Lecture 2.1
Buffer Overflows on the Stack

MICS - 2019

Dr. Alexandre Bartel
www.abartel.net

Challenge

Challenge

- www.dosbox.com
- https://github.com/fabiensanglard/vanilla_duke3D
- http://www.openwatcom.org/
- Make sure you have this in c:/autoexec.bat

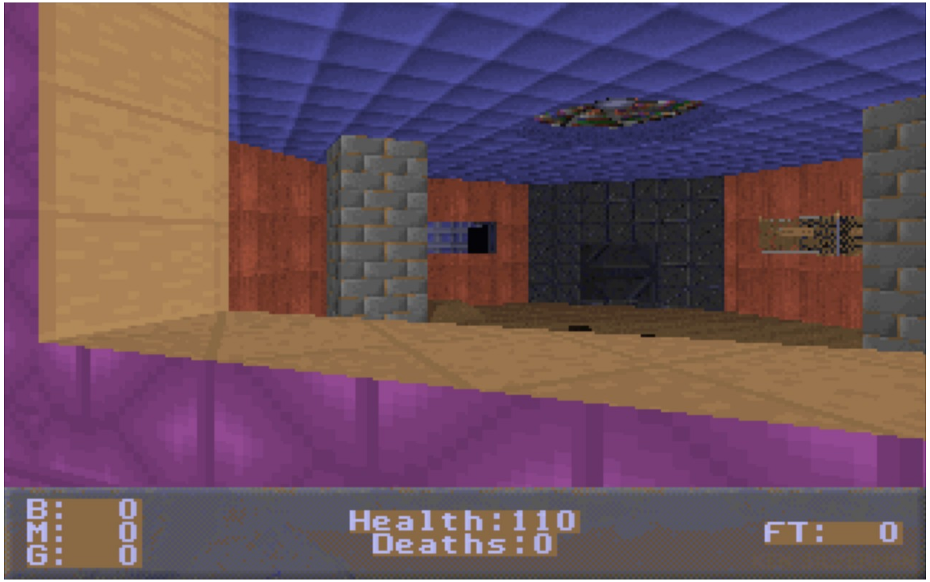
PATH C:\WATCOM\BINW;%PATH%

SET INCLUDE=C:\WATCOM\H

SET WATCOM=C:\WATCOM

SET EDPATH=C:\WATCOM\EDDAT

SET WIPFC=C:\WATCOM\WIPFC



Previously...

Lecture 1

- Software Development
- Software Security
- Software Vulnerability

Lecture 1: Software Development

- Tukey, 1958
- Functions, compilers, documentation (vs. hardware)
- Life-cycle
  - idea, requirements, design, implementation, deployment
- Approaches: Waterfall, Agile
- Goals: less risk, better quality

Lecture 1: Software Security

- Security policy
- Software system tries to maintain the following attributes in accordance with the security policy:
  - C…
  - I…
  - A…
- How? With security mechanisms
  - Access control
  - Sandbox

Lecture 1: Software Vulnerability

- Life cycle:
  - Birth, discovery, disclosure, correction, publicity, scripting, death
- Non-disclosure, full disclosure, responsible disclosure
- CVE number, MITRE

"Introduction to Software Security" Course Plan

2. Memory Attacks and Defenses
   - ➔ Buffer overflow
   - ➔ Heap overflow
   - ➔ Integer overflow
   - ➔ String format vulnerabilities
   - ➔ Type confusion
   - ➔ Use After Free

Prerequisites

1. Course “Introduction to Programming”
   - Array
   - Function
2. Course “Computer Systems”
   - Program execution
   - Stack
   - Heap

