Exploiting Software Vulnerabilities

Software Vulnerabilities

CONTROL-FLOW HIJACKING

Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2022/2023

Master's Degree in Informatics Engineering
UNIVERSITY OF ZARAGOZA
Seminar A.22, Ada Byron building

Outline

1. A little recap

2. Buffer Overflows

3. Defenses against Control-Flow Hijacking Attacks
   - Stack Data Protection
   - Non-Executable Stack
   - Write XOR eXecute (W^X) Pages
   - Address Space Layout
   - Other Techniques of Defense

Outline

1. A little recap
2. Buffer Overflows
3. Defenses against Control-Flow Hijacking Attacks

Recap on. . .

Definitions

System/defender perspective

- **Attack surface**
  - Exposure of a system to attacks
- **Vulnerability**
  - Software flaw that can be exploited by an attacker

Attacker perspective

- **Attack vector**
  - How the attack was carried out
- **Exploit**
  - Succeed by taking advantage of a vulnerability

Types of software vulnerabilities

- **Overflow**
    - **Buffer overflow**
    - **Heap overflow**
- **NULL pointer dereference**
- **Dynamic memory handling**
    - Use-after-free
    - Double free
    - Allocator abuse
- **Number handling**
- **Format strings**
- **Uninitialized memory**
- **Race conditions**

Vulnerability databases

- National Vulnerability Database (NVD), maintained by NIST (`https://nvd.nist.gov/`)
- MITRE CVE (`https://cve.mitre.org/`)
- Bugtraq (`http://www.securityfocus.com/archive/1`)
- …

Today we talk about…

Control-Flow Hijacking

- Attacker's goal: **to seize the target system**
  - **Run arbitrary code to hijack the control flow of a vulnerable application**

Functions

- **Block of instructions that performs a specific task**
- **Three components**:
  - Input (values passed from the caller)
  - Body (code to perform the task)
  - Return value (to the caller)
- **Calling a function involves a branch in the control flow** (i.e., jumping to another location)
  - The return address is usually stored in the caller's stack frame

Functions

```
int x = compute(arg0, arg1, ...)
```

What happens in the backstage before a function runs?

- **Parameters are configured to be passed to the function**
  - Either through the stack or logical registers
- **The address of the next instruction after the call is also saved**

Functions

```
int x = compute(arg0, arg1, ...)
```

What happens in the backstage before a function runs?

- **Parameters are configured to be passed to the function**
  - Either through the stack or logical registers
- **The address of the next instruction after the call is also saved**

What happens in the backstage after a function runs?

- **Return value is set**
  - Normally, the logical register `eax` contains the return value of a function
  - Allocated variables (within the function) are removed from the stack
  - Registers used in the function are restored to their previous values
  - The control is transferred to the saved return address

Functions

- **Standard prologue**
  - Occurs at the beginning of a function
  - Allocates space for local variables (on stack)
  - Saves registers to be reused in the body of the function
- **Standard epilogue**
  - Occurs at the end of a function
  - Normally, undoes what was done in the prologue
  - Cleans up the stack
  - Restores register values

Functions

```
mov edi, edi
```

Common prologue on Windows

- 2-byte length instruction
- Equivalent for a `nop` instruction, since it does *nothing*
- Used to hot-patch a running executable, without stopping and restarting it
  - Can be overwritten with a relative jump of 2 bytes!

Further reading: *Why do Windows functions all begin with a pointless MOV EDI, EDI instruction?.* R. Chen, 2011.

`https://devblogs.microsoft.com/oldnewthing/?p=9583`

Functions

Standard calling conventions

Calling conventions

- Describes **how data is passed in/out of functions**
- **Implementation may vary by compiler**

Functions

Standard calling conventions

Calling conventions

- Describes **how data is passed in/out of functions**
- **Implementation may vary by compiler**

`cdecl` **convention** (most common)

- Arguments are pushed onto the stack from right to left
- Return value is placed in `eax`
- The caller must clean the stack (removing passed parameters)

Functions

Standard calling conventions

`stdcall` **convention**

- Similar to `cdecl`, but *callee* clears the stack
- Convention used in Windows APIs

Functions

Standard calling conventions

`stdcall` convention

- Similar to `cdecl`, but *callee* clears the stack
- Convention used in Windows APIs

`fastcall` convention

- Arguments are passed by registers, put on the stack when a large number of arguments are required
  - For instance, the GCC and Microsoft compilers use the `ecx` and `edx` registers
- The *callee* clears the stack
- Return value is placed in `eax`

Functions

Standard calling conventions

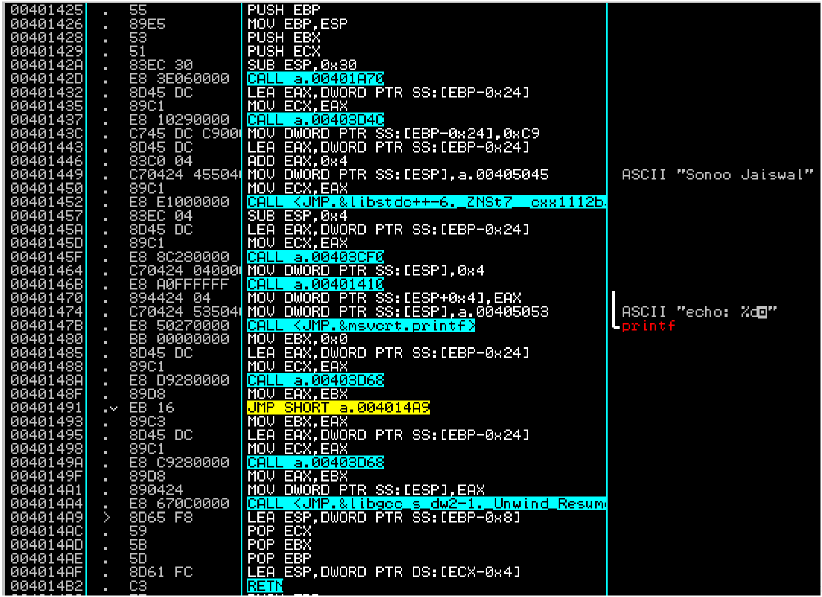
`thiscall` convention

- Used in C++ in object methods (member functions)
- Includes a reference to the `this` pointer
- Depends on the compiler:
  - In Microsoft compilers, `ecx` holds the `this` pointer and the *callee* clears the stack
  - In GNU compilers, the `this` pointer is pushed last and the *caller* clears the stack

