Exploiting Software Vulnerabilities

Software Vulnerabilities

Buffer and Integer Overflows

Universidad Zaragoza

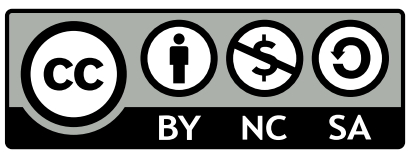
Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2020/2021

Master's Degree in Informatics Engineering

University of Zaragoza

Google Meet



Outline

Universidad Zaragoza

Outline

Universidad Zaragoza

Recap on...

Definitions

System/defender perspective

- **Attack surface**
  - Exposition 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 on taking advantage of a vulnerability

Universidad Zaragoza

Recap on...

Vulnerabilities

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`)
- ...

Universidad Zaragoza

Today we're talking about...

Control-Flow Hijacking

- Attacker's goal: **take over target system**
  - **Execute arbitrary code to hijack the control-flow execution of a vulnerable application**

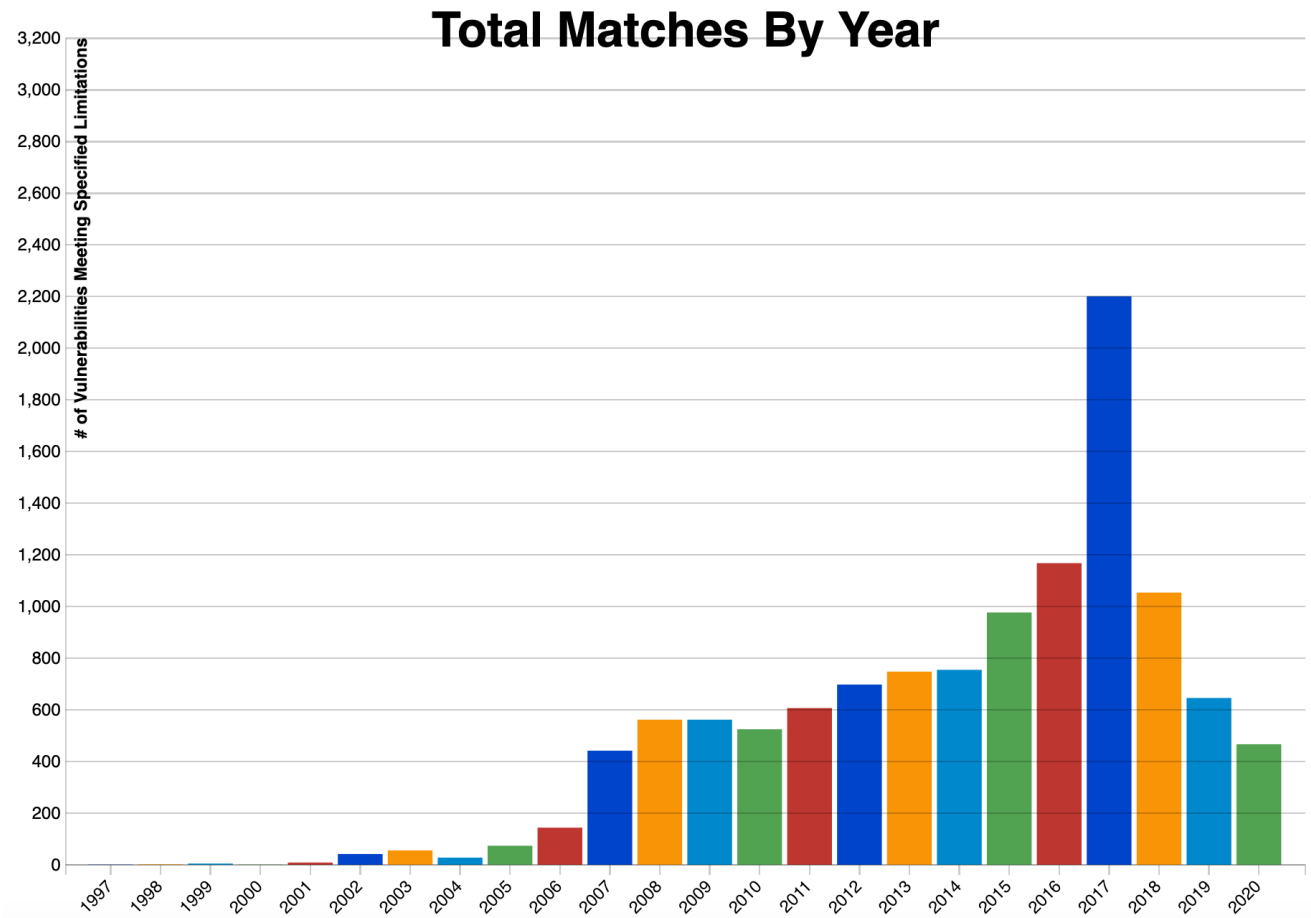
Outline

Universidad Zaragoza

Buffer overflows

Most common vulnerability in C/C++ programs

- Results Type: Statistics
- Category (CWE): CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer



Credits: taken at 03/10/2020, https://nvd.nist.gov/

Universidad Zaragoza

Buffer overflows

A piece 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 or not a user is currently logged in, and other user's information
- **Morris worm** (in Nov 2 1988!)
  - Affected to Sun 3 systems, and VAX computers running variants of 4 BSD UNIX
  - Exploited a buffer overrun 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 stack works**
- **Calling conventions**
- **How syscalls are done**

Something else?...

- **CPU of the target system**
  - Little-endian vs. big-endian
    - Here, **we will consider x86** (in lectures)
- **OS of the target system**
  - UNIX vs. Windows: stack frame changes!
    - Here, **we will consider UNIX-based for theory lessons**

