Lecture #4: Stack Overflow

UCalgary ENSF619
Elements of Software Security

Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)
Partly based on slides by Drew Davidson, University of Kansas and Robert Walls, WPI

Let’s talk about stack smashing

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine File 14 of 16

BugTraq, r00t, and Underground.Org

bring you

Smashing The Stack For Fun And Profit

Aleph One

aleph1@underground.org

> `smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

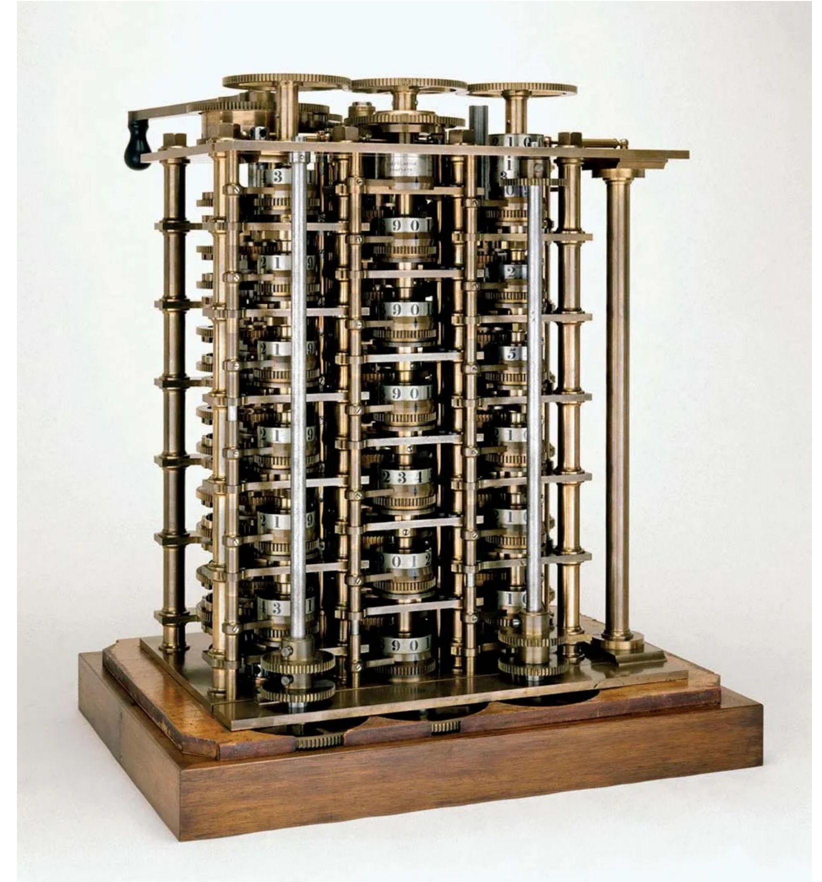
Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt library, at, etc. This paper attempts to explain what buffer overflows are, and how their exploits work. Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with gdb are very helpful but not necessary. We also assume we are working with an Intel x86 CPU, and that the operating system is Linux. Some basic definitions before we begin: A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load time on the data segment. Dynamic variables are allocated at run time on the stack. To overflow is to flow, or fill over the top, brims, or bounds. We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

Data and Code

Consider the History of Computation

The earliest devices recognized as computers were built to perform some specific type of computation