Functions

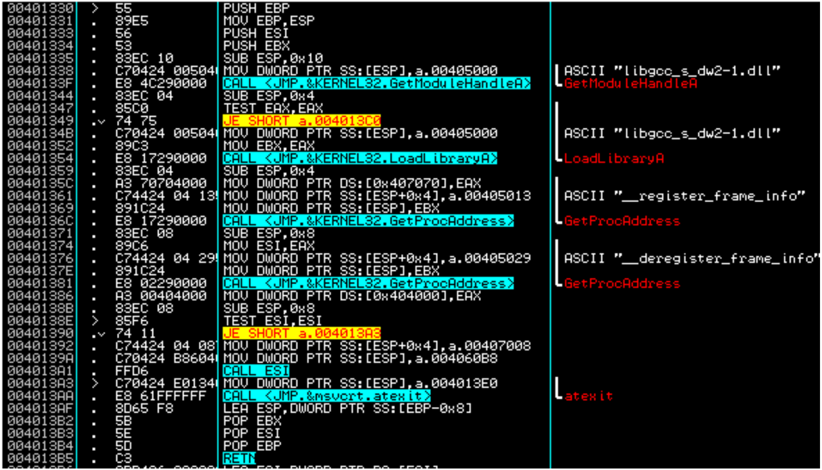
```
#include <iostream>
using namespace std;
class Student {
    public:
        int id; //data member (also instance variable)
        string name; //data member (also instance variable)

    void imprime(){
        cout << this -> id << endl;
        cout << this -> name << endl;
    }
};

_cdecl int echo(int x){
    return x + 8;
}

int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Sonoo Jaiswal";
    s1.imprime();
    printf("echo: %d\n", echo(4));
    return 0;
}
```

Functions



```
00401425  .  55              PUSH EBP
00401426  .  89E5            MOV EBP,ESP
00401428  .  53              PUSH EBX
00401429  .  51              PUSH ECX
0040142A  .  83EC 30         SUB ESP,0x30
0040142D  .  E8 3E060000     CALL a.00401A70
00401432  .  8D45 DC         LEA EAX,DWORD PTR SS:[EBP-0x24]
00401435  .  89C1            MOV ECX,EAX
00401437  .  E8 10290000     CALL a.00403D4C
0040143C  .  C745 DC C900(   MOV DWORD PTR SS:[EBP-0x24],0xC9
00401443  .  8D45 DC         LEA EAX,DWORD PTR SS:[EBP-0x24]
00401446  .  83C0 04         ADD EAX,0x4
00401449  .  C70424 455040   MOV DWORD PTR SS:[ESP],a.00405045             ASCII "Sonoo Jaiswal"
00401450  .  89C1            MOV ECX,EAX
00401452  .  E8 E1000000     CALL <JMP.&libstdc++-6._ZNSt7__cxx1112b.
00401457  .  83EC 04         SUB ESP,0x4
0040145A  .  8D45 DC         LEA EAX,DWORD PTR SS:[EBP-0x24]
0040145D  .  89C1            MOV ECX,EAX
0040145F  .  E8 8C280000     CALL a.00403CF0
00401464  .  C70424 040000   MOV DWORD PTR SS:[ESP],0x4
0040146B  .  E8 A0FFFFFF     CALL a.00401410
00401470  .  894424 04       MOV DWORD PTR SS:[ESP+0x4],EAX
00401474  .  C70424 535040   MOV DWORD PTR SS:[ESP],a.00405053            ASCII "echo: %d□"
0040147B  .  E8 50270000     CALL <JMP.&msvcrt.printf>                    printf
00401480  .  BB 00000000     MOV EBX,0x0
00401485  .  8D45 DC         LEA EAX,DWORD PTR SS:[EBP-0x24]
00401488  .  89C1            MOV ECX,EAX
0040148A  .  E8 D9280000     CALL a.00403D68
0040148F  .  89D8            MOV EAX,EBX
00401491  .v EB 16           JMP SHORT a.004014A9
00401493  .  89C3            MOV EBX,EAX
00401495  .  8D45 DC         LEA EAX,DWORD PTR SS:[EBP-0x24]
00401498  .  89C1            MOV ECX,EAX
0040149A  .  E8 C9280000     CALL a.00403D68
0040149F  .  89D8            MOV EAX,EBX
004014A1  .  890424          MOV DWORD PTR SS:[ESP],EAX
004014A4  .  E8 670C0000     CALL <JMP.&libgcc_s_dw2-1._Unwind_Resum
004014A9  >  8D65 F8         LEA ESP,DWORD PTR SS:[EBP-0x8]
004014AC  .  59              POP ECX
004014AD  .  5B              POP EBX
004014AE  .  5D              POP EBP
004014AF  .  8D61 FC         LEA ESP,DWORD PTR DS:[ECX-0x4]
004014B2  .  C3              RETN
```

