# Fundamentos de Segurança Informática (FSI)

2021/2022 - LEIC

**Manuel Barbosa**
**mbb@fc.up.pt**

# Aula 3
# Controlo (Parte 1)

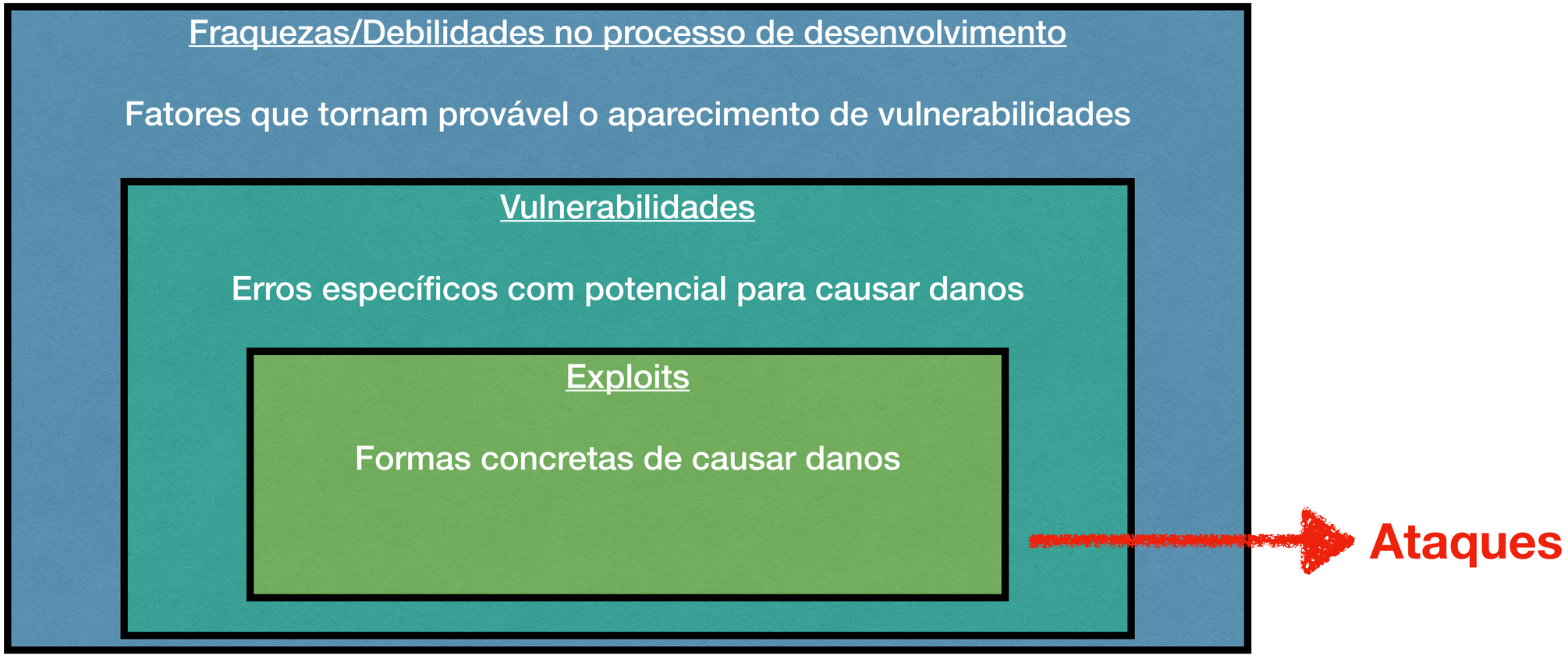
# A Criatura vs O Criador

**Software**

**Nós**

# A Criatura vs O Criador

- O software pode ser manipulado para se voltar contra nós



# Aprender com a história

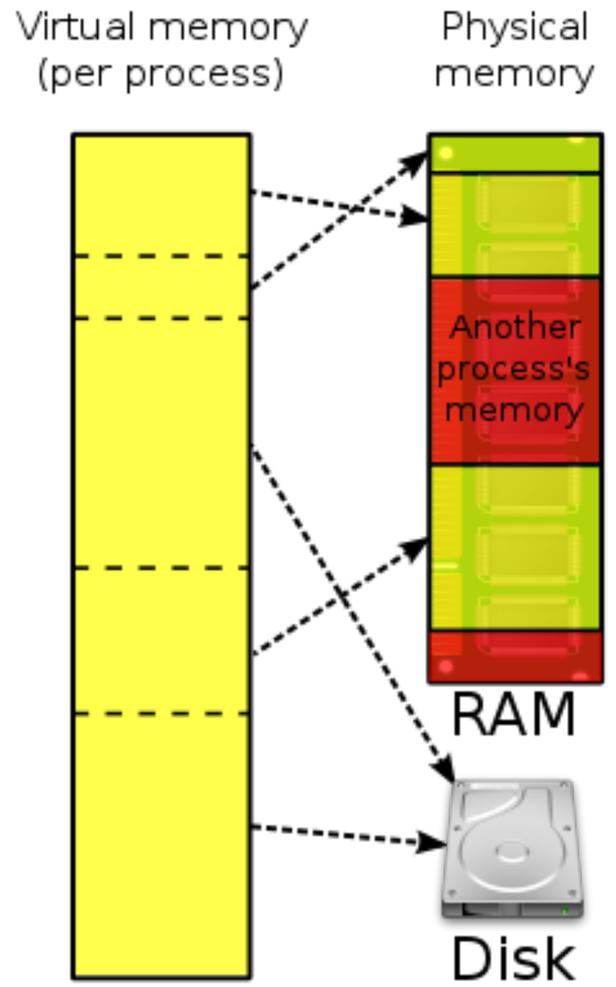
- É importante estudar vulnerabilidades e ataques anteriores
  - Ser capaz de identificar potenciais vulnerabilidades e soluções
  - Incentivo para a correção: conhecimento + falha = negligência
  - Caracterizar as ameaças:
    - avaliar a seriedade de vulnerabilidades num determinado contexto
    - ajudar utilizadores a avaliar o risco

# Vamos começar "bottom-up"

- O software executa numa plataforma computacional
  - Hardware de processamento + Interfaces com o exterior
  - Armazenamento dinâmico (memória) e persistente (disco)
  - Sistema Operativo
- É sempre uma potencial arma contra essa plataforma:
  - Pode permitir **ganhar o controlo sobre essa plataforma**
  - Pode depois ser utilizado para diversos fins: DoS, SpyWare, Jailbreak, etc.