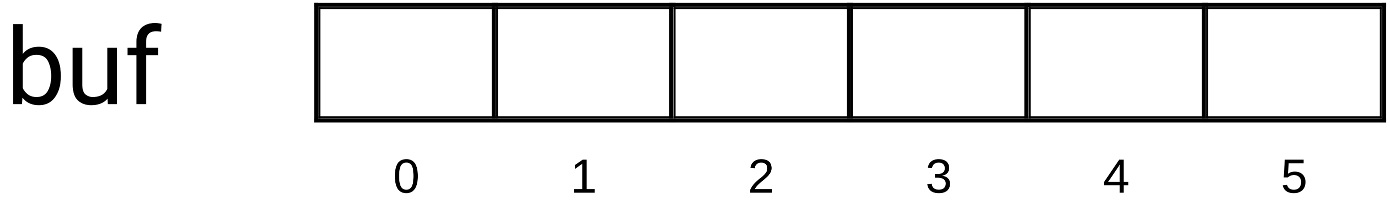
Buffer Overflow Attacks On the Stack

Analyst

- We could use the term “attacker”
- But “analyst” is more neutral
- Vulnerability could be done by security researchers in a lab

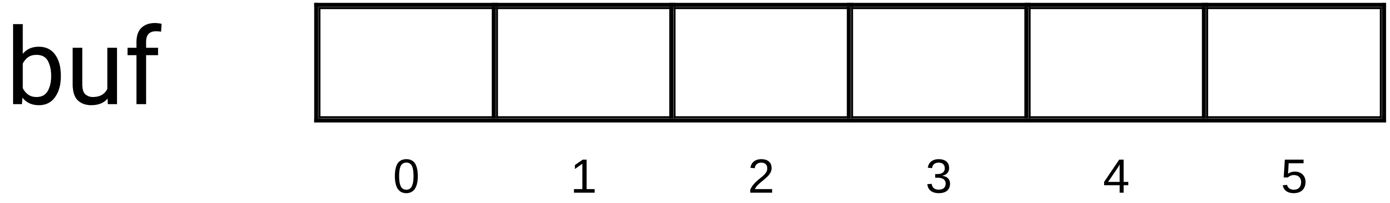
Buffer

- Container for data
- Bytes in memory
- Ex of a byte buffer, named **buf**, of size 6:



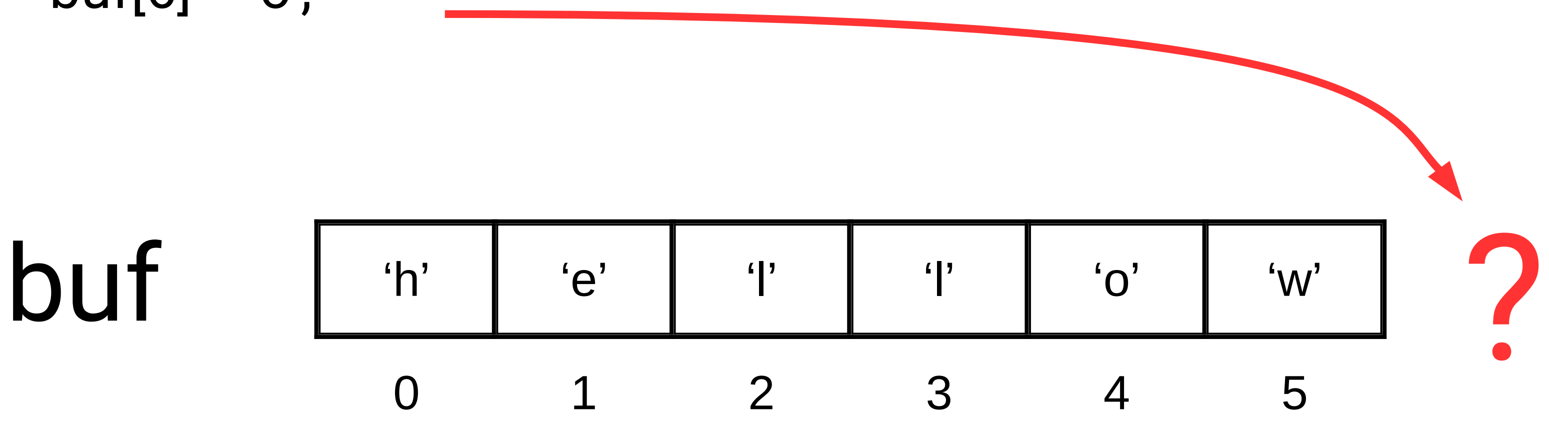
Buffer

- Normal use of **buf** when **0 <= index and index < size(buf)**
  - buf[0] = ‘h’; buf[1] = ‘e’; buf[2] = ‘l’;
  - buf[3] = ‘l’; buf[4] = ‘o’; buf[5] = ‘w’;