Functions



```
00401330  >   55              PUSH EBP
00401331  .   89E5            MOV EBP,ESP
00401333  .   56              PUSH ESI
00401334  .   53              PUSH EBX
00401335  .   83EC 10         SUB ESP,0x10
00401338  .   C70424 00504    MOV DWORD PTR SS:[ESP],a.00405000            ASCII "libgcc_s_dw2-1.dll"
0040133F  .   E8 4C290000     CALL <JMP.&KERNEL32.GetModuleHandleA>       GetModuleHandleA
00401344  .   83EC 04         SUB ESP,0x4
00401347  .   85C0            TEST EAX,EAX
00401349  .v  74 75           JE SHORT a.004013C0
0040134B  .   C70424 00504    MOV DWORD PTR SS:[ESP],a.00405000            ASCII "libgcc_s_dw2-1.dll"
00401352  .   89C3            MOV EBX,EAX
00401354  .   E8 17290000     CALL <JMP.&KERNEL32.LoadLibraryA>           LoadLibraryA
00401359  .   83EC 04         SUB ESP,0x4
0040135C  .   A3 70704000     MOV DWORD PTR DS:[0x407070],EAX
00401361  .   C74424 04 13    MOV DWORD PTR SS:[ESP+0x4],a.00405013        ASCII "__register_frame_info"
00401369  .   891C24          MOV DWORD PTR SS:[ESP],EBX
0040136C  .   E8 17290000     CALL <JMP.&KERNEL32.GetProcAddress>         GetProcAddress
00401371  .   83EC 08         SUB ESP,0x8
00401374  .   89C6            MOV ESI,EAX
00401376  .   C74424 04 29    MOV DWORD PTR SS:[ESP+0x4],a.00405029        ASCII "__deregister_frame_info"
0040137E  .   891C24          MOV DWORD PTR SS:[ESP],EBX
00401381  .   E8 02290000     CALL <JMP.&KERNEL32.GetProcAddress>         GetProcAddress
00401386  .   A3 00404000     MOV DWORD PTR DS:[0x404000],EAX
0040138B  .   83EC 08         SUB ESP,0x8
0040138E  >   85F6            TEST ESI,ESI
00401390  .v  74 11           JE SHORT a.004013A3
00401392  .   C74424 04 08    MOV DWORD PTR SS:[ESP+0x4],a.00407008
0040139A  .   C70424 B8604    MOV DWORD PTR SS:[ESP],a.004060B8
004013A1  .   FFD6            CALL ESI
004013A3  >   C70424 E0134    MOV DWORD PTR SS:[ESP],a.004013E0
004013AA  .   E8 61FFFFFF     CALL <JMP.&msvcrt.atexit>                   atexit
004013AF  .   8D65 F8         LEA ESP,DWORD PTR SS:[EBP-0x8]
004013B2  .   5B              POP EBX
004013B3  .   5E              POP ESI
004013B4  .   5D              POP EBP
004013B5  .   C3              RETN
```

Functions

Pushing data on the stack: `mov` vs. `push`

- `push` always subtracts 4 from the `esp` register
- `mov` puts a value on the stack, but does not subtract from `esp`
- Optimization issue: small performance gain at runtime
    - When used with the `stdcall` convention, the caller must make special settings

Functions

Pushing data on the stack: `mov` vs. `push`

- `push` always subtracts 4 from the `esp` register
- `mov` puts a value on the stack, but does not subtract from `esp`
- Optimization issue: small performance gain at runtime
  - When used with the `stdcall` convention, the caller must make special settings

Inline functions

- Eliminate costly control transfers
- Useful for small functions, as their body is inlined with the caller's body
- No extra overhead for entry/exit
- Occurs often with string-related functions

Outline

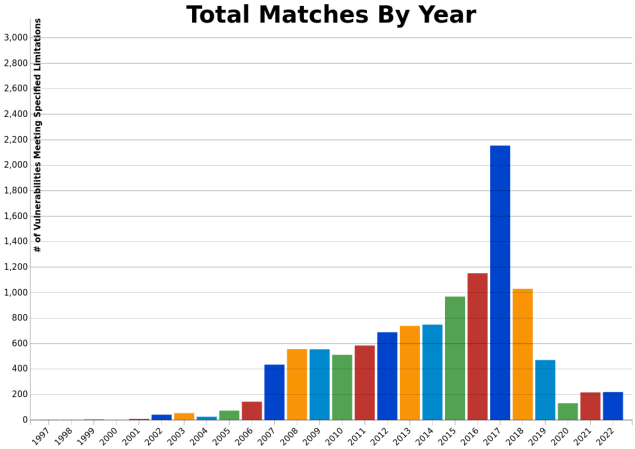
1 A little recap

2 Buffer Overflows

3 Defenses against Control-Flow Hijacking Attacks

Buffer overflows

Most common vulnerability in C/C++ programs

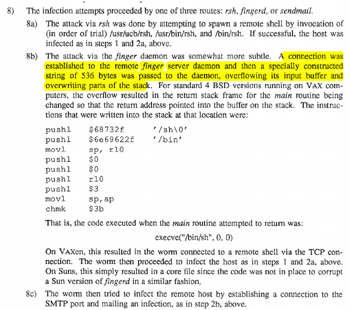


Credits: taken at 27/10/2022, https://nvd.nist.gov/

Buffer overflows

A bit of history – the first BOF exploited

- (BSD-derived) UNIX `fingerd` daemon
  - Utility that allows users to obtain information about other users
  - Usually used to identify the full name or login name of a user, whether a user is currently logged in or not, and other user information
- **Morris worm** (November 2 1988!)
  - Affected Sun 3 systems and VAX computers running 4 BSD UNIX variants
  - Exploited a buffer overflow in `fingerd` to create a remote shell



8) The infection attempts proceeded by one of three routes: *rsh*, *fingerd*, or *sendmail*.

