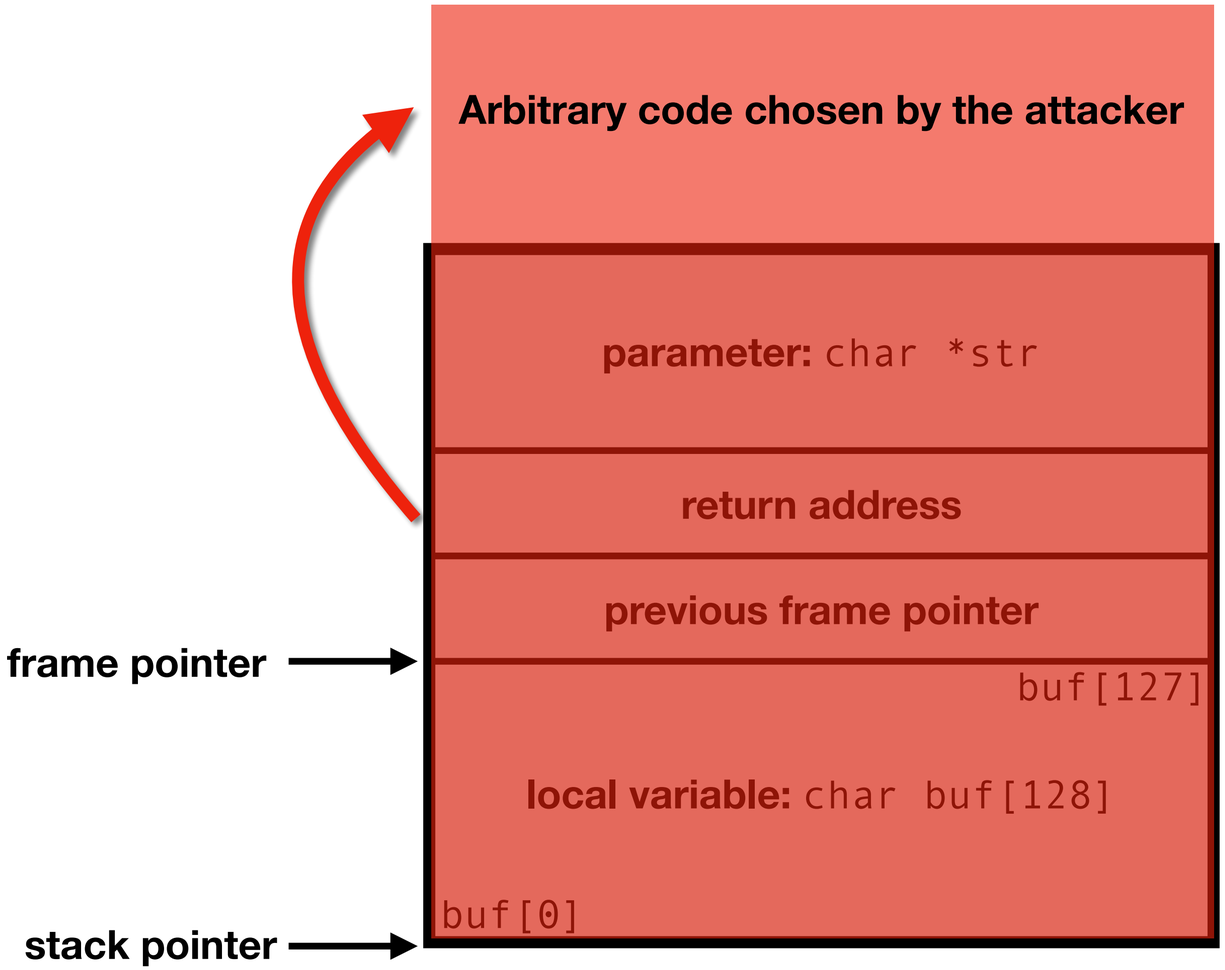
- The function writes to increasing memory positions:




```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    usar(buf);
}
```

- What happens when the return address is accessed when the function returns?

Stack Smashing




```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    usar(buf);
}
```

- If `str` comes from outside the security perimeter:
  - attacker may fill stack with code
  - replace the return address
  - execute arbitrary code

There is no separation between code and data!

Which code to inject?