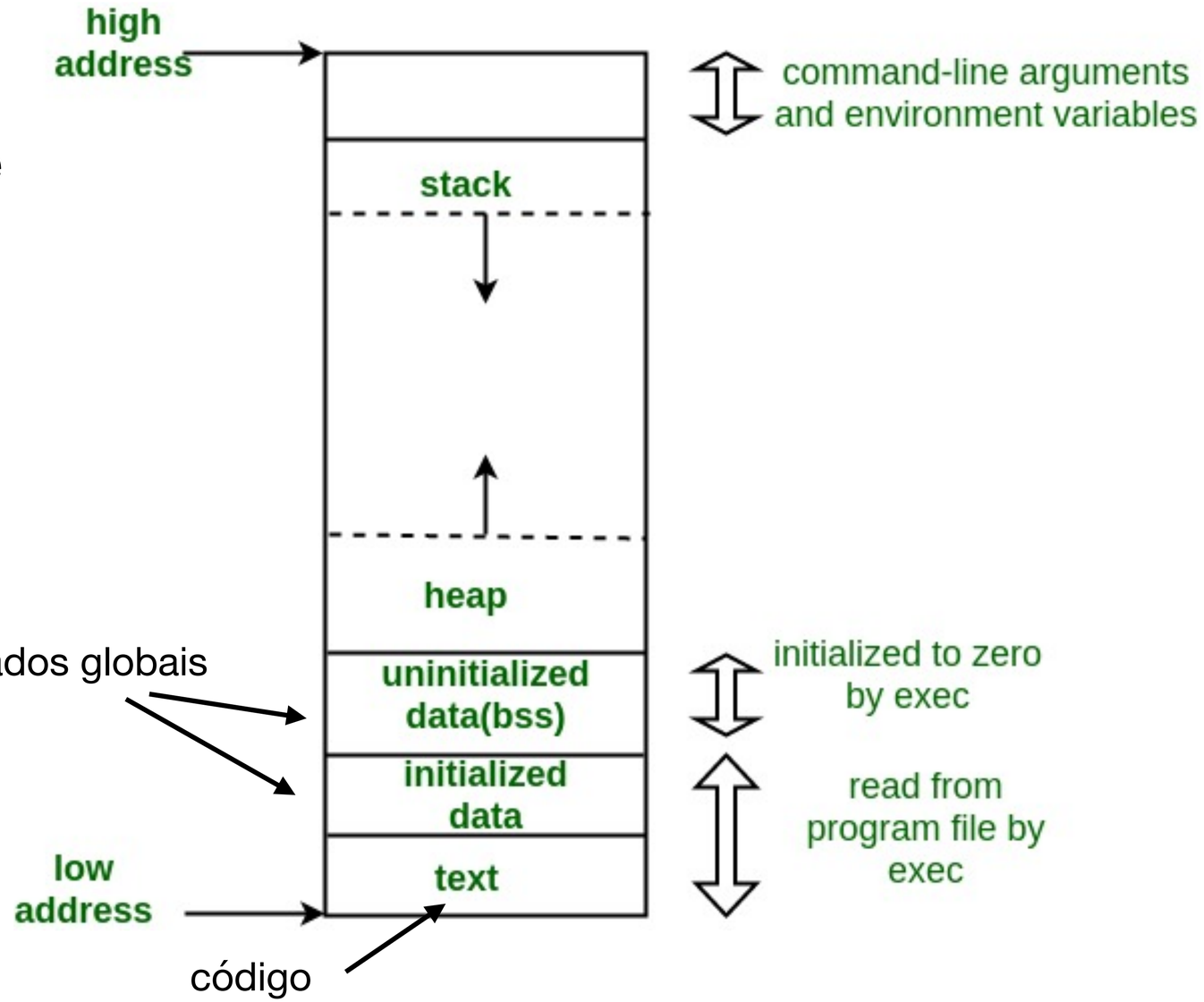
# Modelo de Memória

- Vamos dar exemplos num sistema Linux típico
- Em outros SO haverá diferenças, mas serão de pormenor
- Cada processo tem acesso a memória virtual, que é "simulada" pelo SO (Wikipedia)