Universidad Zaragoza

Buffer overflows

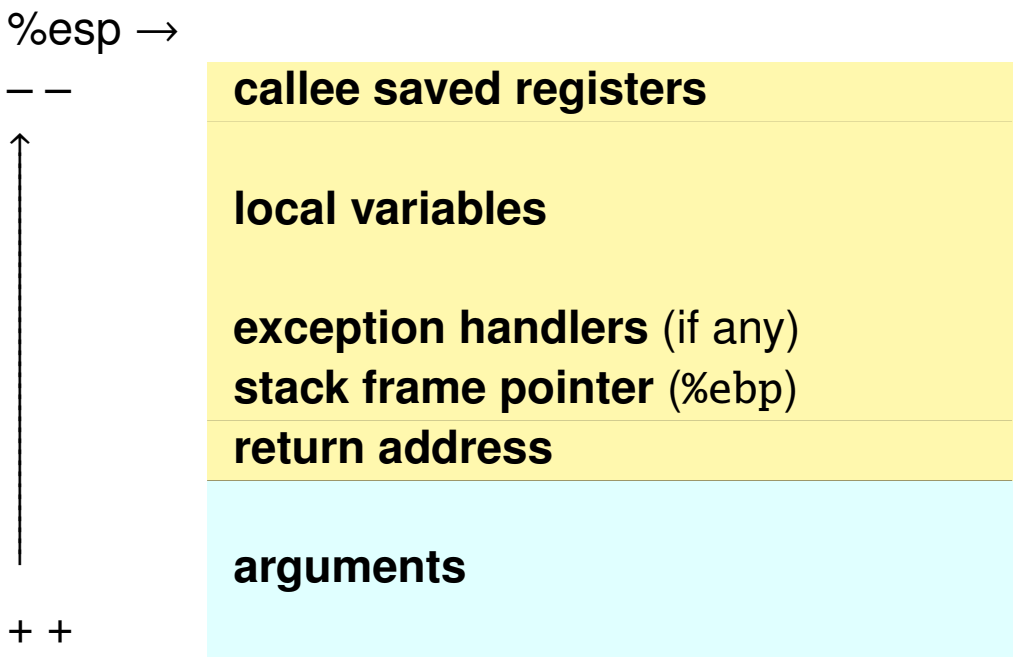
Linux x86 process memory layout

`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 command `cat /proc/<PROCESS PID>/maps`

Buffer overflows

Stack frame



- **Stack Pointer (`%esp`)**: top of the stack
- **Base Pointer (`%ebp`)**: base of 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

Recap on stack concepts

- Stack stores abstract data
- **Last-In-First-Out (LIFO) policy**
- **Assembly instructions of interest**:
  - `push`: **inserts** a new element in the top of the stack, **and decrements** `%esp` in 4 bytes (size of dword)
  - `pop`: **eliminates** the element in the top of the stack, and **increments** `%esp` in 4 bytes (size of dword)

Universidad Zaragoza

Buffer overflows

Recap on stack concepts

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

Universidad Zaragoza 1542

Buffer overflows

Recap on stack concepts

On 32-bit architectures

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

On 64-bit architectures

- Also stores function arguments, **but differs from 32-bit archs**:
  - **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 7th argument onward, stack is used
  - **Microsoft ABI**: only 4 registers are used (`%rcx`, `%rdx`, `%r8`, and `%r9`); from 5th argument onward, stack is used
- **Return address**
- **Local variables**

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

Universidad Zaragoza

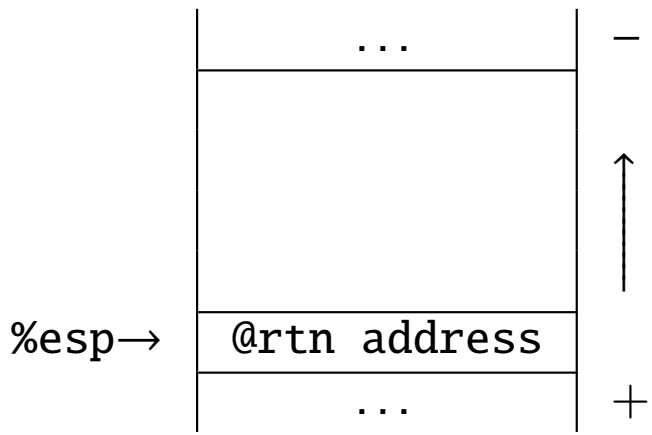
Buffer overflows

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:  push ebp
```



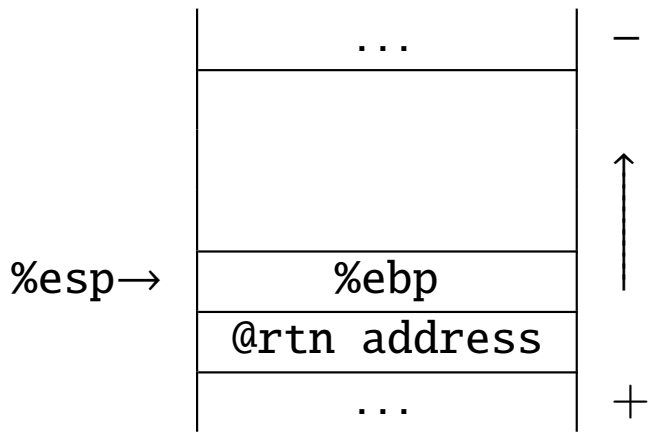
Buffer overflows

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: mov ebp, esp



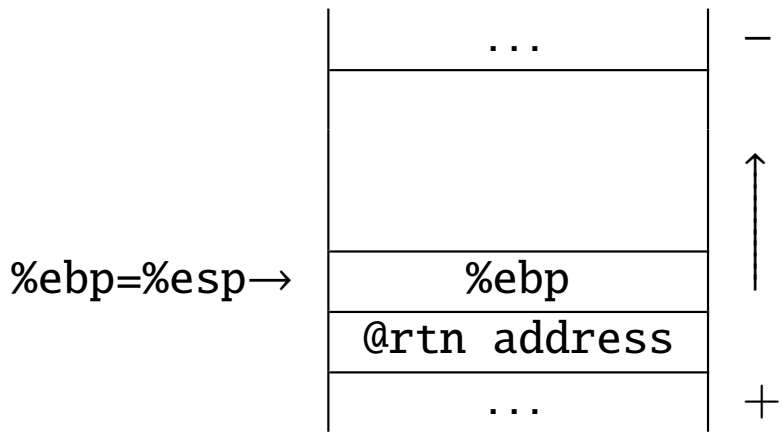
Buffer overflows

Example