The Holy Grail of RCE

- “Shell code” is a sequence of instructions that executes shell commands:
  - written in Assembly code
  - preceded by some NOP operations (because its memory address may vary)
  - injected on memory to be executed on return ⇒ buffer overflow

```
/*

Title:  Linux x86 - execve("/bin/bash", ["/bin/bash", "-p"], NULL) - 33 bytes
Author: Jonathan Salwan
Mail:   submit@shell-storm.org
Web:    http://www.shell-storm.org

!Database of Shellcodes http://www.shell-storm.org/shellcode/


sh sets (euid, egid) to (uid, gid) if -p not supplied and uid < 100
Read more: http://www.faqs.org/faqs/unix-faq/shell/bash/#ixzz0mzPmJC49

sassembly of section .text:

08048054 <.text>:
 8048054:       6a 0b                   push   $0xb
 8048056:       58                      pop    %eax
 8048057:       99                      cltd
 8048058:       52                      push   %edx
 8048059:       66 68 2d 70             pushw  $0x702d
 804805d:       89 e1                   mov    %esp,%ecx
 804805f:       52                      push   %edx
 8048060:       6a 68                   push   $0x68
 8048062:       68 2f 62 61 73          push   $0x7361622f
 8048067:       68 2f 62 69 6e          push   $0x6e69622f
 804806c:       89 e3                   mov    %esp,%ebx
 804806e:       52                      push   %edx
 804806f:       51                      push   %ecx
 8048070:       53                      push   %ebx
 8048071:       89 e1                   mov    %esp,%ecx
 8048073:       cd 80                   int    $0x80

*/

#include <stdio.h>

char shellcode[] = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70"
                   "\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61"
                   "\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52"
                   "\x51\x53\x89\xe1\xcd\x80";

int main(int argc, char *argv[])
{
        fprintf(stdout,"Length: %d\n",strlen(shellcode));
        (*(void(*)()) shellcode)();
}
```

http://shell-storm.org/shellcode/

The devil is in the details

- Building a working **exploit** requires a lot of careful “fine tuning”
  - E.g., the codification of the program as an array of bytes cannot contain `'\0'` (why?)
- A robust exploit will work consistently in arbitrary executions of the vulnerable code in similar platforms
- Many variants/tricks are used to achieve such robustness
- E.g., to avoid having to guess the specific address where the *shellcode* is stored:
  - Place the *shellcode* in the stack immediately above the return address
  - Point the return address to a memory address with the instruction `jmp sp`

Brief history [wikipedia]

- The concept is documented since 1972!
- First hostile use documented in 1988 (Morris worm)
- Generalised knowledge:
  - "Smashing the stack for fun and profit", Elias Levy/Aleph One, 1996
- Many famous/infamous examples, including both funny and disruptive ones:
  - Wii, PS2, XBox jailbreaks

[source: GameBoy Pokemon Red]
https://youtu.be/Nh3X8zSmTwY

fingerd: Morris Worm (1988)

Finger Daemon Buffer Overflow

Vulnerability Description

Brief Description: The *finger*(1) daemon is vulnerable to a buffer overrun attack, which allows a network entity to connect to the *fingerd*(8) port and get a *root* shell.

Detailed Description: *Fingerd* is a daemon that responds to requests for a listing of current users, or specific information about a particular user. It reads its input from the network, and sends its output to the network. On many systems, it ran as the *superuser* or some other privileged user. The daemon, *fingerd* uses *gets*(3) to read the data from the client. As *gets* does no bounds checking on its argument, which is an array of 512 bytes and is allocated on the stack, a longer input message will overwrite the end of the stack, changing the return address. If the appropriate code is loaded into the buffer, that code can be executed with the privileges of the *fingerd* daemon.

Component(s): finger, fingerd

Version(s): Versions before Nov. 6, 1989.

Operating System(s): All flavors of the UNIX operating system.

https://minnie.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c

```
main(argc, argv)
    char *argv[];
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
        fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets(line);
```

Student @ Cornell

⇓

Felon @ US CFFA

⇓

Professor @ MIT

“payload-free” worm that just disseminated itself

Why so common?

- String/buffer manipulation via libraries such as `libc`
- Many functions are outright unsafe:
  - Do not guaranteed reads/writes in designated memory regions
  - `strcpy`, `strcat`, `gets`, `scanf`
- Other functions may check bounds (e.g. `strncpy`) but do not guarantee that string is properly terminated
- Or simply code implemented from scratch with similar problems:
  - It is often assumed that input comes from a trusted source ⇒ 💀
- **Memory management vulnerabilities are really subtle!** 

libpng (2004)

libpng png_handle_tRNS() Buffer Overflow May Let Remote Users Execute Arbitrary Code

SecurityTracker Alert ID: 1011848
SecurityTracker URL: http://securitytracker.com/id/1011848
CVE Reference: GENERIC-MAP-NOMATCH *(Links to External Site)*
Updated: Oct 21 2004
Original Entry Date: Oct 21 2004
Impact: Execution of arbitrary code via network, User access via network
Fix Available: Yes **Vendor Confirmed:** Yes

Lack of input validation

Description: A buffer overflow vulnerability was reported in libpng. A remote user may be able to execute arbitrary code on the target system.