Data and Code

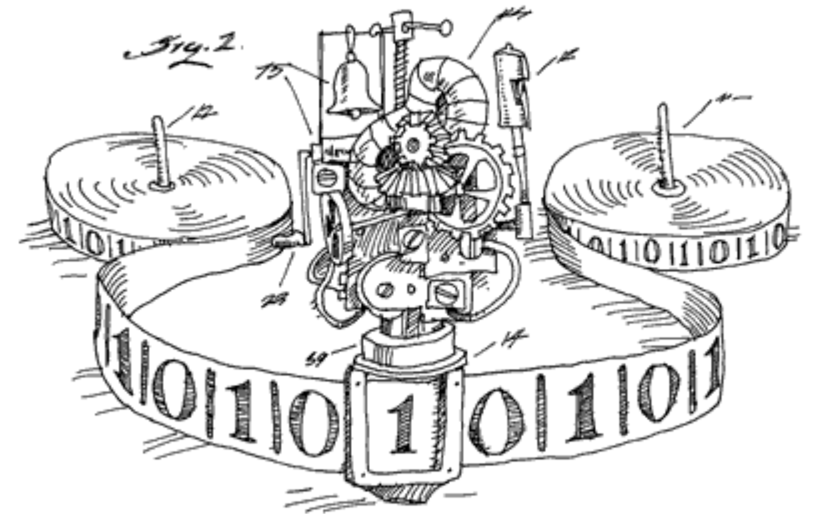
Consider the History of Computation

The earliest devices recognized as computers were built to perform some specific type of computation

Algorithmic Purpose specified by hardware

Consider the theory analogy: a Turing Machine to compute a Fibonacci Sequence

- Fibonacci computation encoded into the state machine
- Input number encoded into the tape at start
- Output number encoded onto the tape at halt



Data and Code

A major paradigm shift: The Universal Computation Machine

The hardware contains generally-useful instructions

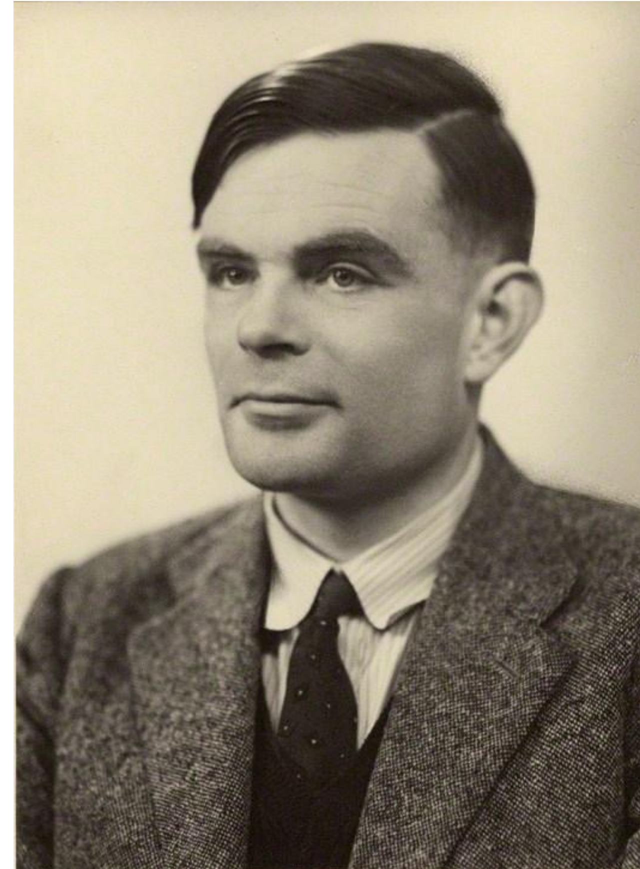
A particular algorithm is encoded in terms of those instructions

The Theory: The Universal Turing Machine

Consider the theory analogy: a Turing Machine that computes

<u>any</u> function

- “Instruction set” encoded into the state machine
- Desired algorithm encoded into the tape at start
- Input to the algorithm encoded into the tape at start as well
- Output number encoded onto the tape at halt