```
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
```



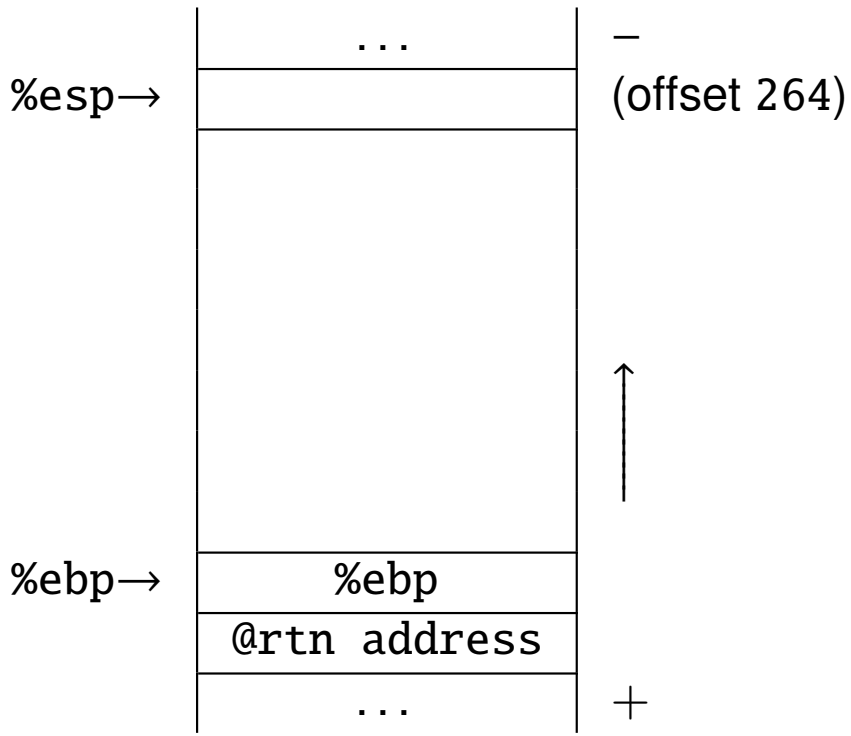
Buffer overflows

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)



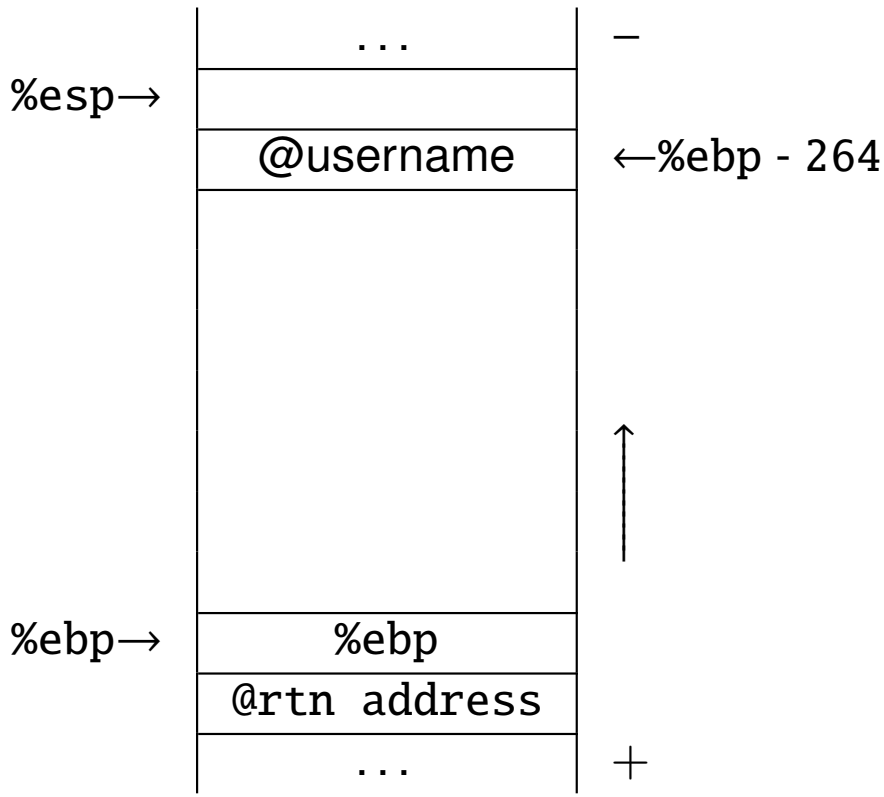
Buffer overflows

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
```

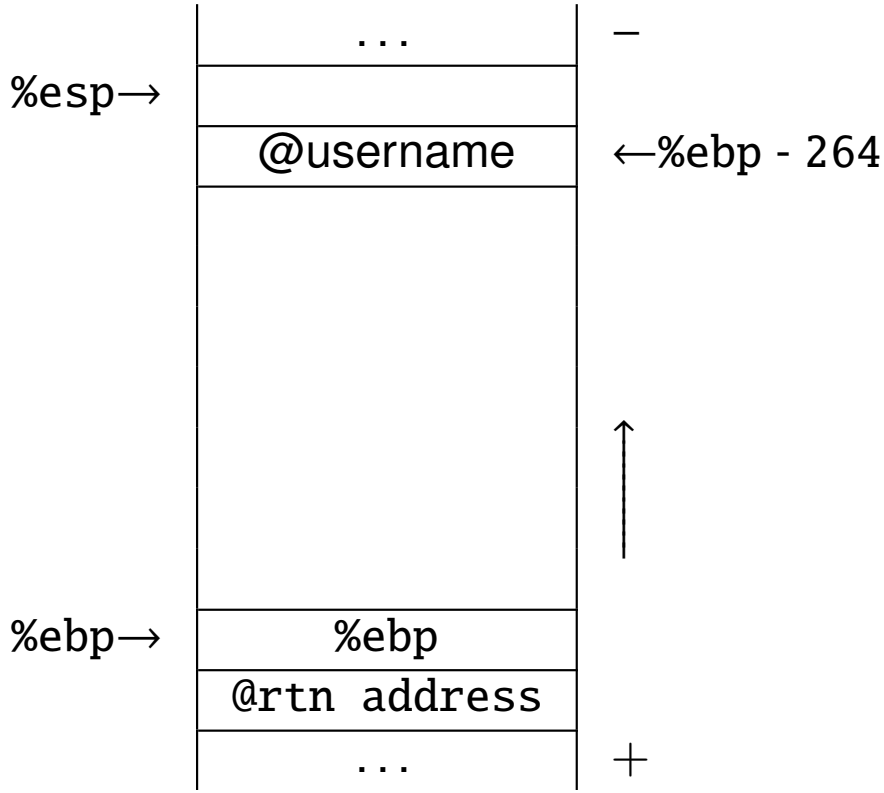
```
%eip:  lea eax, [ebp-264]
```



Buffer overflows

Example

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

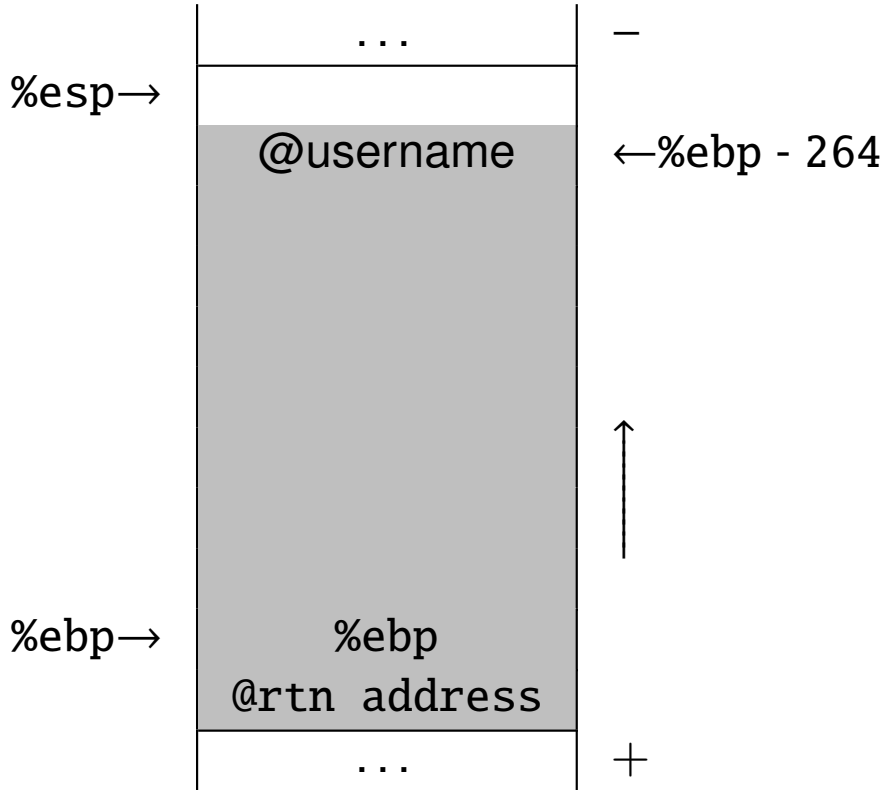


- What if *username* is > 264 bytes long?