Debian reported that an image with certain height or width values can trigger an overflow in the png_handle_tRNS() function in pngrutil.c.
Impact: A remote user may be able to execute arbitrary code on a system that uses libpng.
Solution: Versions 1.0.17 and 1.2.7 include the fix and are available at:

http://sourceforge.net/project/showfiles.php?group_id=5624

[Editor's note: The fix may have been introduced into earlier versions.]
Vendor URL: libpng.sourceforge.net/ *(Links to External Site)*
Cause: Boundary error
Underlying OS: Linux (Any), UNIX (Any)

```
if (!(png_ptr->mode & PNG_HAVE_PLTE))
{
   /* Should be an error, but we can cope with it */
   png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette)
{
   png_warning(png_ptr, "Incorrect tRNS chunk length");
   png_crc_finish(png_ptr, length);
   return;
}
```

We can see, if the first warning condition is hit, the length check is missed due to the use of an "else if".

http://scary.beasts.org/security/CESA-2004-001.txt

libpng (2004)

CVE Details
The ultimate security vulnerability datasource

Search
(e.g.: CVE-2009-1234 or 2010-1234 or 20101234)
View CVE

Log In Register What's the CVSS score of your company?
Vulnerability Feeds & WidgetsNew www.itsecdb.com

Switch to https://
Home
Browse :
Vendors
Products
Vulnerabilities By Date
Vulnerabilities By Type
Reports :
CVSS Score Report
CVSS Score Distribution
Search :
Vendor Search
Product Search
Version Search
Vulnerability Search
By Microsoft References
Top 50 :
Vendors
Vendor Cvss Scores
Products
Product Cvss Scores
Versions
Other :
Microsoft Bulletins
Bugtraq Entries

Vulnerability Details : CVE-2004-0597

Multiple buffer overflows in libpng 1.2.5 and earlier, as used in multiple products, allow remote attackers to execute arbitrary code via malformed PNG images in which (1) the png_handle_tRNS function does not properly validate the length of transparency chunk (tRNS) data, or the (2) png_handle_sBIT or (3) png_handle_hIST functions do not perform sufficient bounds checking.

Publish Date : 2004-11-23 Last Update Date : 2018-10-12

Collapse All Expand All Select Select&Copy Scroll To Comments External Links
Search Twitter Search YouTube Search Google

– CVSS Scores & Vulnerability Types

CVSS Score	**10.0**
Confidentiality Impact	Complete (There is total information disclosure, resulting in all system files being revealed.)
Integrity Impact	Complete (There is a total compromise of system integrity. There is a complete loss of system protection, resulting in the entire system being compromised.)
Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Execute Code Overflow
CWE ID	CWE id is not defined for this vulnerability

Off-by-1 is enough (realpath, 2003)

```
/*
 * Join the two strings together, ensuring that the right thing
 * happens if the last component is empty, or the dirname is root.
 */
if (resolved[0] == '/' && resolved[1] == '\0')
        rootd = 1;
else
        rootd = 0;

if (*wbuf) {
        if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
                errno = ENAMETOOLONG;
                goto err1;
        }
        if (rootd == 0)
                (void)strcat(resolved, "/");
        (void)strcat(resolved, wbuf);
```

Details:
========

An off-by-one bug exists in fb_realpath() function. An overflow occurs when the length of a constructed path is equal to the MAXPATHLEN+1 characters while the size of the buffer is MAXPATHLEN characters only. The overflowed buffer lies on the stack.

The bug results from misuse of rootd variable in the calculation of length of a concatenated string:

source: https://www.securityfocus.com/bid/8315/info

https://www.exploit-db.com/exploits/22974

Copy

The 'realpath()' function is a C-library procedure to resolve the canonical, absolute pathname of a file based on a path that may contain values such as '/', './', '../', or symbolic links. A vulnerability that was reported to affect the implementation of 'realpath()' in WU-FTPD has lead to the discovery that at least one implementation of the C library is also vulnerable. FreeBSD has announced that the off-by-one stack- buffer-overflow vulnerability is present in their libc. Other systems are also likely vulnerable.