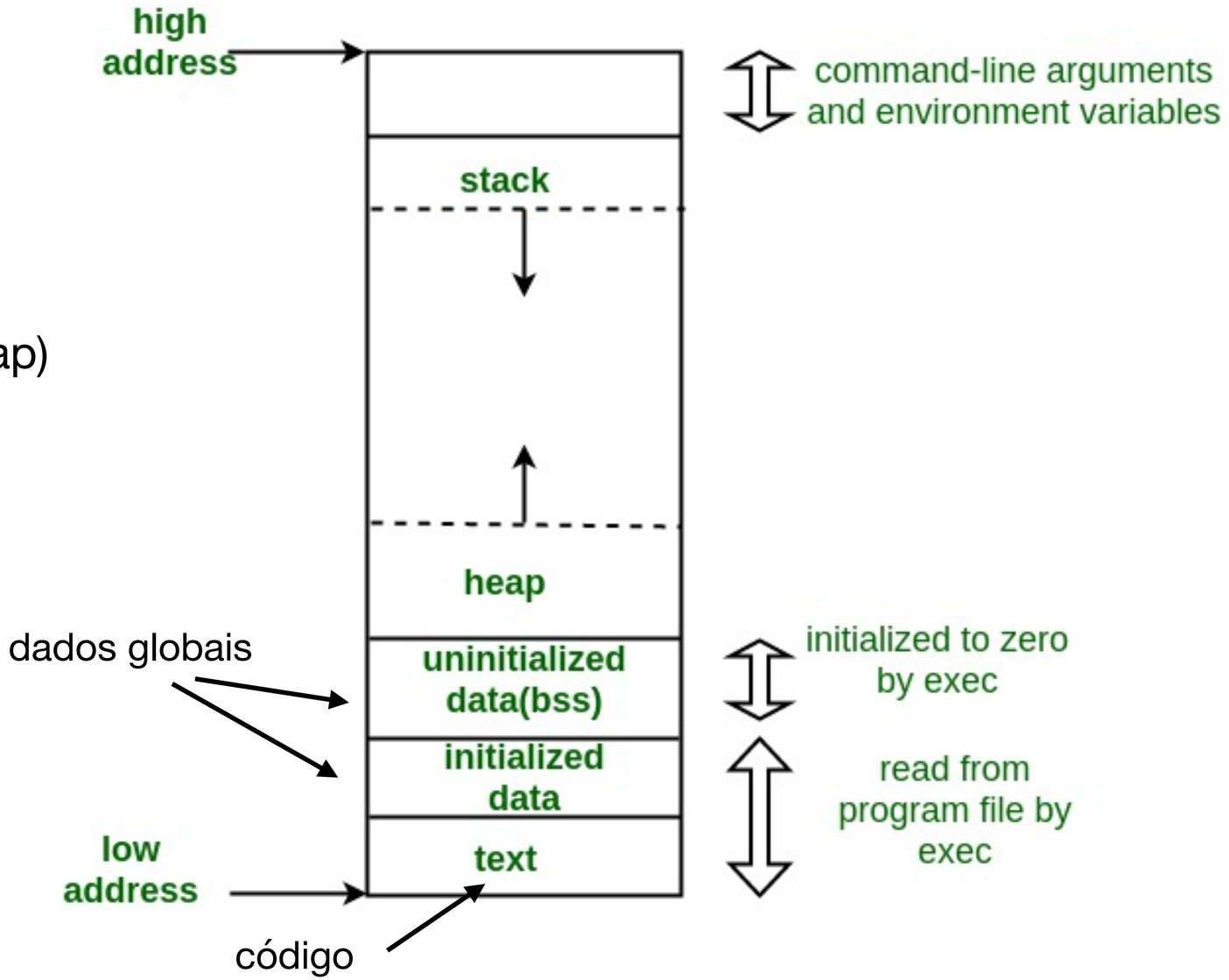
# Modelo de Memória

- Quanto escrevemos um programa numa linguagem de baixo nível (e.g., C)
- Segue-se uma convenção na gestão dessa memória [geeksforgeeks.org]
- Essa convenção é estabelecida pela arquitectura e pelo Sistema Operativo.



# Modelo de Memória

- Memória com código (text)
- Memória com dados globais/ estáticos (data)
- Memória alocada "on demand” (heap)
- Memória alocada automaticamente (stack):
  - endereço de retorno
  - variáveis locais
  - argumentos para funções