Data and Code

A MAJOR PARADIGM SHIFT: THE UNIVERSAL COMPUTATION MACHINE

The hardware contains generally-useful instructions

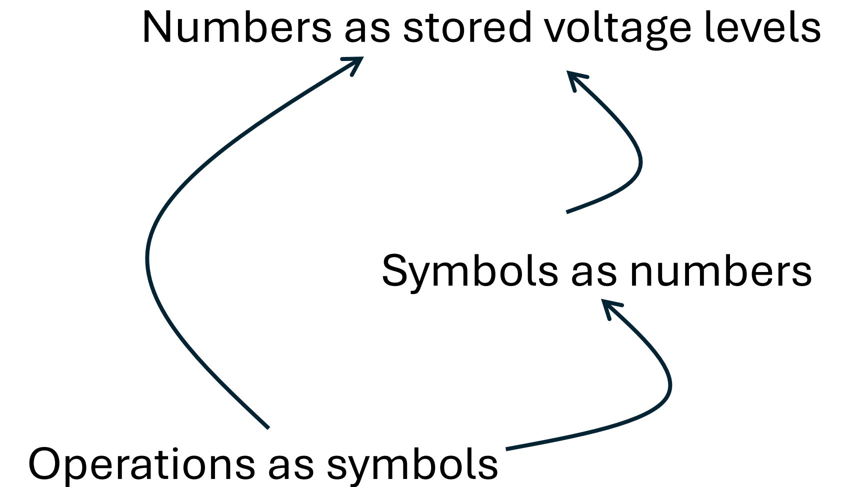
A particular algorithm is encoded in terms of those instructions

THE THEORY: THE UNIVERSAL TURING MACHINE

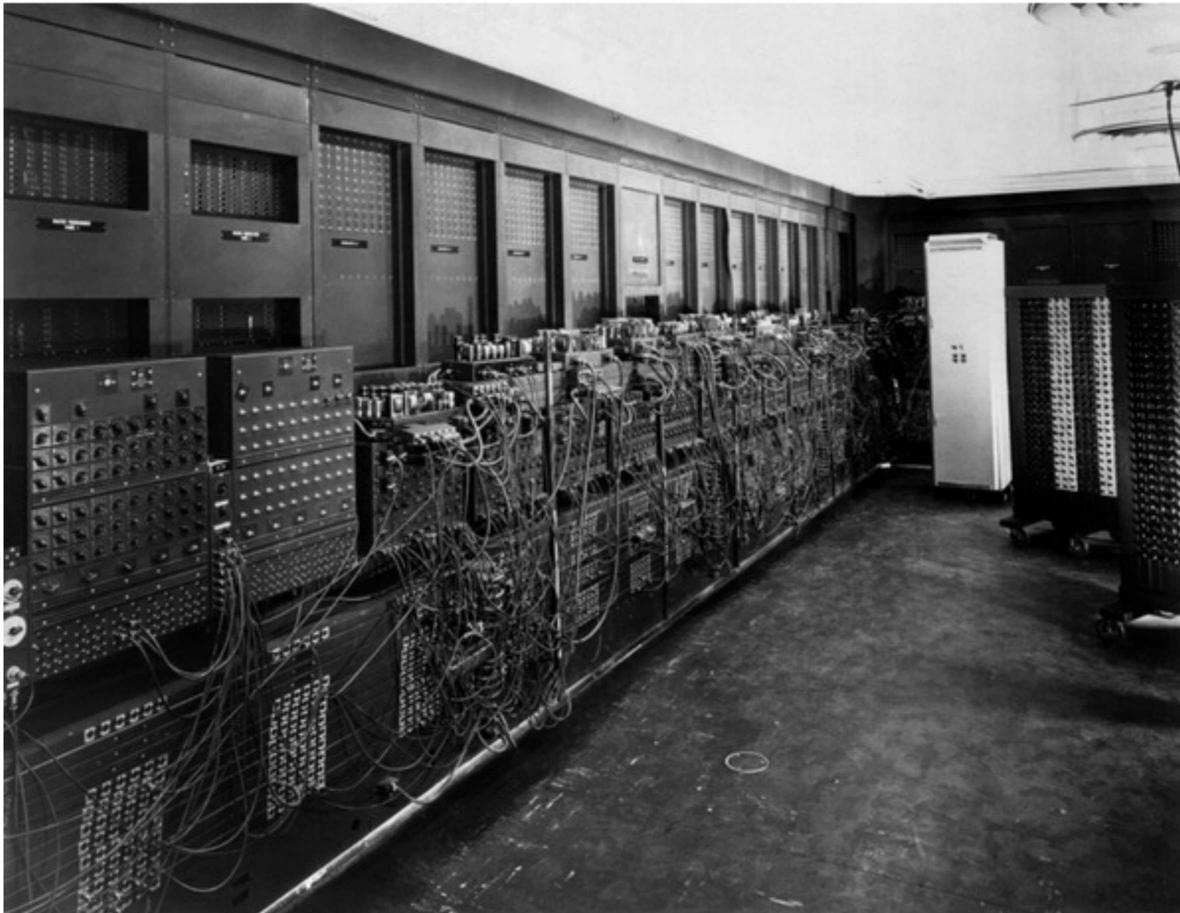
Consider the theory analogy: a Turing Machine that computes