Buffer overflows

Example

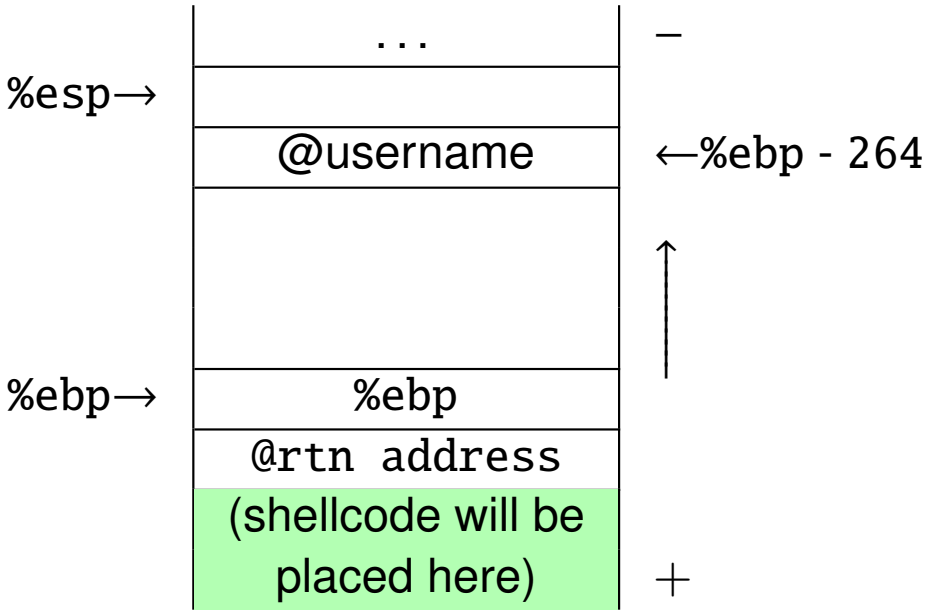
```
void readName(){
  char username[256];
  printf("Type user name: ");
  scanf("%s", username);
}
```



- What if *username* is > 264 bytes long?
  - **Adjacent memory to *username* is overwritten**, since `scanf` does not check any buffer boundary (it is an *unsafe function*)
  - **Arbitrary code execution**, since `%eip` will pop the value from stack when 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 *in the stack*

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

Further reading: *Smashing The Stack For Fun And Profit.* Aleph One, Phrack 49, vol. 7, Nov. 1996,

`http://phrack.org/issues/49/14.html`

Buffer overflows

Some details on shellcodes

- In this example, **shellcode cannot contain NULL characters**
  - Also CR (`0x0D`), LF (`0x0A`) bytes are invalid, in case of string-based shellcodes given as inputs
  - Other constraints may arise: only numbers, only letters, ...
- **Overflow should not crash execution** (at least until the vulnerable function ends)

Universidad Zaragoza

Buffer overflows

Unsafe libc functions – (non-exhaustive list)

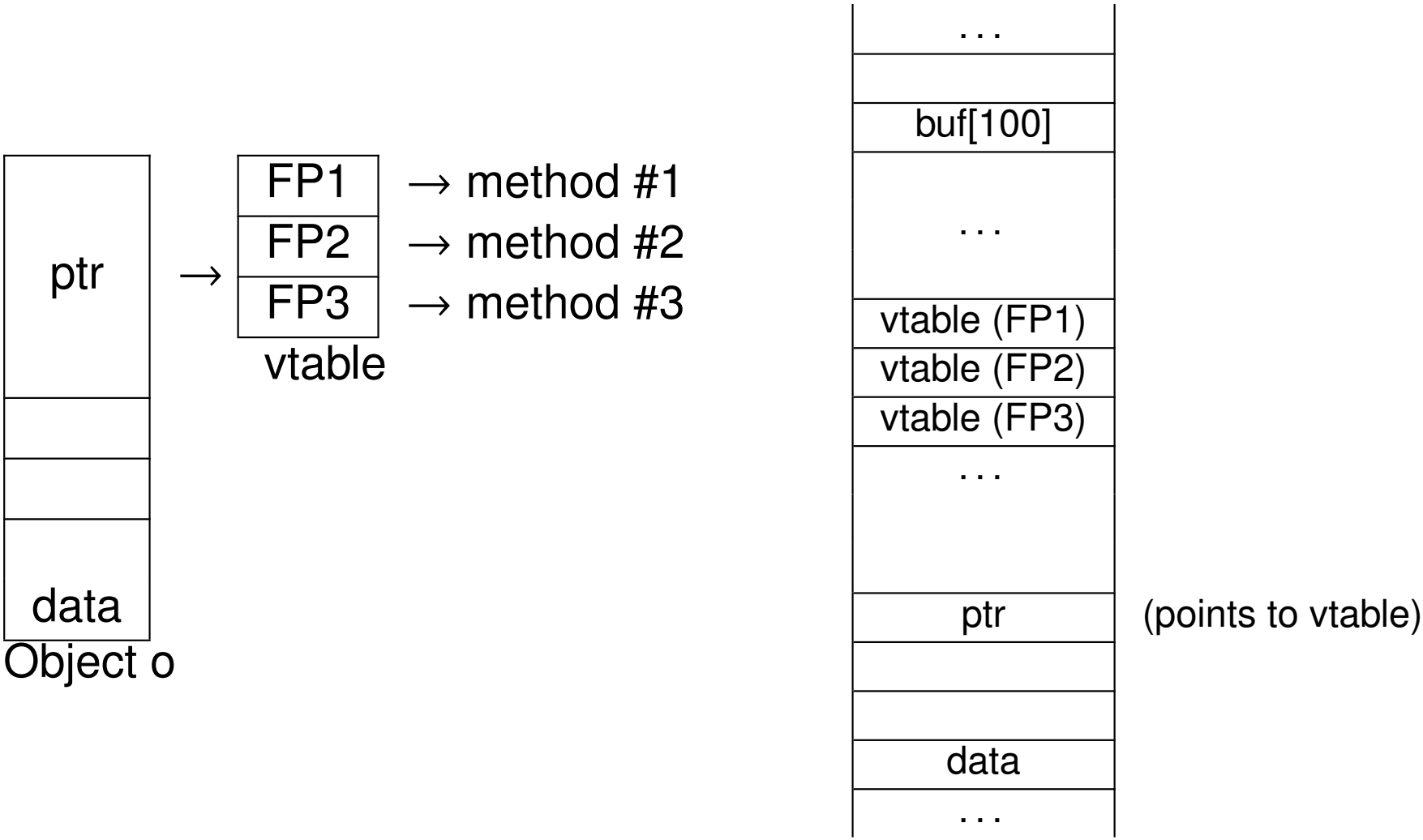
- `strcpy` → `strncpy` → `strlcpy/strcpy_s` (Windows CRT)
- `strcat` → `strncat` → `strlcat/strcat_s` (Windows CRT)
- `strtok`
- `sprintf` → `snprintf`
- `vsprintf` → `vsnprintf`
- `gets` → `fgets/gets_s` CRT)
- `scanf/sscanf` → `sscanf_s` (Windows CRT)
- `snscanf` → `_snscanf_s` (Windows CRT)
- `strlen`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

Some safe versions are misleading

- `strncpy`, `strncat` may leave strings unterminated – be cautious!