8a) The attack via *rsh* was done by attempting to spawn a remote shell by invocation of (in order of trial) /usr/ucb/rsh, /usr/bin/rsh, and /bin/rsh. If successful, the host was infected as in steps 1 and 2a, above.

8b) The attack via the *finger* daemon was somewhat more subtle. A connection was established to the remote *finger* server daemon and then a specially constructed string of 536 bytes was passed to the daemon, overflowing its input buffer and overwriting parts of the stack. For standard 4 BSD versions running on VAX computers, the overflow resulted in the return stack frame for the *main* routine being changed so that the return address pointed into the buffer on the stack. The instructions that were written into the stack at that location were:

```
pushl   $68732f     '/sh\0'
pushl   $6e69622f   '/bin'
movl    sp, r10
pushl   $0
pushl   $0
pushl   r10
pushl   $3
movl    sp,ap
chmk    $3b
```

That is, the code executed when the *main* routine attempted to return was:

execve("/bin/sh", 0, 0)

On VAXen, this resulted in the worm connected to a remote shell via the TCP connection. The worm then proceeded to infect the host as in steps 1 and 2a, above. On Suns, this simply resulted in a core file since the code was not in place to corrupt a Sun version of *fingerd* in a similar fashion.

8c) The worm then tried to infect the remote host by establishing a connection to the SMTP port and mailing an infection, as in step 2b, above.

Further reading: *The internet worm program: an analysis*. E.H. Spafford. 1989. SIGCOMM Comput. Commun. Rev. 19, 1, 17–57. doi: 10.1145/66093.66095

Buffer overflows
What we need to know

- **How the stack works**
- **Calling conventions**
- **How system calls are made**

Anything else?. . .

- **Target system CPU**
  - Little-endian vs. big-endian
- **Target system operating system**
  - UNIX vs. Windows: stack frame changes!

Buffer overflows

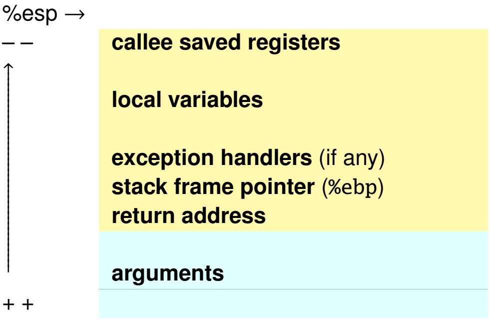
Linux x86 process memory layout

`0xFFFFFFFF`	kernel space
	(1GiB)
`0xC0000000`	user stack
⇓	
`0x40000000`	shared libraries
⇑	runtime heap
	bss
⇑	static data
`0x08048000`	(ELF binary loaded here)
0	unused

Check output of: `cat /proc/<PROCESS PID>/maps`

Buffer overflows

Stack frame



%esp →

– –

callee saved registers

local variables

exception handlers (if any)

stack frame pointer (`%ebp`)

return address

arguments

+ +

- **Stack Pointer (`%esp`)**: top of the stack
- **Base Pointer (`%ebp`)**: base of the current frame
- **Function arguments belong to the previous stack frame**
  - **Each function defines its own stack frame**

Note: *Stack grows to lower memory addresses*

Buffer overflows

Stack concepts summary

- The stack stores abstract data
- **Last-In-First-Out (LIFO) policy**
- **Assembly instructions of interest**:
  - `push`: **inserts** an item on top of the stack, **and decreases** `%esp` by 4 bytes (dword size)
  - `pop`: **eliminates** the item at the top of the stack, and **increments** `%esp` by 4 bytes

Buffer overflows

Stack concepts summary

- The stack stores abstract data
- **Last-In-First-Out (LIFO) policy**
- **Assembly instructions of interest**:
  - `push`: **inserts** an item on top of the stack, **and decreases** `%esp` by 4 bytes (dword size)
  - `pop`: **eliminates** the item at the top of the stack, and **increments** `%esp` by 4 bytes
  - `call`: **inserts** as the address of the next instruction which immediately follows the `call` on top of the stack, **and decreases** `%esp` **by 4 bytes**
  - **Return of functions.** `%esp` is incremented after execution. They accept an optional immediate value, which increments more %esp
    - `retn`: near return, **retrieves the top of the stack and sets it as %eip**
    - `retf`: far return, **retrieves two dwords from the top of the stack and sets them as %eip and `cs` (code segment), respectively**. Note that although `cs` is word size, it takes two dwords off from stack!

Buffer overflows

Stack concepts summary

On 32-bit architectures

- **Function arguments**
- **Return address**
- **Local variables**

On 64-bit architectures

- Also stores function arguments, **but differs from 32-bit architectures**:
  - **UNIX uses System V Application Binary Interface (ABI)**: first 6 integer (or pointer) arguments to a function are passed in registers (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`); from the 7th argument onwards, the stack is used
  - **Microsoft ABI**: only 4 registers are used (`%rcx`, `%rdx`, `%r8`, and `%r9`); from the 5th argument onwards, the stack is used
- **Return address**
- **Local variables**

Further reading: `http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64`

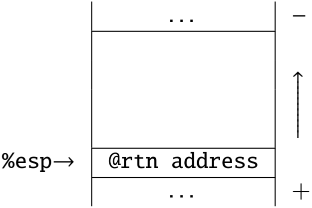
Buffer overflows

Example

```c
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```

```
readName:
        push    ebp
        mov     ebp, esp
        sub     esp, 264
        sub     esp, 12
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        sub     esp, 8
        lea     eax, [ebp-264]
        push    eax
        push    OFFSET FLAT:.LC1
        call    __isoc99_scanf
        add     esp, 16
        leave
        ret
```

%eip: push ebp



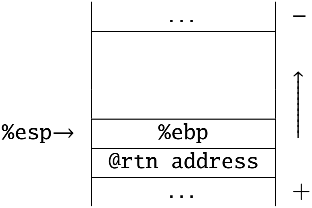
Buffer overflows

Example

```
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```

%eip: mov ebp, esp

```
readName:
        push    ebp
         mov ebp, esp
        sub     esp, 264
        sub     esp, 12
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        sub     esp, 8
        lea     eax, [ebp-264]
        push    eax
        push    OFFSET FLAT:.LC1
        call    __isoc99_scanf
        add     esp, 16
        leave
        ret
```



...