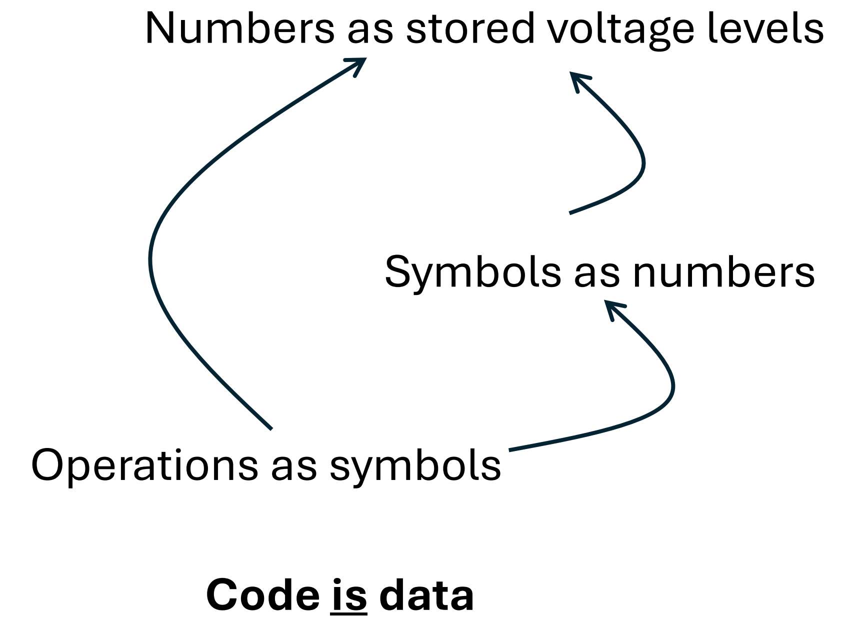
any function

- “Instruction set” encoded into the state machine
- Desired algorithm encoded into the tape at start
- Input to the algorithm encoded into the tape at start as well
- Output number encoded onto the tape at halt