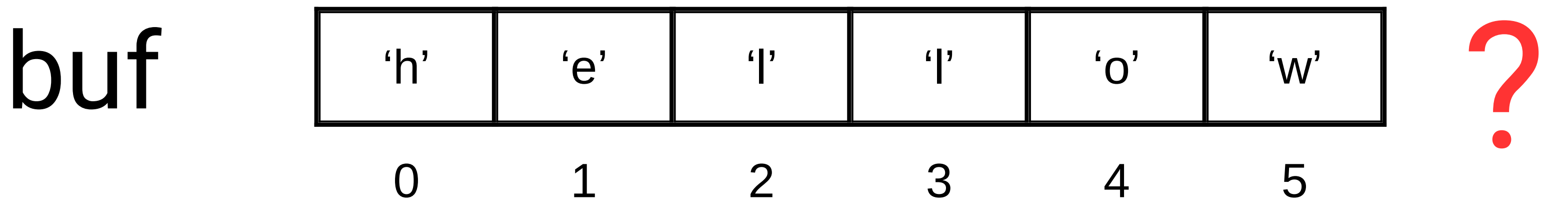
Buffer Overflow

- In C : no validity check for the index
- Overflow of **buf** when **index >= size(buf)**
  - buf[6] = ‘o’;



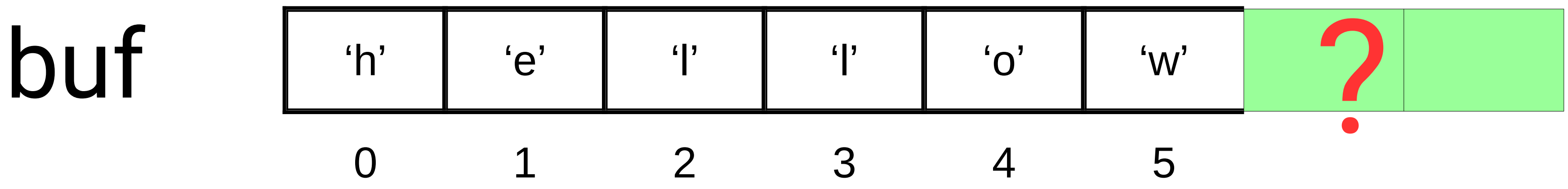
Buffer Overflow

- What happens depends on the context
  1. Nothing
  2. Segmentation fault
  3. Custom code execution



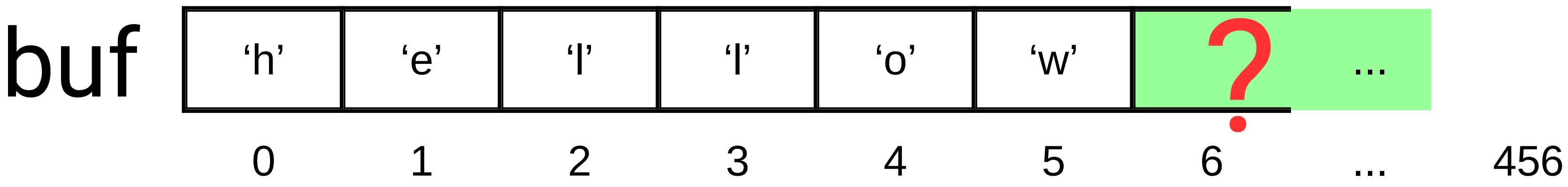
Buffer Overflow

1. Nothing
   - For performance reasons, the OS might allocate more bytes
   - Overflowing **buf** by a few bytes might not result in an error



Buffer Overflow

2. Segmentation Fault
   - If the index is large it will eventually reach a memory zone not allocated to the program
   - The OS detects it and stops the program
   - The overflow might also corrupt existing data