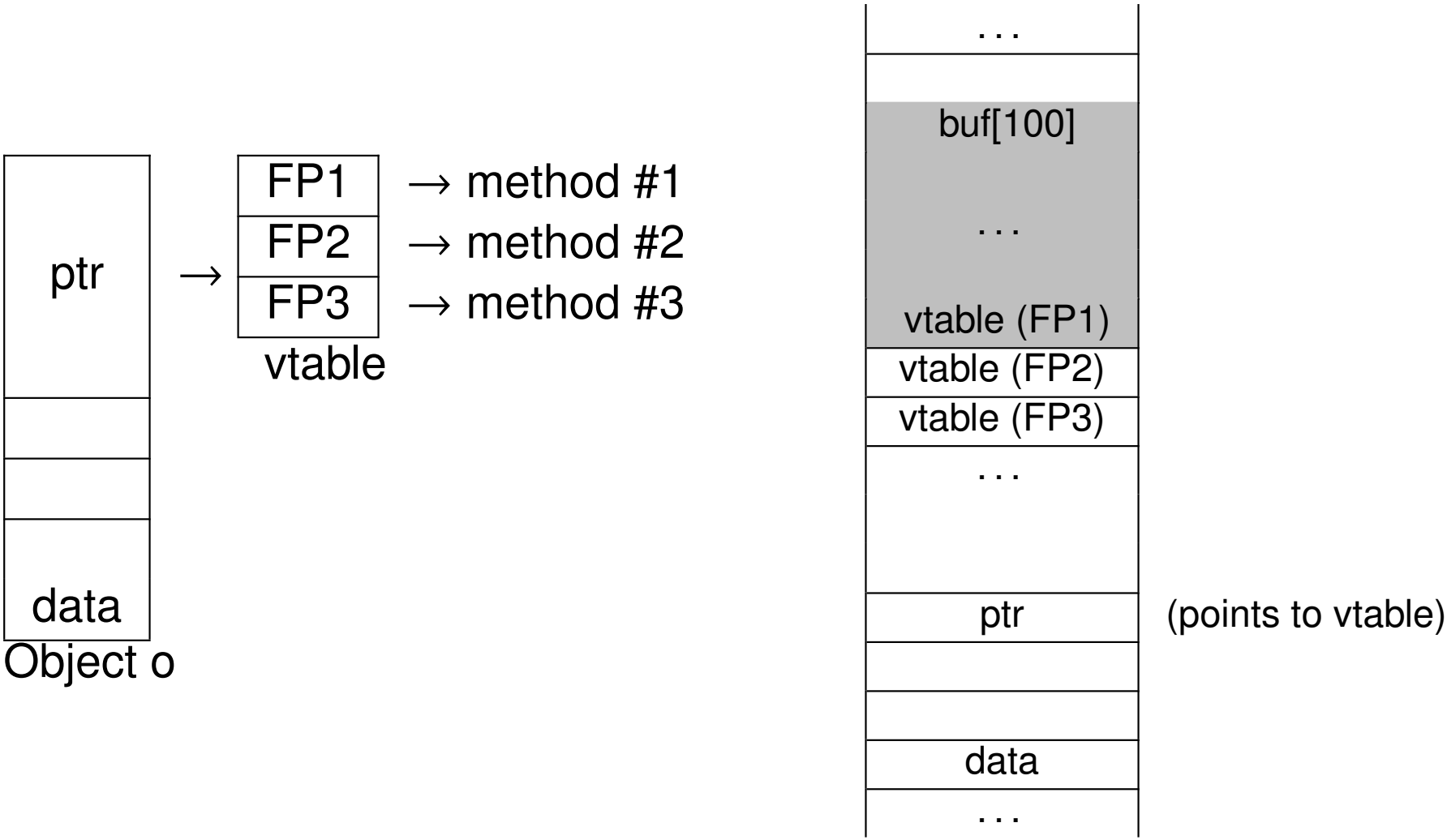
Buffer overflows

Corrupting method pointers – Heap overflow



Buffer overflows

Corrupting method pointers – Heap overflow



Buffer overflows

How to hunt overflows...

Find the overflow

- Configure the OS properly (core dump?)
- Issue malformed inputs **with specific endings**
  - Automated tools (fuzzers)
- **When application crashes, look for such a specific ending**

Buffer overflows

How to hunt overflows...

Find the overflow

- Configure the OS properly (core dump?)
- Issue malformed inputs **with specific endings**
  - Automated tools (fuzzers)
- **When application crashes, look for such a specific ending**

Build the exploit

- **Analyze the overflow conditions**
- Check if overflow may lead to arbitrary code execution
  - Not easy, given the last defenses incorporated at OS level

Universidad Zaragoza

Defeating control-flow hijacking attacks

Approaches

1. **Fix bugs**:
   - Audit software to find bugs (automated tools exist – soundness?)
   - Re-code software in a type-safe language
2. **Allow overflow, but prevent the execution of injected code**
3. **Insert run-time code to detect overflows**
   - Process is stopped when the overflow is detected

Universidad Zaragoza

Defeating control-flow hijacking attacks

Approaches

1. **Fix bugs**:
   - Audit software to find bugs (automated tools exist – soundness?)
   - Re-code software in a type-safe language
2. **Allow overflow, but prevent the execution of injected code**
3. **Insert run-time code to detect overflows**
   - Process is stopped when the overflow is detected

Wait a minute...

Everything happens in the stack, so how can we protect the stack?

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

Universidad Zaragoza 1542

Defeating control-flow hijacking attacks

Stack cookies

Concept

- Detect stack-based overflows by:
  1. **Insert in the stack a magic number (done in the function prologue)**
  2. **Check such a value in the function epilogue**

Software-based implementation

- Compiler flag
  - `-fstack-protector` (gcc)
  - /GS (Microsoft VC++)

Universidad Zaragoza

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)

Defeating it is still possible

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

Universidad Zaragoza

Defeating Control-Flow Hijacking Attacks

Stack cookies

Ok, that's a nice idea. But, first of all...

why the heck the stack has executable permissions?