−

↑

%esp→ %ebp

@rtn address

...

+

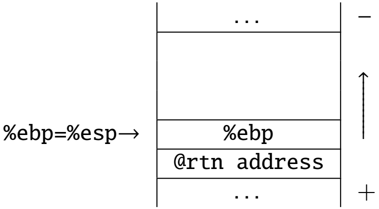
Buffer overflows

Example

```c
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```

%eip: sub esp, 264

```
readName:
        push    ebp
        mov     ebp, esp
        sub esp, 264
        sub     esp, 12
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        sub     esp, 8
        lea     eax, [ebp-264]
        push    eax
        push    OFFSET FLAT:.LC1
        call    __isoc99_scanf
        add     esp, 16
        leave
        ret
```



...

−

%ebp=%esp→ %ebp

@rtn address

...

+

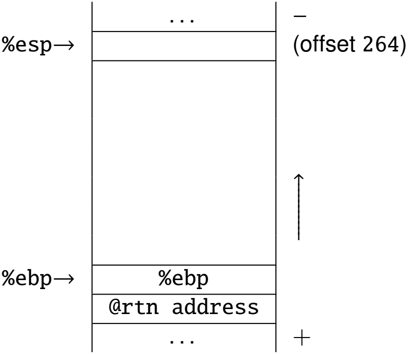
Buffer overflows

Example

```
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```

```
readName:
        push    ebp
        mov     ebp, esp
         sub esp, 264
        sub     esp, 12
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        sub     esp, 8
        lea     eax, [ebp-264]
        push    eax
        push    OFFSET FLAT:.LC1
        call    __isoc99_scanf
        add     esp, 16
        leave
        ret
```

`%eip:  sub esp, 264` (after)



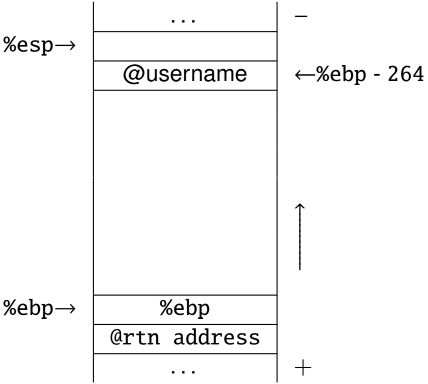
Buffer overflows

Example

```
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```

```
readName:
        push    ebp
        mov     ebp, esp
        sub     esp, 264
        sub     esp, 12
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        sub     esp, 8
        lea eax, [ebp-264]
        push    eax
        push    OFFSET FLAT:.LC1
        call    __isoc99_scanf
        add     esp, 16
        leave
        ret
```

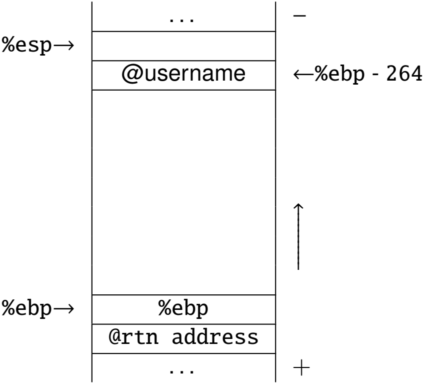
`%eip:  lea eax, [ebp-264]`



Buffer overflows

Example

```
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```

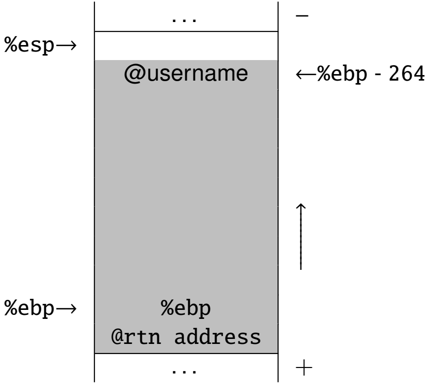


- What if *username* is more than 264 bytes long?

Buffer overflows

Example

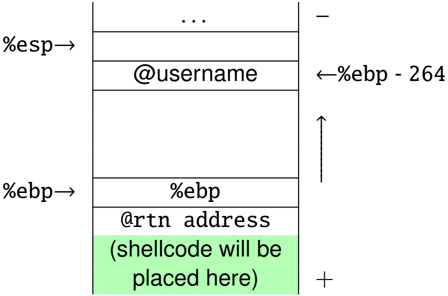
```
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```



- What if *username* is more than 264 bytes long?
  - **The adjacent memory to *username* is overwritten**, since `scanf` does not check for any buffer limits (it is an *insecure function*)
  - **Arbitrary code execution**, since `%eip` will pop the top value of the stack when the function returns!

Buffer overflows

Basic stack exploit