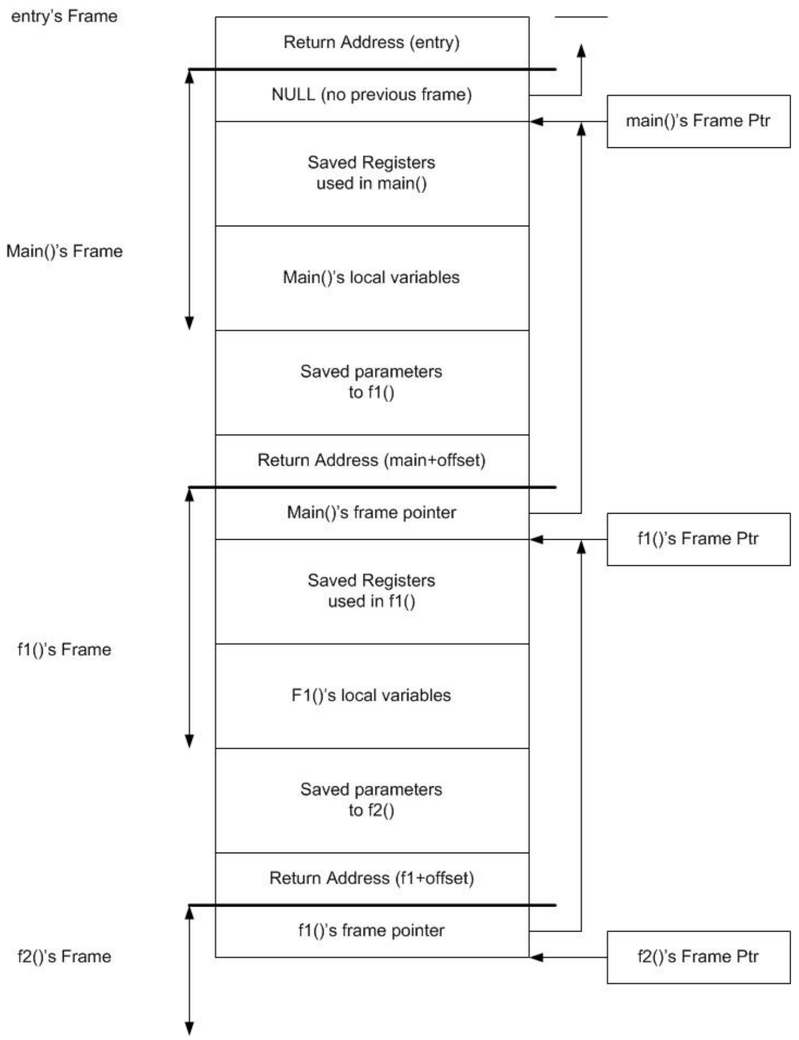
# Usurpação de Controlo

- Ocorre quando um input externo leva o programa a quebrar a convenção:
  - alterando a sequência de instruções que é executada
  - substituindo a sequência de instruções esperada por uma sequência de instruções controlada pelo atacante
  - pode ocorrer na stack, na heap, nas chamadas ao sistema, etc.
  - muito mais difícil hoje do que há 30 anos atrás
  - mas ainda muito comum

# Smashing the Stack (buffer overflows)

# Funcionamento típico da Stack

- Cada função funciona com base num contexto local chamado stack frame:
  - frame pointer guardado num registo (e.g., rbp) aponta para a base dessa região (variáveis locais)
  - stack pointer guardado num registo (e.g., rsp) aponta para o topo (endereço mais baixo
  - contém: parâmetros, dados guardados para o retorno, variáveis locais, outros valores guardados temporariamente.