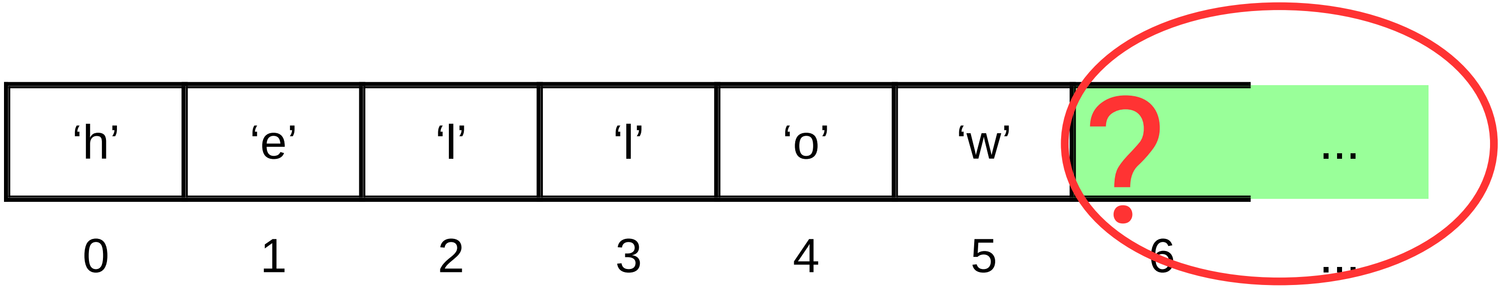
Buffer Overflow

What data could be next to the buffer?

3. Custom Code Execution

- The overflowing bytes have to redirect the execution flow to the analyst's code
- Need to understand what could be overwritten by the overflow