Reportedly, this vulnerability has been successfully exploited against WU-FTPD to execute arbitrary instructions.

NOTE: Patching the C library alone may not remove all instances of this vulnerability. Statically linked programs may need to be rebuilt with a patched version of the C library. Also, some applications may implement their own version of 'realpath()'. These applications would require their own patches. FreeBSD has published a large list of applications that use 'realpath()'. Administrators of FreeBSD and other systems are urged to review it. For more information, see the advisory 'FreeBSD-SA-03:08.realpath'.

https://bugtraq.securityfocus.narkive.com/bzojnPB1/wu-ftpd-fb-realpath-off-by-one-bug

The potential was there! (1998)

The poisoned NUL byte

From: okir () MONAD SWB DE (Olaf Kirch)
Date: Wed, 14 Oct 1998 11:42:46 +0200

Summary: you can exploit a single-byte buffer overrun to gain root privs.

The interesting thing about this overrun is that it was by just a single byte. And yes, it not just crashed the process, it provided a root shell. It took me a while to figure out, but what it boils down to is this:

At the beginning of the function, realpath copies the argument (1024 bytes) To a local buffer (sized MAXPATHLEN, i.e. 1024 bytes). Thus, the terminating 0 byte of the string gets scribbled over the next byte, which happens to be the lowest byte of %ebp, the frame pointer of the calling function. At function entry, its value was 0xbffff3ec. After the strcpy, it becomes 0xbffff300.

During the remainder of realpath(), nothing exciting happens, but when the function returns, %ebp is restored from stack, which effectively shifts down the calling function's stack frame by 0xec bytes.

The calling function now does a few things with local data, dereferences some pointers (by sheer dumb luck these pointers contain random but valid addresses), and returns, restoring the %esp and %ebp registers from stack. With the stack having shifted down 0xec bytes, it picks up the return address from the local buffer containing the exploit code...

https://seclists.org/bugtraq/1998/Oct/109

And still exists (2014)

News and updates from the Project Zero team at Google

Monday, August 25, 2014

The poisoned NUL byte, 2014 edition

Posted by Chris Evans, Exploit Writer Underling to Tavis Ormandy

Back in this 1998 post to the Bugtraq mailing list, Olaf Kirch outlined an attack he called "The poisoned NUL byte". It was an off-by-one error leading to writing a NUL byte outside the bounds of the current stack frame. On i386 systems, this would clobber the least significant byte (LSB) of the "saved %ebp", leading eventually to code execution. Back at the time, people were surprised and horrified that such a minor error and corruption could lead to the compromise of a process.

Fast forward to 2014. Well over a month ago, Tavis Ormandy of Project Zero disclosed a glibc NUL byte off-by-one overwrite into the heap. Initial reaction was skepticism about the exploitability of the bug, on account of the malloc metadata hardening in glibc. In situations like this, the Project Zero culture is to sometimes "wargame" the situation. geohot quickly coded up a challenge and we were able to gain code execution. Details are captured in our public bug. This bug contains analysis of a few different possibilities arising from an off-by-one NUL overwrite, a solution to the wargame (with comments), and of course a couple of different variants of a full exploit (with comments) for a local Linux privilege escalation.

Inspired by the success of the wargame, I decided to try and exploit a real piece of software. I chose the "pkexec" setuid binary as used by Tavis to demonstrate the bug. The goal is to attain root privilege escalation. Outside of the wargame environment, it turns out that there are a series of very onerous constraints that make exploitation hard. I did manage to get an exploit working, though, so read on to see how.

Project Zero

googleprojectzero.blogspot.com

Project Zero is a team of security analysts employed by Google tasked with finding zero-day vulnerabilities. It was announced on 15 July 2014. Wikipedia

Founded: July 15, 2014

https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html

Are there no alternatives?

- Why not use higher-level languages with better memory safety guarantees? Java, JavaScript, Python, Rust, Haskell, etc
  - They often compile to and/or use C/LLVM libraries
- Virtual machines, sandboxes, browsers, OSes still have to be programmed using low-level languages
- In specialised domains (e.g., cryptographic code), humans are still better at aggressively optimising code than compilers
- Many attacks chain together various of these vulnerabilities
- As we will keep seeing, input is always potentially malicious

Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CSE127: Stack Buffer Overflows + Buffer Overflows and Stack Smashing
  - CS155: Basic Control Hijacking Attacks
  - CS343: Control Flow + Control Flow II + Control Flow III