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

# Funcionamento típico da Stack

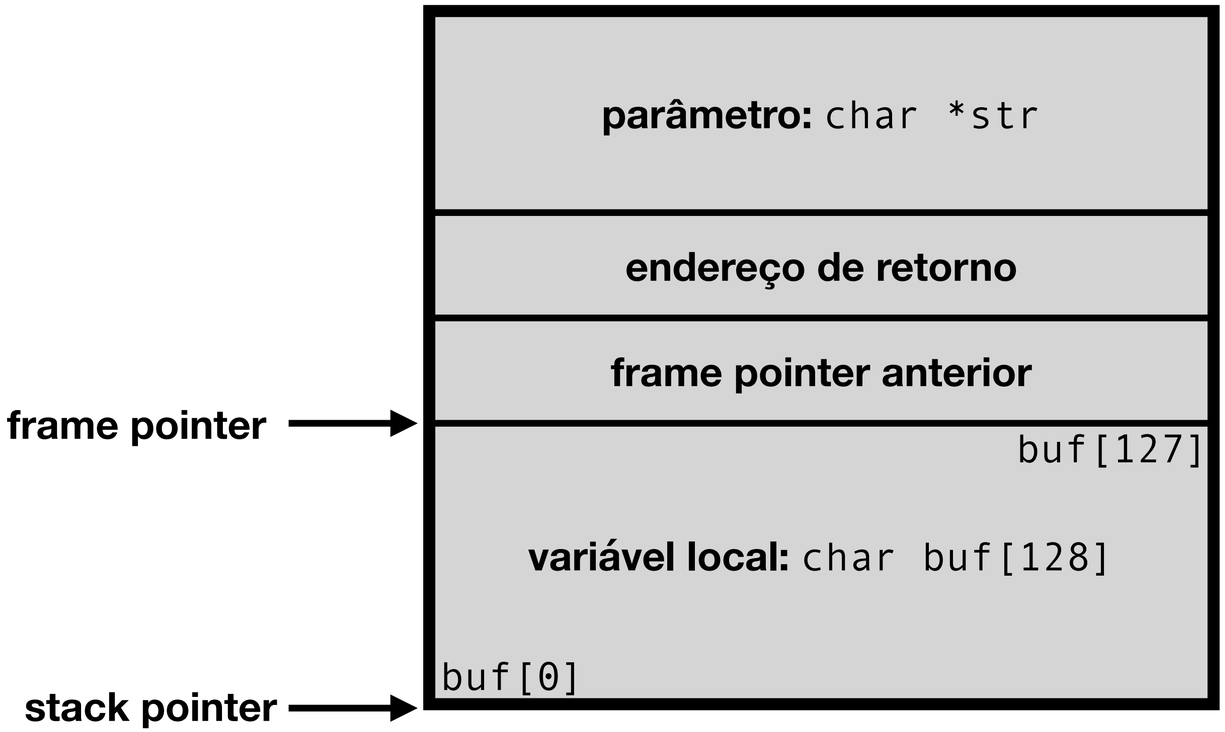
```
void
bar(int a, int b)
{
    int x, y;

    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         #
foo:        # --------- 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

# Stack Smashing

- Quando entramos nesta função, a stack frame tem o seguinte aspeto:



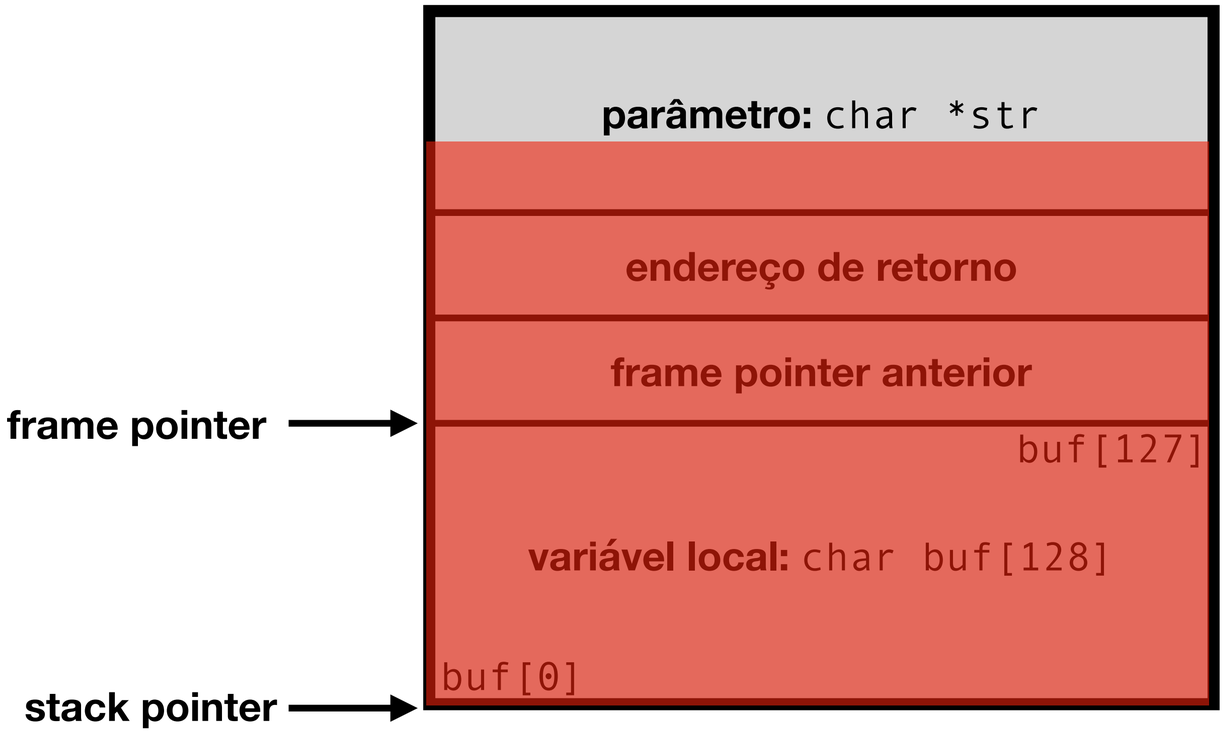