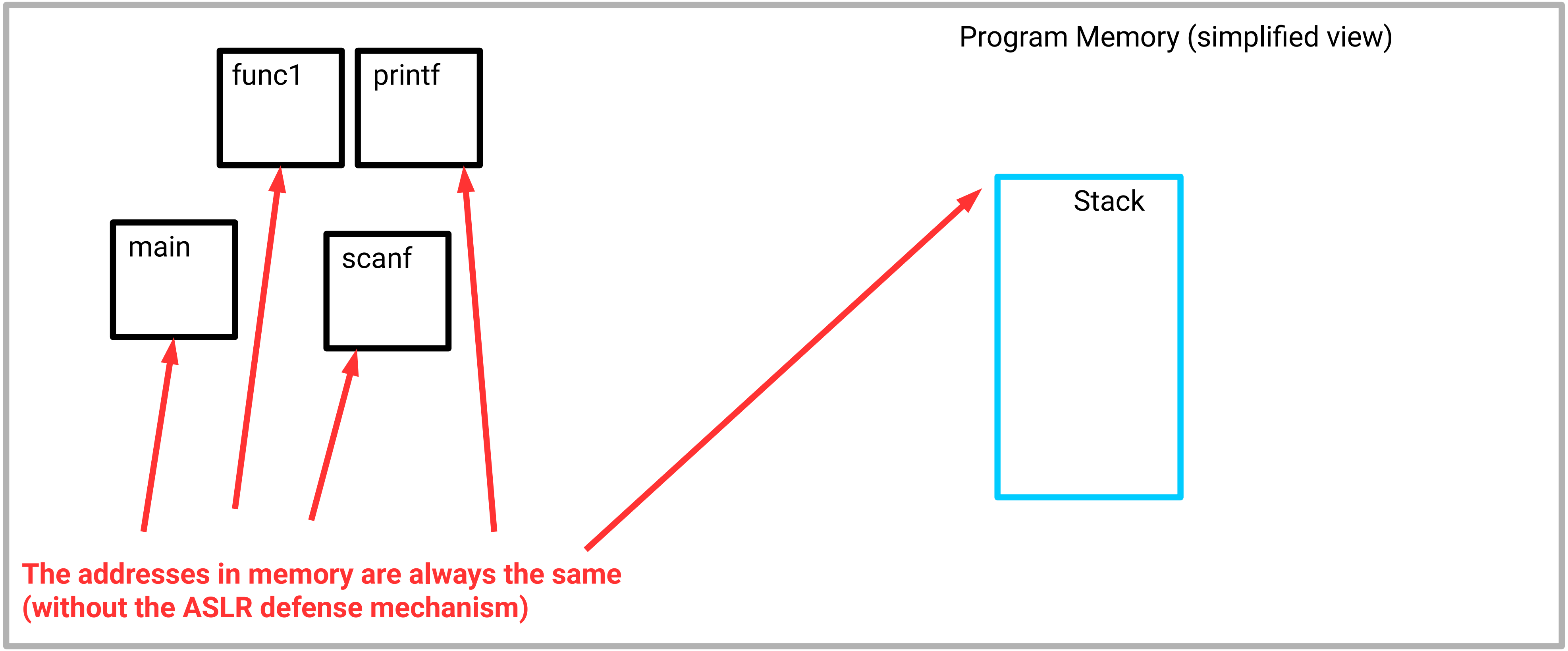
buf



Custom Code Execution

- A C program is made of <u>**functions**</u>
- Every instruction of a function is identified by a memory address (@)
- When a function F1 calls F2, the @ of the next instruction of F1 is stored on the <u>**stack**</u>
- Local variables (e.g. integers, floats, **buffers**) are also stored on the stack

Custom Code Execution



Custom Code Execution

	void main() {
@code + 0x00	func1();
@code + 0x01	return;
	}

Local variables are on the stack

	void func1() {
	byte[6] **buf**;
@code + 0x02	scanf("%s", &buf);
@code + 0x03	return;
	}

Memory @	Value
...	
@stack + 0x08	
@stack + 0x07	
@stack + 0x06	
@stack + 0x05	
@stack + 0x04	
@stack + 0x03	
@stack + 0x02	
@stack + 0x01	buf[0]
@stack + 0x00	@code + 0x01

stack grows this way

Custom Code Execution

If the analyst inputs the following string:

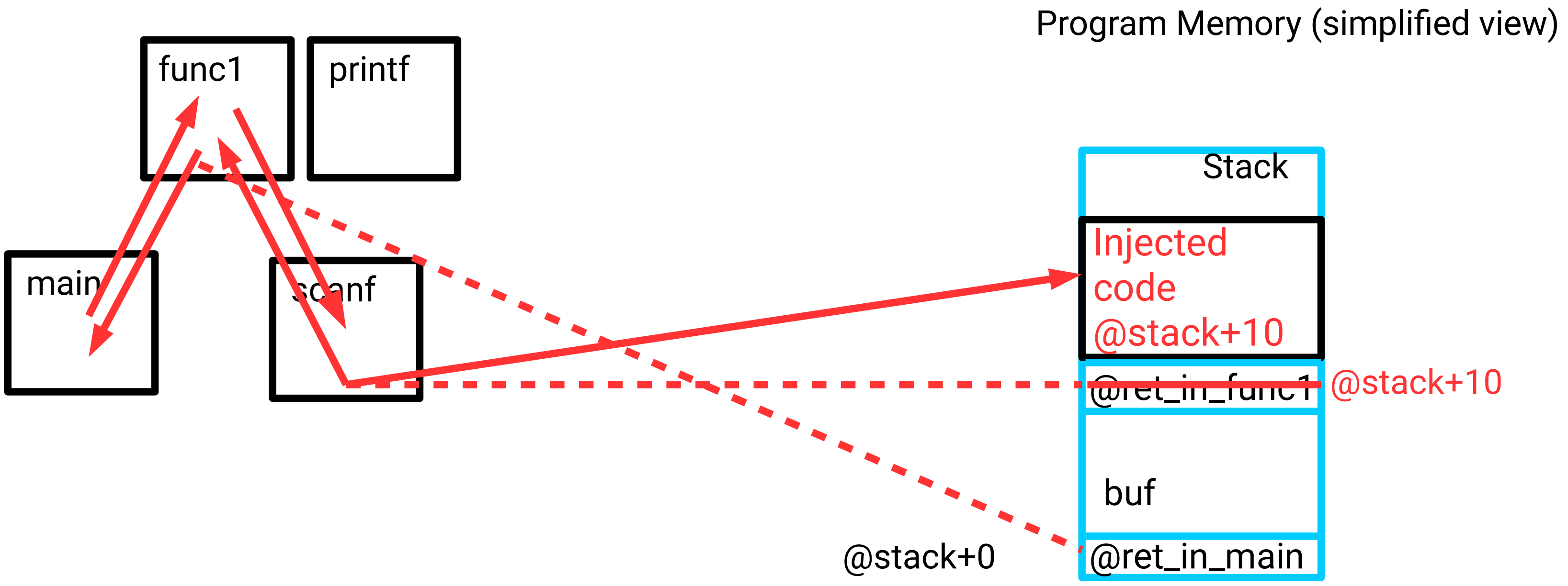
‘f’	‘o’	‘o’	‘b’	‘a’	‘r’	@HCF	i0	i1	i2	...