Data and Code



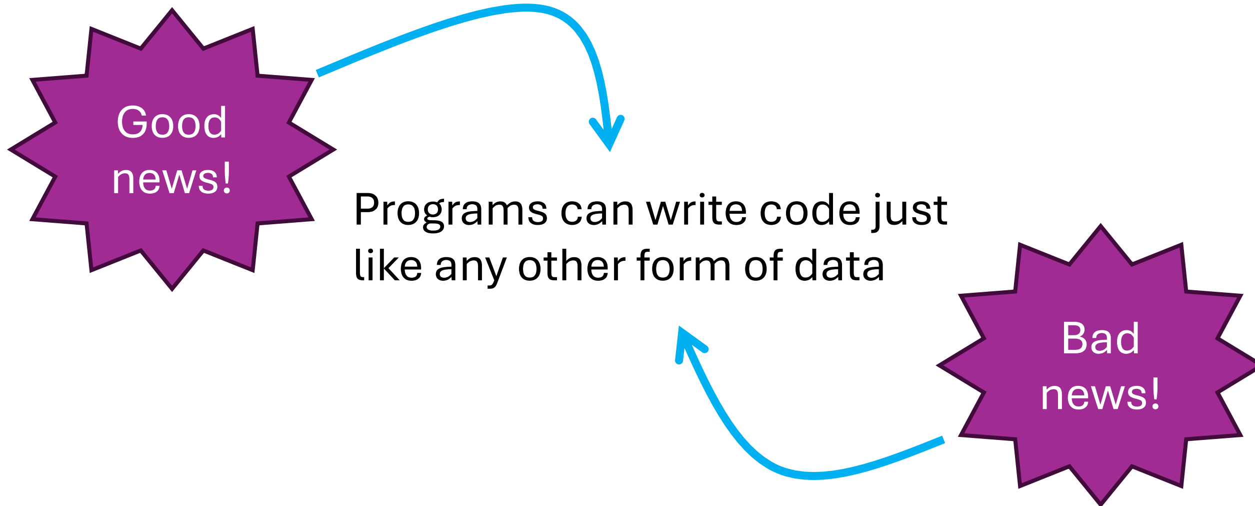


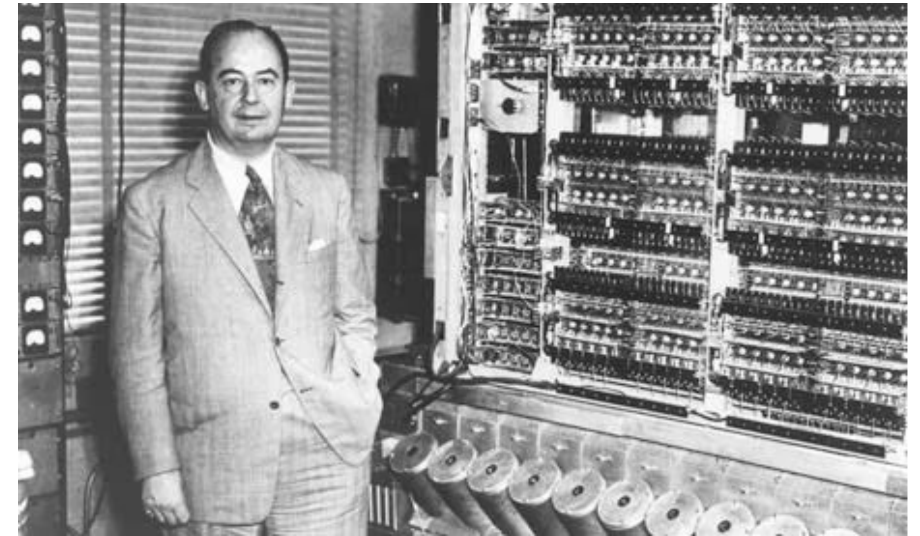
Code <u>is</u> data

Data and Code

The Von Neumann architecture

Another big idea: Code and data share memory





Code <u>is</u> data

Why is this “bad news?

- Well, fundamentally data and code share the same memory
- Normally, programs will be architected in such a way that:
  - Code has its own region (text segment)
  - Data has its own region(s) (stack, heap, globals)
- However, this is an **artificial limitations**!
- There is no fundamental region why code could not exist among data
- Code itself is nothing special… just another sequence of bytes

How do computers execute?

- We have seen how the stack works in the last lecture
- The only missing piece is code execution itself
- It uses a little register called the **program counter**