```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    usar(buf);
}
```

- E se `str` tiver tamanho maior que 128?

# Stack Smashing

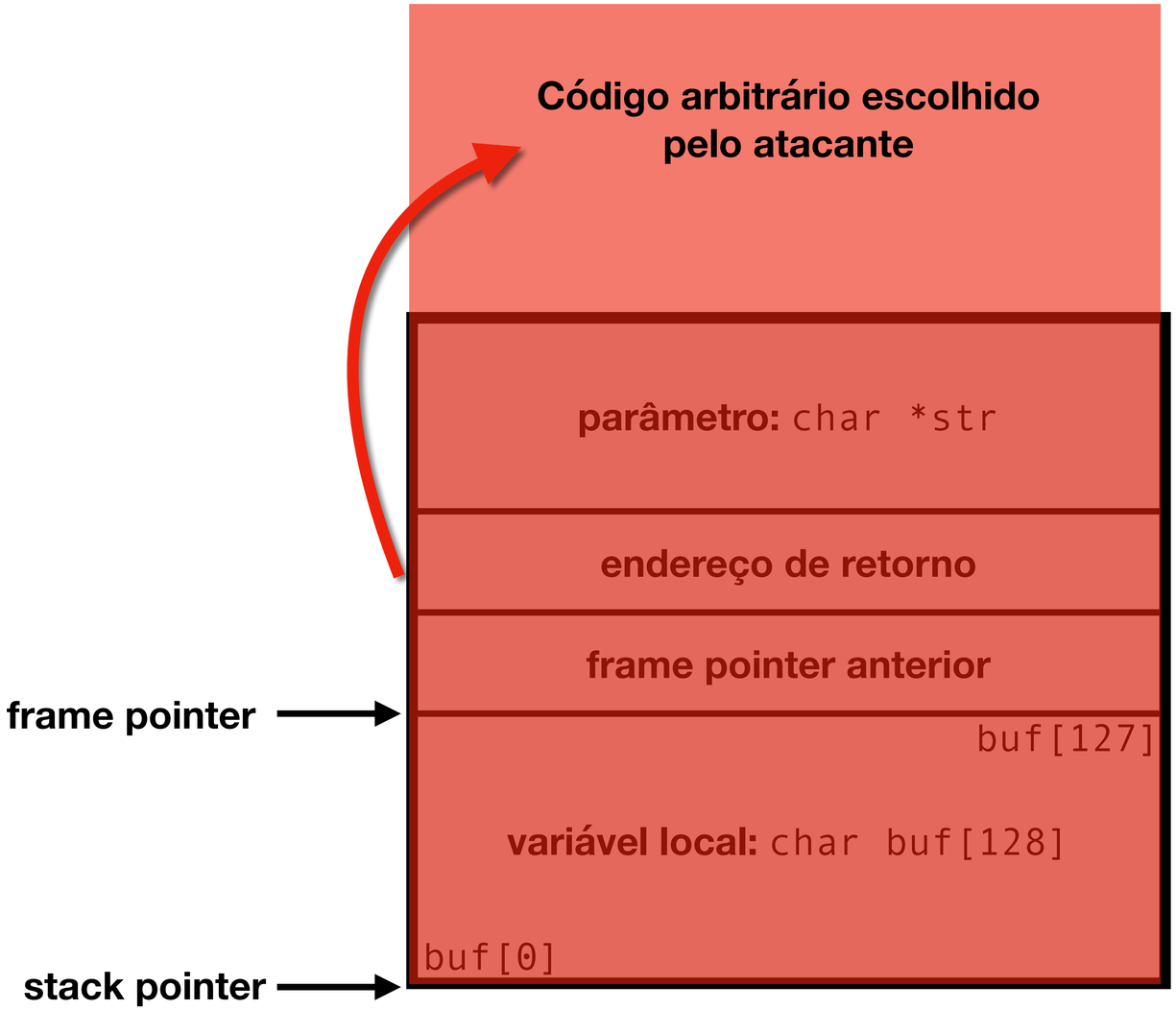
- A função escreve para posições crescentes na memória:

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




- O que acontece quando for executado o código que recupera o endereço de retorno no final da função?

# Stack Smashing




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

- Se `str` vier de fora do perímetro de segurança:
  - atacante pode preencher stack com código
  - substituir endereço de retorno
  - executar código arbitrário

# Que código injetar?

# Que código injetar?

- "Shell code" é uma sequência de instruções que executa comandos shell:
  - codificado em código máquina
  - precedido de algumas operações NOP (porque o seu endereço em memória pode variar)
  - injetado em memória para ser executada no retorno da função onde se provoca o buffer overflow

**Intel x86**

Linux/x86 - setuid + setgid + stdin re-open + exceve - 71 bytes *by Andre*
Linux/x86 - Followtheleader custom execve-shellcode Encoder/Decoder
Linux/x86 - ROT-7 Decoder execve - 74 bytes *by Stavros Metzidakis*
Linux/x86 - Add map in /etc/hosts file - 77 bytes *by Javier Tejedor*
Linux/x86 - Obfuscated - chmod({passwd,shadow}) - add new root user
Linux/x86 - setreuid() + exec /usr/bin/python - 54 bytes *by Ali Razmjoo*
Linux/x86 - chmod + Add new root user with password + exec sh - 378
Linux/x86 - Shell Reverse TCP Shellcode - 74 bytes *by Julien Ahrens*
Linux/x86 - Shell Bind TCP Shellcode Port 1337 - 89 bytes *by Julien Ahr*