Memory @	Value
	...
...	
@stack + 0x08	
@stack + 0x07	
@stack + 0x06	‘r’
@stack + 0x05	‘a’
@stack + 0x04	‘b’
@stack + 0x03	‘o’
@stack + 0x02	‘o’
@stack + 0x01	‘f’
@stack + 0x00	@code + 0x01

stack grows this way

- The analyst changed the return @ of scanf!
- The goal is to set hijack the control flow to execute the analyst’s injected code (e.g., @HCF = @stack + 0x08).
- Remember: all function/stack @ are fixed!

Custom Code Execution



The control flow of the program can be changed by exploiting the buffer overflow vulnerability to change the value of @ret_in_func1!

Consequences

1. The analyst can execute custom code
   - Usually a reverse shell (connects back to the analyst’s computer)
2. The injected code runs with the privileges of the vulnerable process
   - The analyst can use one (or more) other vulnerability to gain more privilege

Defenses

- Two main approaches to limit exploitation
  - Stack canary (aka stack cookie)
  - Data Execution Prevention (DEP)
- Two main approaches to prevent attacks bypassing DEP:
  - Address Space Layout Randomization (ASLR)
  - Control Flow Integrity (CFI)

Buffer Overflow: A bit of History

- 1972: first public mention in “Computer Security Technology Planning Study” [1]
- 1988: Morris Worm exploited a buffer overflow in the *finger* daemon [2]
- 1996: Famous tutorial “Smash the Stack for Fun and Profit” published in Phrack 49 by Elias Levy (aka Aleph One) [3]

[1] Anderson, James P. Computer Security Technology Planning Study. Volume 2. Anderson (James P) and Co Fort Washington PA, 1972
[2] Spafford, Eugene H. "The internet worm incident." European Software Engineering Conference. Springer, Berlin, Heidelberg, 1989
[3] One, Aleph. "Smashing the stack for fun and profit (1996)." See http://www. phrack. org/show. Php

Buffer Overflow: A bit of History

- 2017: Exploiting buffer overflows in Intel ME [1]
  - Intel ME: proprietary autonomous subsystem in Intel's processor running even if the computer is asleep
  - Recent example on how to bypass stack cookie
- 2018: Buffer overflow in AMD’s [2]

[1] Ermolov, Mark, and Maxim Goryachy. "How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine." Black Hat Europe (2017).
[2] https://www.bleepingcomputer.com/news/security/security-flaw-in-amds-secure-chip-on-chip-processor-disclosed-online/ (6 January 2018)

Next Lecture: Heap-Based Buffer Overflow

- We have seen how to exploit a buffer overflow when the buffer is on the stack
- The buffer could also be on the **heap**
  - buf = malloc(6)
- How to exploit the vulnerability depends on the target program
- Redirect the control flow to the analyst's code by, e.g., overwriting a function pointer

Take Home

- Buffer overflow comes with certain programming languages such as C and C++
- The problem has been know for more than 45 years
- Still exist (and exploitable) in 2018 despite mitigation techniques
- Mitigation techniques makes it harder to exploit