Defeating control-flow hijacking attacks

Write XOR eXecute (W^X) memory pages

Concept

- Stack (and heap) stores data to be read/written, but never executed ⇒ **Stack and heap memory zones are marked as non-executable**

Hardware support (a special bit is added in every Page Table Entry of committed memory)

- **NX-bit** on AMD Athlon 64
- **XD-bit** on Intel P4 Prescott
- **XN-bit** on ARM v6

Software support

- **Linux** (via PaX project); OpenBSD
- **Windows, since XP SP2** (Data Execution Prevention, DEP)
  - /NXCOMPAT compiler flag (MSVC)

Limitations: some programs may need an executable heap (e.g., JIT)

Defeating control-flow hijacking attacks

Breaking W⊕X protection

Control-flow is redirected to the stack

- W⊕X prevents execution. Roughly speaking, you (as attacker) are fucked up

Defeating control-flow hijacking attacks

Breaking W⊕X protection

Control-flow is redirected to the stack

- W⊕X prevents execution. Roughly speaking, you (as attacker) are fucked up

Wait a minute!

Defeating control-flow hijacking attacks

Breaking W⊕X protection

Control-flow is redirected to the stack

- W⊕X prevents execution. Roughly speaking, you (as attacker) are fucked up

Wait a minute!

IDEA

Since we can write the stack... and stack also stores the return addresses of the control-flow when (legitimately) diverted... **can we use memory addresses pointing to ALREADY EXISTING code?**

Defeating control-flow hijacking attacks

Breaking W⊕X protection

Control-flow is redirected to the stack

- W⊕X prevents execution. Roughly speaking, you (as attacker) are fucked up

Wait a minute!

IDEA

Since we can write the stack... and stack also stores the return addresses of the control-flow when (legitimately) diverted... **can we use memory addresses pointing to ALREADY EXISTING code?** → **Yes!**

Defeating control-flow hijacking attacks

Breaking W⊕X protection

Control-flow is redirected to the stack

- W⊕X prevents execution. Roughly speaking, you (as attacker) are fucked up

Wait a minute!

IDEA

Since we can write the stack... and stack also stores the return addresses of the control-flow when (legitimately) diverted... **can we use memory addresses pointing to ALREADY EXISTING code?** → **Yes!**

Return-Oriented Programming (ROP)

Defeating control-flow hijacking attacks

Breaking W⊕X protection

Control-flow is redirected to the stack

- W⊕X prevents execution. Roughly speaking, you (as attacker) are fucked up

Wait a minute!

IDEA

Since we can write the stack... and stack also stores the return addresses of the control-flow when (legitimately) diverted... **can we use memory addresses pointing to ALREADY EXISTING code?** → **Yes!**

Return-Oriented Programming (ROP)

- Code resides in memory pages that already have execution privileges
- Since these pages can execute, **they are not covered by W⊕X protection**

Defeating control-flow hijacking attacks

Return-Oriented-Programming attacks

ROP enables an adversary to induce arbitrary execution behavior while injecting no code (just pointers to existing code!)

Universidad Zaragoza

Defeating control-flow hijacking attacks

Return-Oriented-Programming attacks

ROP enables an adversary to induce arbitrary execution behavior while injecting no code (just pointers to existing code!)

ROP attacks

- Hijack control-flow **without executing new binary code**
- **Redirect control-flow to chunks of code already available in the memory space of the process**
  - Recall **x86 ISA has variable size!**
  - ROP gadget: set of instructions that ends with `retn`

Defeating control-flow hijacking attacks

Return-Oriented-Programming attacks

ROP enables an adversary to induce arbitrary execution behavior while injecting no code (just pointers to existing code!)

ROP attacks

- Hijack control-flow **without executing new binary code**
- **Redirect control-flow to chunks of code already available in the memory space of the process**
  - Recall **x86 ISA has variable size!**
  - ROP gadget: set of instructions that ends with `retn`

```
b8 89 41 08 c3      mov eax, 0xc3084189


89 41 08            mov [ecx+8], eax
c3                  ret
```

Defeating control-flow hijacking attacks

Return-Oriented-Programming attacks

ROP enables an adversary to induce arbitrary execution behavior while injecting no code (just pointers to existing code!)

ROP attacks

- Hijack control-flow **without executing new binary code**
- **Redirect control-flow to chunks of code already available in the memory space of the process**
  - Recall **x86 ISA has variable size!**
  - ROP gadget: set of instructions that ends with `retn`

```
                                                        ...
                                                esp →   0x7c37638d   → pop ecx; ret
b8 89 41 08 c3      mov eax, 0xc3084189                 0xF13C1A02
                                                        0x7c341591   → pop edx; ret
                                                        0xBAADF00D
89 41 08            mov [ecx+8], eax                    0x7c367042   → xor eax, eax; ret
c3                  ret                                 0x7c34779f   → add eax, ecx; ret
                                                        0x7c347f97   → mov ebx, eax; ret
                                                        ...
```

Defeating control-flow hijacking attacks

- **Adversary controls the order of execution of ROP gadgets**
- **ROP chain**: set of ROP gadgets chained by the adversary

Universidad Zaragoza

Defeating control-flow hijacking attacks

- **Adversary controls the order of execution of ROP gadgets**
- **ROP chain**: set of ROP gadgets chained by the adversary
- *Can we bypass the W⊕X protection?*
  - Build a ROP chain to deactivate the protection! First, set CPU registers to specific values. Then,
    - Execute `memprot()` syscall (in GNU/Linux)
    - Execute *SetDEPProcessPolicy()* (in Windows)
    - ...

Defeating control-flow hijacking attacks

- **Adversary controls the order of execution of ROP gadgets**
- **ROP chain**: set of ROP gadgets chained by the adversary
- *Can we bypass the W⊕X protection?*
  - Build a ROP chain to deactivate the protection! First, set CPU registers to specific values. Then,
    - Execute `memprot()` syscall (in GNU/Linux)
    - Execute *SetDEPProcessPolicy()* (in Windows)
    - ...
  - In theory, yes. In practice, the current software defenses incorporated in the modern OS make it very difficult (although not impossible)

Defeating control-flow hijacking attacks

- **Adversary controls the order of execution of ROP gadgets**
- **ROP chain**: set of ROP gadgets chained by the adversary
- *Can we bypass the W⊕X protection?*
  - Build a ROP chain to deactivate the protection! First, set CPU registers to specific values. Then,
    - Execute `memprot()` syscall (in GNU/Linux)
    - Execute *SetDEPProcessPolicy()* (in Windows)
    - ...
  - In theory, yes. In practice, the current software defenses incorporated in the modern OS make it very difficult (although not impossible)

Executional adversary power

- **Existing binary code in the process's memory space determines what an adversary can do**

Universidad Zaragoza
1542

Defeating control-flow hijacking attacks

ROP attacks

Ok, that's a great idea. But, note that the adversary needs to know where the binary code is loaded...

why the heck the binary code is always loaded in the same addresses?