Linux/x86 - sockfd trick + dup2(0,0),dup2(0,1),dup2(0,2) + execve /bin/sh
Linux/x86 - shutdown -h now - 56 bytes *by Osanda Malith Jayathissa*
Linux/x86 - chmod 0777 /etc/shadow (a bit obfuscated) Shellcode - 51 b
Linux/x86 - /bin/nc -le /bin/sh -vp 17771 - 58 bytes *by Oleg Boytsev*
Linux/x86 - JMP-FSTENV execve shell - 67 bytes *by Paolo Stivanin*
Linux/x86 - shift-bit-encoder execve - 114 bytes *by Shihao Song*
Linux/x86 - Copy /etc/passwd to /tmp/outfile - 97 bytes *by Paolo Stivan*
Linux/x86 - jump-call-pop execve shell - 52 bytes *by Paolo Stivanin*
Linux/x86 - Download + chmod + exec - 108 bytes *by Daniel Sauder*
Linux/x86 - reads /etc/passwd and sends the content to 127.1.1.1 port
Linux/x86 - Multi-Egghunter *by Ryan Fenno*
Linux/x86 - Obfuscated tcp bind shell - 112 bytes *by Russell Willis*
Linux/x86 - Obfuscated execve /bin/sh - 30 bytes *by Russell Willis*
Linux/x86 - egghunter shellcode *by Russell Willis*
Linux/x86 - Reverse TCP bind shell - 92 bytes *by Russell Willis*
Linux/x86 - Set /proc/sys/net/ipv4/ip_forward to 0 & exit() - 83 bytes *by*
Linux/x86 - TCP bind shell - 108 bytes *by Russell Willis*
Linux/x86 - Encrypted execve /bin/sh with uzumaki algorithm - 50 bytes
Linux/x86 - Mutated Execve Wget - 96 bytes *by Geyslan G. Bem*
Linux/x86 - Mutated Fork Bomb - 15 bytes *by Geyslan G. Bem*
Linux/x86 - Mutated Reboot - 55 bytes *by Geyslan G. Bem*
Linux/x86 - Tiny read /etc/passwd file - 51 bytes *by Geyslan G. Bem*
Linux/x86 - Tiny Execve sh Shellcode - 21 bytes *by Geyslan G. Bem*
Linux/x86 - Insertion Decoder Shellcode - 33+ bytes *by Geyslan G. Bem*

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

# ShellCode