Program Memory

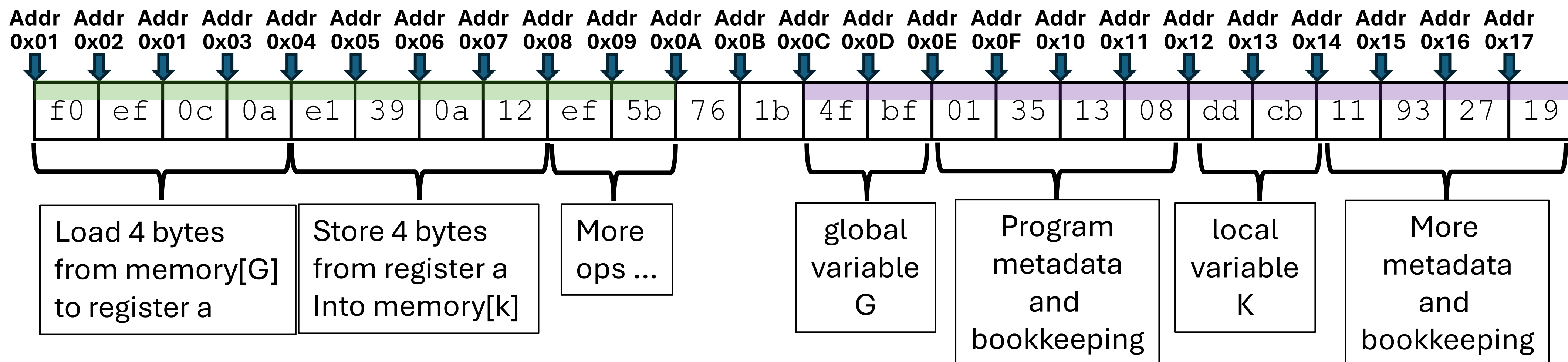
A 1-D ARRAY

Code

Load 2 bytes memory[G] into register a
Store register a into 4 bytes memory[K]
More ops …

Data

Global variable G
Foo’s local variable K



Simulating Source Constructs

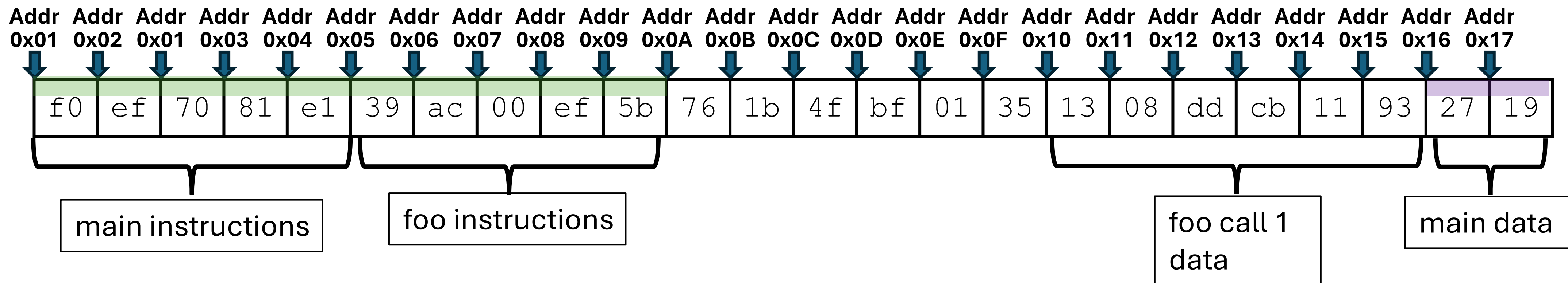
Assume main calls foo
foo (recursively) calls foo

Code

All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K



Simulating Source Constructs

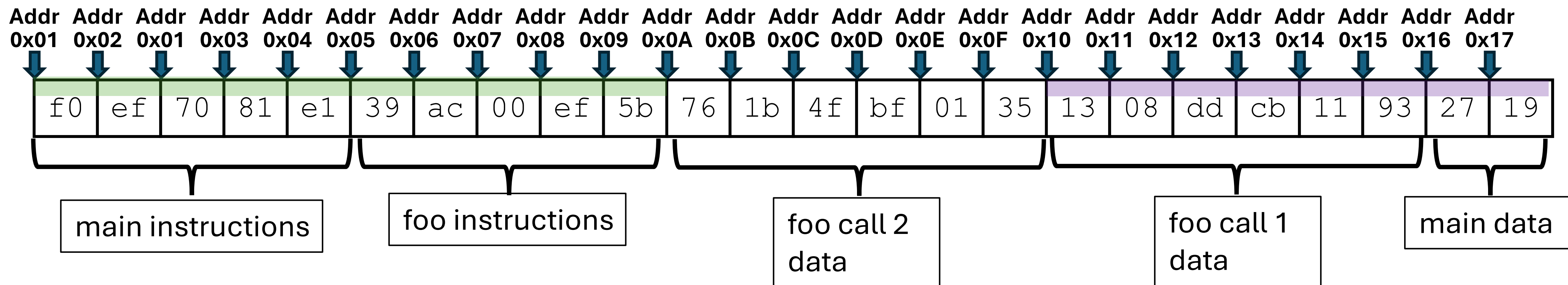
Assume main calls foo
foo (recursively) calls foo