Defeating control-flow hijacking attacks

Address Space Layout Randomization (ASLR)

Concept

- **Map shared libraries to random locations in process memory**
- **PIE (Position Independent Execution) in Linux** (introduced in 2005)
  - Deactivation: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
  - Activation: echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
- Introduced in Windows Vista (2007)

Software support

- `-pie`, `-fpie` (gcc); `/DYNAMICBASE` (MSVC)

Limitations

- **Low randomness space size** (in 32 bits)
- **Some libraries may not accept ASLR enabled!**

Defeating control-flow hijacking attacks

Address Space Layout Randomization (ASLR) in Windows

Stack location:

- Current processor's time stamp counter (TSC) shifted and masked into a 5-bit value (1/32 choices)
- Added to another 9-bit TSC-derived value to conform the final stack base address

Heap location:

- TSC shifted and masked into a 5-bit value (1/32 choices), multiplied by 64KiB
- Possible heap address ranges from `0x00000000` to `0x001f0000`

Universidad Zaragoza

Defeating control-flow hijacking attacks

Address Space Layout Randomization (ASLR) in Windows

Executable images location:

- **A load offset is calculated by computing a delta value each time an executable is loaded**
- Pseudo-random 8-bit number → only one of 256 possible locations
  - TSC is shifted by four places, and then divided modulo 254 and added 1
  - The result is then multiplied by the allocation granularity of 64 KiB
- **This delta value is added to the preferred load address of the executable**

Universidad Zaragoza

Defeating control-flow hijacking attacks

Address Space Layout Randomization (ASLR) in Windows

Shared libraries location:

- **The load offset is computed with a per-boot, system-wide value called the image bias**
  - Stored in a global memory state structure (`MI_SYSTEM_INFORMATION`) in the `MiState.Sections.ImageBias` field)
- **Computed only once per boot**
- **Shared memory region between `0x50000000` and `0x78000000`**
- First DLL is always `ntdll`. We can compute its image base address as:
  - In 32 bits: `0x78000000 - (ImageBias + NtDllSizein64KBChunks)*0x10000`
  - In 64 bits: `0x7FFFFFF0000 - (ImageBias64High + NtDllSizein64KBChunks)*0x10000`

Universidad Zaragoza

Other techniques of defense

Probabilistic methods

- **Instruction Set Randomization**
- **Data Space Randomization**: randomizes the representation of data stored in memory (not the location). Encrypts all variables, not only 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 the location). Encrypts all variables, not only 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 it gets used

Other techniques of defense

Probabilistic methods

- **Instruction Set Randomization**
- **Data Space Randomization**: randomizes the representation of data stored in memory (not the location). Encrypts all variables, not only 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 it gets used

Other control-flow hijacking defenses

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

Universidad Zaragoza

Outline

Universidad Zaragoza

Integer Vulnerabilities

Types of integer vulnerabilities

- **Overflow**
  - Occurs at run-time when the result of an integer expression exceeds the maximum value for its respective type
  - For instance, two unsigned 8-bit integers may require up to 16 bits
- **Underflow**
  - Occurs at run-time when the result of an integer expression is smaller than its minimum value. Thus, it wraps to the maximum integer for the type
  - For instance, subtracting $0 - 1$ and storing the result in an unsigned 16-bit integer will result in a value of $2^{16} - 1$ (not -1)
- **Truncation**
  - Occurs when assigning an integer with a larger width to a smaller width
  - For instance, casting an `int` to a `short` discards the leading bits of the `int` value, resulting in (potential) information loss
- **Signedness error**
  - Occurs when a signed integer is interpreted as unsigned, or vice-versa
  - In two-complement representation, such conversions cause the sign bit to be interpreted as the most significant bit (i.e., $2^{32} - 1 \neq -1$)

Universidad Zaragoza 1542

Integer Vulnerabilities

Examples

```
struct pixmap {
  unsigned char *p;
  int x;
  /* xsize */
  int y;
  /* ysize */
  int bpp;
};
typedef struct pixmap pix;
......
void readpgm(char *name, pix * p) {
   /* read pgm */...
   pnm_readpaminit(fp, &inpam);
  p->x=inpam.width;
  p->y=inpam.height;
   if(!(p->p=(char *)malloc(p->x*p->y)))
     F1("Error at malloc");
   for(i=0; i<inpam.height; i++){
     pnm_readpamrow(&inpam, tuplerow);
   for(j = 0; j<inpam.width; j++)
     p->p[i*inpam.width+j]=sample;
  }
}
```

```
void getComm(unsigned int len, char *src){
   unsigned int size;
   size = len - 2;
   char *comm = (char *)malloc(size + 1);
   memcpy(comm, src, size);
   return;
}
```

```
static inline u32 *decode_fh(u32 *p, struct svc_fh *fhp) {
   int size;
   fh_init(fhp, NFS3_FHSIZE);
   size = ntohl(*p++);
   if (size > NFS3_FHSIZE)
      return NULL;
   memcpy(&fhp->fh_handle.fh_base, p, size);
   fhp->fh_handle.fh_size = size;
   return p + XDR_QUADLEN(size);
}
```

```
int detect_attack(u_char *buf, int len, u_char *IV){
   static word16 *h = (word16 *) NULL;
   static word16 n = HASH_MIN_ENTRIES;
   register word32 i, j;
   word32 l;
   ...
   for(l=n; l<HASH_FACTOR(len/BSIZE); l=l<<2);
   if (h == NULL) {
      debug("Install crc attack detector.");
      n = l;
      h = (word16 *) xmalloc(n*sizeof(word16));
   } else
      for (c=buf, j=0; c<(buf+len); c+=BSIZE, j++){
         for (i = HASH(c) & (n - 1); h[i] != UNUSED; i = (i + 1) & (n - 1))
            ...;
         h[i] = j;
      }
   }
```

Integer Vulnerabilities

Exploiting integer bugs

Usually, they are exploited indirectly

- **Arbitrary code execution**
  - For instance, insufficient memory allocation exploited by buffer overflows, heap overflows, overwrite attacks, etc.
- **Denial-of-Service**
  - For instance, excessive memory allocation or infinite loops
- **Bypassing sanitization attacks**
  - For instance, bypassing an upper bounds check that does not take into account unexpected negative integer values
- **Logic errors**
  - For instance, manipulating reference counter forcing a referenced object to be freed prematurely

Further reading: *Understanding Integer Overflow in C/C++*. W. Dietz et al. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 2

(December 2015) doi: 10.1145/2743019

Universidad Zaragoza

Integer Vulnerabilities

Consequences

- **Silent breakage**
  - Compiler optimizations may result in exploiting undefined behavior
- **Time bombs**
  - Today works, but improvements on optimization technologies may exploit it
- **Illusion of predictability**
  - Some compilers, at some optimization levels, have predictable behavior for some undefined operations
- **Informal dialects**
  - Some compilers have flags to force two-complement behavior in signed overflows
- **Non-standard standards**
  - Meaning of overflow changes over standards (e.g., $1 << 31$)
    - Implementation-defined in ANSI C and C++98
    - Undefined by C99 and C11