```
xor     eax, eax
push    eax
push    0x68732f2f
push    0x6e69622f
mov     ebx, esp
push    eax
push    ebx
mov     ecx, esp
mov     al, 0xb
int     0x80
```

NOTE: shellcode runs *on the stack*

1. **Insert your shellcode on the stack**
   - Shellcode: originally, the minimal code to launch a shell (i.e., exec("/bin/sh")). Today, any code injected regardless of its purpose
2. **Manipulate `@rtn address` to return to your shellcode**
   - Look for assembly instructions that allow redirection of execution to `%esp`
   - When the vulnerable function ends, the shellcode runs!

Further reading: *Smashing The Stack For Fun And Profit*. Aleph One, Phrack 49 (1996), `http://phrack.org/issues/49/14.html`

Buffer overflows

Insecure libc functions – (non-exhaustive list)

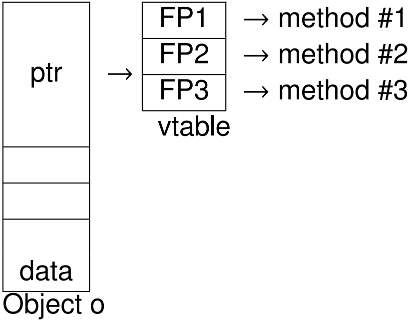
- `strcpy` → `strncpy` → `strlcpy/strcpy_s` (Windows CRT)
- `strcat` → `strncat` → `strlcat/strcat_s` (Windows CRT)
- `strtok`
- `sprintf` → `snprintf`
- `vsprintf` → `vsnprintf`
- `gets` → `fgets/gets_s`
- `scanf/sscanf` → `sscanf_s` (Windows CRT)
- `snscanf` → `_snscanf_s` (Windows CRT)
- `strlen` → `strnlen_s` (Windows CRT)

Some safe versions are misleading

- `strncpy`, `strncat` can leave strings unfinished – be careful!

Buffer overflows

Corrupting method pointers – Heap overflow



ptr →

FP1 → method #1

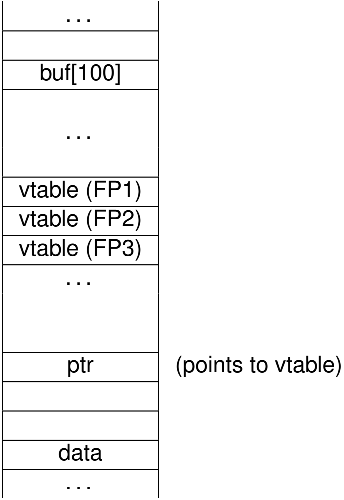
FP2 → method #2

FP3 → method #3

vtable

data

Object o



. . .

buf[100]

. . .

vtable (FP1)

vtable (FP2)

vtable (FP3)

. . .

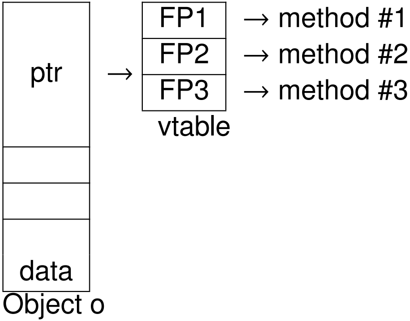
ptr (points to vtable)

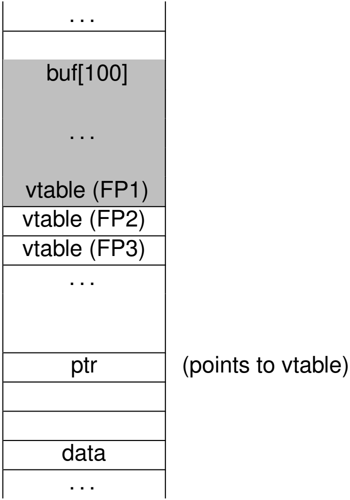
data

. . .

Buffer overflows

Corrupting method pointers – Heap overflow





Buffer overflows

How to hunt overflows…

Find the overflow

- Configure the operating system correctly (core dump?)
- Issue malformed inputs **with specific endings**
  - Automated tools (fuzzers)
- **If the application crashes, check the CPU registers for the endings**

Buffer overflows

How to hunt overflows...

Find the overflow

- Configure the operating system correctly (core dump?)
- Issue malformed inputs **with specific endings**
    - Automated tools (fuzzers)
- **If the application crashes, check the CPU registers for the endings**

Build the exploit

- **Analyze overflow conditions**
- Check if the overflow can lead to arbitrary code execution
    - Not easy, given the latest built-in defenses at the OS level

Outline

1 A little recap

2 Buffer Overflows

3 Defenses against Control-Flow Hijacking Attacks
- Stack Data Protection
- Non-Executable Stack
- Write XOR eXecute (W^X) Pages
- Address Space Layout
- Other Techniques of Defense

Universidad Zaragoza

Defeating control-flow hijacking attacks
Approaches

1. **Fix bugs**:
   - Audit software to find bugs (there are automated tools – soundness?)
   - Re-code software in a type-safe language
2. **Allow overflow, but prevent injected code from running**
3. **Insert runtime code to detect overflows**
   - Process stops when overflow is detected

Further readings: *SoK: Eternal War in Memory*. L. Szekeres, M. Payer, T. Wei and D. Song. 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, 2013, pp. 48–62. doi: 10.1109/SP.2013.13

Memory Errors: The Past, the Present, and the Future. V. van der Veen, N. dutt-Sharma, L. Cavallaro, H. Bos (2012). In Research in Attacks, Intrusions, and Defenses. RAID 2012. LNCS, vol 7462. Springer. doi: 10.1007/978-3-642-33338-5_5

Defeating control-flow hijacking attacks

Stack data protection

Stack cookies

- Detect stack-based overflows by:
  1. **In the function prologue, push a magic number**
  2. **In the function epilogue, check this value**

Defeating Control-Flow Hijacking Attacks

Stack cookies

- Initial ideas come from `StackGuard` (Crispin Cowan, 1997)
- **Enhanced by Hiroaki Etoh with `ProPolice`** (2000)
  - Later renamed to **SSP (Stack-Smashing Protector)**, included in mainstream GCC version 4.1
- **Types of canaries**:
  - **Null canary** (all zeros; `0x00000000`)
  - **Terminator canary** (`0x000aff0d`)
    - `0x00` stops `strcpy()` (and related functions)
    - `0x0a` and `0x0d` stop `gets()` (and related functions)
    - `0xff` (EOF) stops other functions
  - **Random canary**

Defeating control-flow hijacking attacks

Stack cookies

How to protect information stored after the vulnerable buffer?

- **Add a canary after each buffer and check each time** before accessing any other data stored after it
  - Good idea, may be a compiler modification
  - However, not practical: performance impact
- **Reorder local variables on the stack** to move the sensitive data out of the way of the buffer overflow
  - Side effect of compiler optimizations
  - Implemented as an intentional protection in `ProPolice`: *ideal stack layout*
    - Places local buffers at the end of the stack frame
    - Relocates other local variables before them
  - Also introduced by Microsoft Visual Studio (`/GS` feature)

Defeating control-flow hijacking attacks

Stack cookies

Ideal stack layout does not always exist...

- Multiple local buffers are placed one after another
- Structure members cannot be rearranged (interoperability issues)
- Particular structures (like arrays of pointers) can be overflowed or be treated as sensitive information, depends on the semantics
- Functions with a variable number of arguments remain unprotected
- Dynamically created buffers on the stack (e.g., `alloca()`) are placed at the top of the stack frame

Defeating control-flow hijacking attacks

Stack cookies