Code

All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K



Simulating Source Constructs

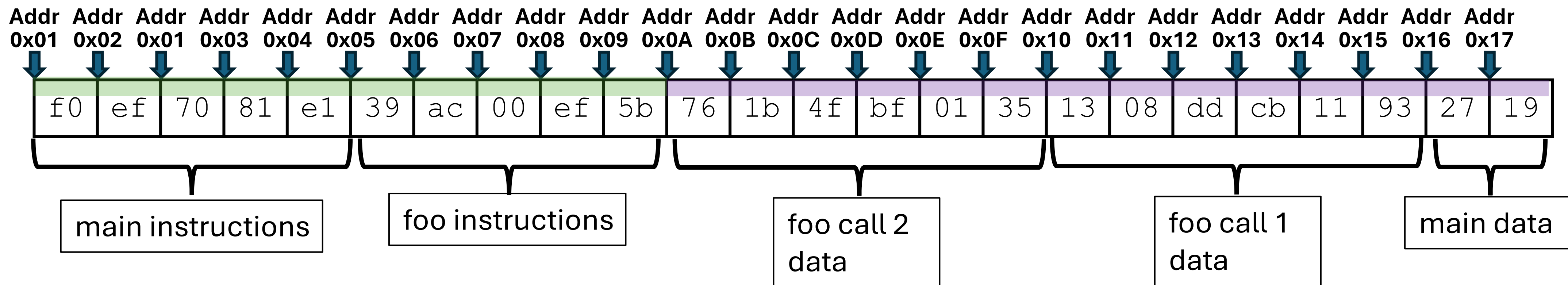
Assume main calls foo
foo (recursively) calls foo

Code

All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K



Simulating Source Constructs

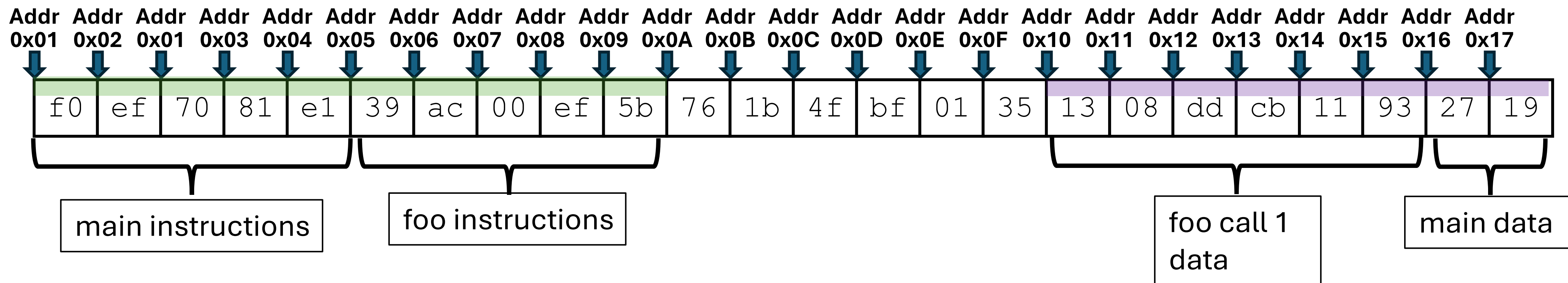
Assume main calls foo
foo (recursively) calls foo