Universidad Zaragoza

Integer vulnerabilities

Wraparound – unsigned data types

- **Any computation involving unsigned operands can never overflow**
- *What if the result of an operation cannot be represented by the resulting unsigned integer type?*
  - **The result is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type**
- It can occur with **addition and multiplication operations**
  - n-bit add/sub operation needs $n + 1$ bits of precision
  - Similarly, n-bit multiplication needs $2n$ bits of precision

Universidad Zaragoza

Integer vulnerabilities

Wraparound – unsigned data types

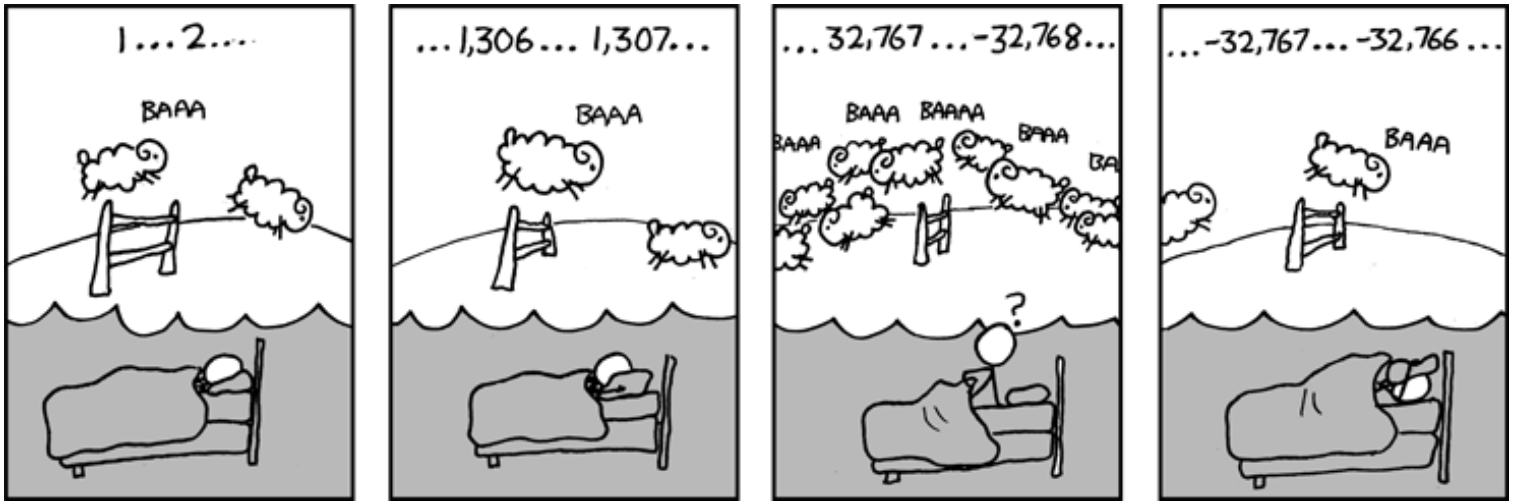
How to avoid them?

- **Check for the wraparound, either before performing the operation which would make it occur or afterwards**
- The limits from <limits.h> are helpful, but be aware since **naive use of them does not work**:

```
unsigned i, j, sum;
if (sum + i > UINT_MAX)
    // Too big error
else
    sum += i;
```

Integer vulnerabilities

Overflow – signed data types



- **Occurs when a signed integer operation results in a value that cannot be represented in the resulting type**
- **Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, or both**
- Since signed integer overflow produces a silent wraparound in most existing C compilers, **some programmers assume that this is a well-defined behavior**

Integer vulnerabilities

Finding – and fixing – overflows

Shift operations

- **Operand values are bounds-checked**
- If the check is passed, then the shift is performed

Arithmetic operations

- **Problem: n-bit add/sub operation needs $n + 1$ bits of precision**
- Similarly, n-bit multiplication needs $2n$ bits of precision
  - In C, addition, subtraction, multiplication, negation, and division may all result in overflow or underflow
  - When -2^{n-1} is negated or divided by -1, the result overflows and wraps back to -2^{n-1} itself!
- **Three different methods** can be applied

Universidad Zaragoza
1542

Integer vulnerabilities

Finding – and fixing – overflows

Detecting overflow for an operation on two signed integers s_1 and s_2

- **Precondition test**:

$$((s_1 > 0) \wedge (s_2 > 0) \wedge (s_1 > (\texttt{INT\_MAX} - s_2))) \vee$$
$$((s_1 < 0) \wedge (s_2 < 0) \wedge (s_1 < (\texttt{INT\_MAX} - s_2)))$$

- **CPU flag post-condition test**
- **Width extension post-condition test**
  - Convert s_1 and s_2 into a wider type
  - Perform the operation
  - Check whether the result is in bounds w.r.t. the original (narrower) type

Universidad Zaragoza

Integer vulnerabilities

Truncation

- **Occurs as the result of assignment or casting from a type with greater width to a type with lesser width**
- **Data may be lost if the value cannot be represented in the resulting type**
  - For instance, adding `c1` and `c2` in the following program fragment produces a value outside the range of `unsigned char`, considering an implementation where `unsigned char` is represented using 8 bits ($2^8 - 1$ or 255).

```
unsigned char sum, c1, c2;

c1 = 200;
c2 = 90;
sum = c1 + c2;
```

Integer vulnerabilities

Signedness errors

Rules of conversions

- **Set of rules that provides a mechanism to yield a common type** when
  - Both operands of a binary operator are balanced to a common type
  - The second and third arguments of the conditional operator (`? :`) are balanced to a common type
- Balancing conversions involve two operands of different types
  - One or both operands may be converted
- **Many operators that accept integer operands perform conversions using the usual arithmetic conversions**, including `*`, `/`, `%`, `+`, `-`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&`, `^`, `|`, and the condition operator `? :`

Integer vulnerabilities

Signedness errors

Example of conversion on x86-32

```
unsigned int ui = UINT_MAX;
signed char c = -1;

if (c == ui) {
    printf("-1 = 4,294,967,295?\n");
}
```

Integer vulnerabilities

Signedness errors

Some notes

- **Implicit conversions simplify C language programming**
- **Be aware: conversions have the potential for lost or misinterpreted data**
- Avoid conversions that result in:
  - **Loss of value** (conversion to a type where the magnitude of the value cannot be represented)
  - **Loss of sign** (conversion from a signed type to an unsigned type resulting in loss of sign)
- **Integer conversions to a type with greater rank and the same signedness are guaranteed safe for all data values on all conforming implementations**

Universidad Zaragoza

Exploiting Software Vulnerabilities

Software Vulnerabilities

Buffer and Integer Overflows

Universidad Zaragoza

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

Course 2020/2021

Master's Degree in Informatics Engineering

University of Zaragoza

Google Meet