```
/*

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/

# O diabo está nos detalhes

- A construção de um exploit na prática exige muito "fine tuning"
  - E.g., a codificação do código como array de bytes não pode conter `'\0'` (porquê?)
- Um exploit robusto funcionará de forma consistente em execuções arbitrárias do código vulnerável em plataformas do mesmo tipo
- Muitas variantes/truques permitem obter essa robustez
- Exemplo para evitar ter de adivinhar endereço específico onde reside *shellcode*:
  - colocar shell code no topo da stack depois do retorno
  - substituir endereço de retorno por endereço na memória de código (mais estáveis) onde esteja o opcode `jmp sp`

# Breve história [wikipedia]

- O conceito está documentado desde 1972!
- A primeira utilização hostil documentada foi em 1988 (Morris worm)
- Democratização/conhecimento generalizado:
  - "Smashing the stack for fun and profit", Elias Levy/Aleph One, 1996
- Muitos exemplos famosos/infames, incluindo alguns divertidos:
  - Wii, PS2, XBox jailbreaks

# 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.

```
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);
```

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

# Porquê tão comum?

- Manipulação de strings/buffers usando bibliotecas tipo `libc`
- Muitas funções são simplesmente unsafe:
  - não garantem escrita limitada região de memória pré-definida
  - `strcpy`, `strcat`, `gets`, `scanf`
- Noutros casos  isso é garantido, mas pode ser criado outro tipo de problemas: `strncpy` escreve em espaço pré-definido mas não garante que string está corretamente terminada
- Ou simplesmente código implementado de raiz com os mesmos problemas:
  - assume-se que o input vem de fonte confiável => 💀

# 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

**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

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 |

# realpath: basta um off-by-1 (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, "/");
```

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:

Copy

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

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

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

# O potencial estava lá!

## 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

# E continua a existir

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.

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

# Como evitar estes ataques?