```
readName:
        push    ebp
        mov     ebp, esp
        sub     esp, 264
        sub     esp, 12
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        sub     esp, 8
        lea     eax, [ebp-264]
        push    eax
        push    OFFSET FLAT:.LC1
        call    __isoc99_scanf
        add     esp, 16
        leave
        ret
```

(stack cookies disabled)

```
readName:
        push    ebp
        mov     ebp, esp
        sub     esp, 280
        mov     eax, DWORD PTR gs:20
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        sub     esp, 12
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
        sub     esp, 8
        lea     eax, [ebp-268]
        push    eax
        push    OFFSET FLAT:.LC1
        call    __isoc99_scanf
        add     esp, 16
        mov     eax, DWORD PTR [ebp-12]
        xor     eax, DWORD PTR gs:20
        je      .L2
        call    __stack_chk_fail
.L2:
        leave
        ret
```

(stack cookies enabled)

Bypassing it is still possible

- On Windows, *SEH-based exploits*
- On UNIX-like systems, *we need a memory leak* (or bruteforce)

Defeating control-flow hijacking attacks

BOF exploitation steps

1. Place code in the stack (in the same vulnerable buffer)
2. Overwrite a return address
3. Jump to it

Defeating control-flow hijacking attacks

BOF exploitation steps

1. Place code in the stack (in the same vulnerable buffer)
2. Overwrite a return address
3. Jump to it

Non-executable stack

- First implemented for DEC on Alpha in Feb 1999
- **Enabled by default on most desktop platforms**, such as Linux, macOS, and Windows
- **Main weaknesses**:
  - **Still allows the return address to be abused**, overwriting it with an arbitrary location
  - **Does not prevent the execution of code** already present in the process memory or code injected in other data areas

Defeating control-flow hijacking attacks
Non-executable stack

Bypassing techniques

- `return-into-libc` (`ret2libc` for short)
  - Use `libc` function addresses as return addresses
  - The attacker does not require any shellcode to take control of a target, they simply redirect the execution of the control flow as they wish
  - We will talk about this more in deep in the last part of the course!
- **Improved techniques**:
  - `ret2plt`
  - `ret2syscall`
  - `ret2strcpy`, `ret2gets` (or `read()`, `recv()`, `recvfrom()` variants)
  - `ret2data`
  - `ret2text`, `ret2code`, `ret2dl-resolve`
  - Chained `ret2code` (or chained `ret2libc`)

Defeating control-flow hijacking attacks

W^X (memory) pages

- **Logical extension of non-executable stacks**
- **Non-executable writable pages and non-writable executable pages**

Defeating control-flow hijacking attacks

W^X (memory) pages

- **Logical extension of non-executable stacks**
- **Non-executable writable pages and non-writable executable pages**
- Term coined by Theo de Raadt (founder and main architect of OpenBSD)
- First implementation of W^X : 1972! (Multics on the GE-645 mainframe)

Defeating control-flow hijacking attacks

Write XOR eXecute (W^X) pages

- **The PaX project** (Oct 2000)
  - Linux kernel patch for Intel x86 hardware
  - Today, it is available for almost all hardware platforms
  - It was never included in mainstream Linux distribution, although today most distributions have some kind of W^X
- **On-chip support for non-executable pages** came a bit later
  - NX: Non-eXecutable feature (AMD Athlon 64; Sept 2003)
  - ED: Execute-Disable feature (Intel P4 Prescott; Feb 2004)
  - XN: eXecute-Never feature (ARM v6)
- **Software that took advantage of hardware support emerged** a few months later
  - Linux kernel patches (via PaX project)
  - Microsoft Windows XP Service Pack 2 (Data Execution Prevention; DEP – opt-in by default)

Defeating control-flow hijacking attacks

Can we still execute arbitrary injected code when W^X is on?

- *Do we really need to inject new code?* Otherwise, `ret2code`
- *Is there a page with W+X permissions?* If so, `ret2strcpy` or `ret2gets`
- *Can we chain the existing code, using* `ret2code`*, to write an executable file to disk and then run it?*

Defeating control-flow hijacking attacks

Can we still execute arbitrary injected code when W^X is on?

- *Do we really need to inject new code?* Otherwise, `ret2code`
- *Is there a page with W+X permissions?* If so, `ret2strcpy` or `ret2gets`
- *Can we chain the existing code, using `ret2code`, to write an executable file to disk and then run it?*
- *Is there a way to turn the protection off?*
  - `SetProcessDEPPolicy` / `ZwSetInformationProcess` on Windows platforms
- *Can we change the permissions of a specific memory region from W^X to W+X?*
  - `VirtualProtect` on Windows platforms
  - `mprotect` on GNU/Linux platforms
    - **note**: PaX **does not** allow a page to be W+X, nor X after W
    - In kernel, it requires the memory address to be aligned to 4KiB
- *Can we create a new memory region with W+X permissions?*
  - `VirtualAlloc()` on Windows platforms
  - `mmap()` on Unix-like platforms
    - As before, not allowed if PaX is installed
  - You will first need to copy the injected code and then jump there (chained `ret2code`: `mmap-strcpy-code`)

Defeating control-flow hijacking attacks

- `ret2libc` **allows us to bypass non-executable stacks**
  - Addresses of functions are known and are part of the attacker’s input

Defeating control-flow hijacking attacks

- `ret2libc` **allows us to bypass non-executable stacks**
  - Addresses of functions are known and are part of the attacker's input

ASCII Armored Address Space (AAAS)

- **Linux kernel patch that loaded all shared libraries into memory addresses starting with a null byte**
- Similar idea to terminator canaries
- Protects against `strcpy`-like exploitation, but not `gets`
- **Still vulnerable to other** `ret2-` **attacks**

Defeating control-flow hijacking attacks

Address Space Layout Randomization (ASLR)

- **Randomizes the address of everything** (libraries, image, stack, and heap)
- Prevents the attacker from knowing where to jump or where to point pointers
- First implemented in PaX for Linux in 2001:
  - "*unless every address is randomized and unpredictable, there's always going to be room for some kind of attack*"
- Introduced in Windows Vista (2007)
- **NOTE**: if the attacker can inject code and there is enough room for `nops`, **an approximate address can be enough** to achieve reliable code execution
  - This technique is known as `NOP-sled` or `NOP slide`

Defeating control-flow hijacking attacks
ASLR

- *Is there anything left in a predictable address?*
  - In most cases, yes:
    - Images are usually compiled to run in a fixed known memory address
    - No relocatable shared dynamic libraries
    - Improvement: **PIE (Position Independent Execution) code, on Linux platforms** (2005)
  - `ret2code` approaches
- *Can we guess the randomly generated addresses?*
  - It depends. Low entropy on 32-bits
  - On 32-bit Windows, even lower entropy
- *Is there a clever way to find these addresses?*
  - Is there a memory leak available?
  - Brute-forcing is always an option

Defeating control-flow hijacking attacks

ASLR

Some final remarks

- On Windows, threads of the same application share the memory layout
- On Unix, fork processes replicates the parent memory layout

ASLR is a very strong protection against code execution exploits, but most operating systems do not offer a complete solution

Defeating control-flow hijacking attacks
ASLR on Windows

Stack location:

- The time stamp counter (TSC) of the current processor is shifted and masked to a 5-bit value (2^5 options)
- Added to another 9-bit TSC-derived value to make up the base address of the stack

Heap location:

- TSC shifted and masked to a 5-bit value (2^5 options), multiplied by 64KiB
- The possible heap address ranges from `0x00000000` to `0x001f0000`

Defeating control-flow hijacking attacks

ASLR on Windows

Executable images location:

- **Load displacement by calculating a δ value each time an app runs**
- 8-bit pseudo-random number → only one of 256 possible locations
  - TSC shifts four places, and then divides modulo 254 and adds 1
  - The result is then multiplied by the allocation granularity of 64 KiB
- **This δ value is added to the preferred load address of the image file**

Defeating control-flow hijacking attacks

Address Space Layout Randomization (ASLR) in Windows

Shared libraries location:

- **Load offset is calculated with a system-wide per-boot value called the image bias**
  - Stored in a global memory state structure (`MI_SYSTEM_INFORMATION`), in field `MiState.Sections.ImageBias`)
- **Calculated only once per startup**
- **Shared memory region between** `0x50000000` **and** `0x78000000`
- First DLL is always `ntdll`. We can calculate its image base address as:
  - `0x78000000 - (ImageBias + NtDllSizein64KBChunks)*0x10000` (32-bit)
  - `0x7FFFFFFF0000 - (ImageBias64High + NtDllSizein64KBChunks)*0x10000` (64-bit)

Other techniques of defense

Probabilistic methods

- **Instruction Set Randomization**
- **Data Space Randomization**: randomizes the representation of data stored in memory (not location). Encrypts all variables, not just pointers, and using different keys

Other techniques of defense

Probabilistic methods

- **Instruction Set Randomization**
- **Data Space Randomization**: randomizes the representation of data stored in memory (not location). Encrypts all variables, not just pointers, and using different keys

Generic methods

- **Data Integrity**: spatial memory integrity (protect against invalid memory writes)
- **Data Flow Integrity**: checks read instructions to detect data corruption before use

Other techniques of defense

Probabilistic methods

- **Instruction Set Randomization**
- **Data Space Randomization**: randomizes the representation of data stored in memory (not location). Encrypts all variables, not just pointers, and using different keys

Generic methods

- **Data Integrity**: spatial memory integrity (protect against invalid memory writes)
- **Data Flow Integrity**: checks read instructions to detect data corruption before use

Other defenses against hijacking the flow of control

- **Code Pointer Integrity**
- **Control Flow Integrity** (CFI)

Exploiting Software Vulnerabilities

Software Vulnerabilities

CONTROL-FLOW HIJACKING

Universidad Zaragoza
1542
Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2022/2023

Master's Degree in Informatics Engineering
UNIVERSITY OF ZARAGOZA
Seminar A.22, Ada Byron building