Code

All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K



Simulating Source Constructs

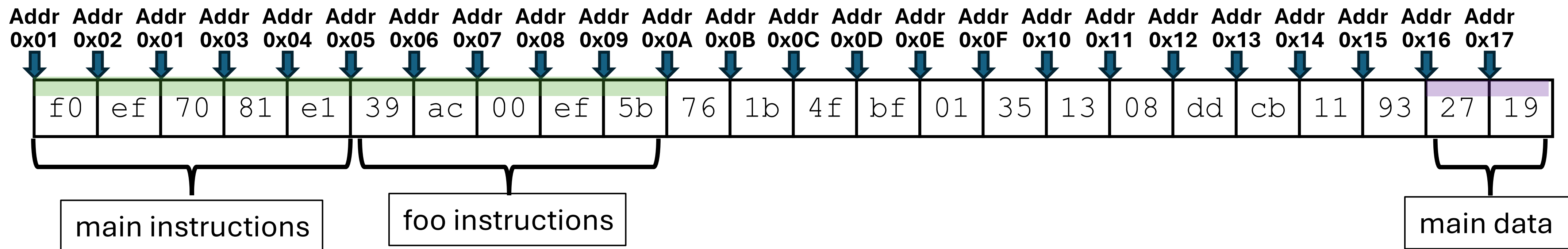
Assume main calls foo
foo (recursively) calls foo

Code

All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K

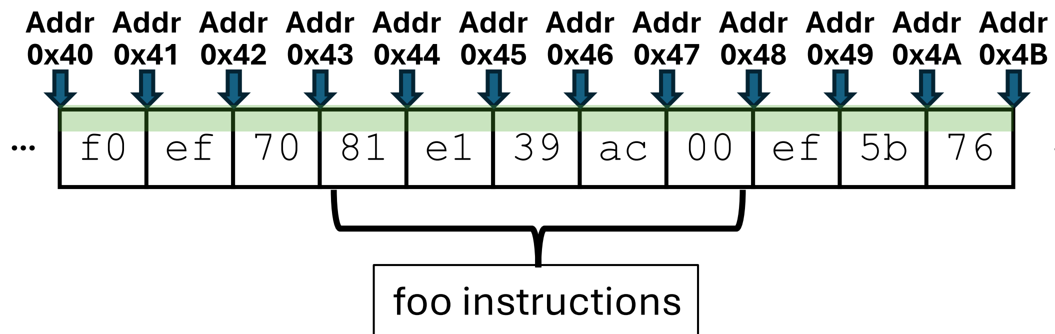


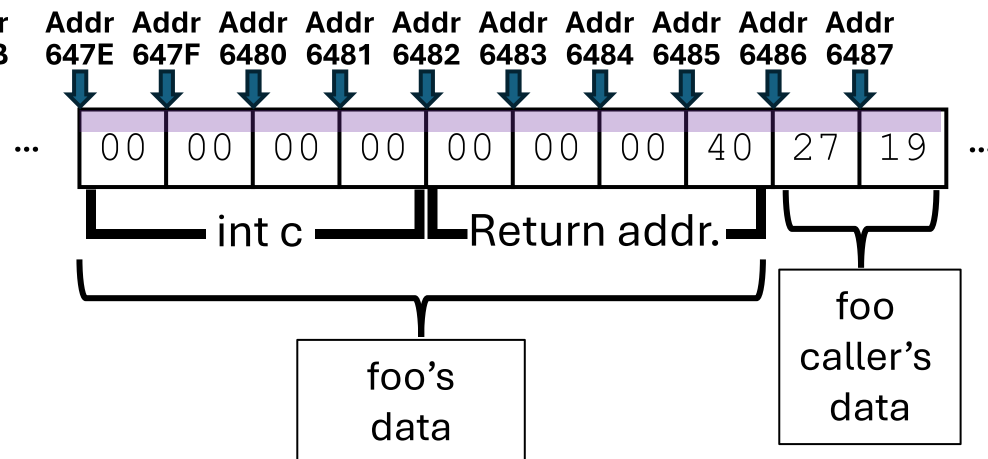
Some more key points

- The **instruction or program counter** (IC/PC) is a hardware register that indicates the next instruction to execute—the PC is also called the instruction pointer.
- Arguments are passed to functions via registers in 64-bit x86 as defined by the **calling convention**.
- The PC will be incremented sequentially until the program has a conditional instruction, or a **function call/return**
- The program uses a stack-like data structure to track calls and returns

Breaking Bounds

```
foo(){
    int c = getc(stdout);
    *((int *)c) = 2;
}
```



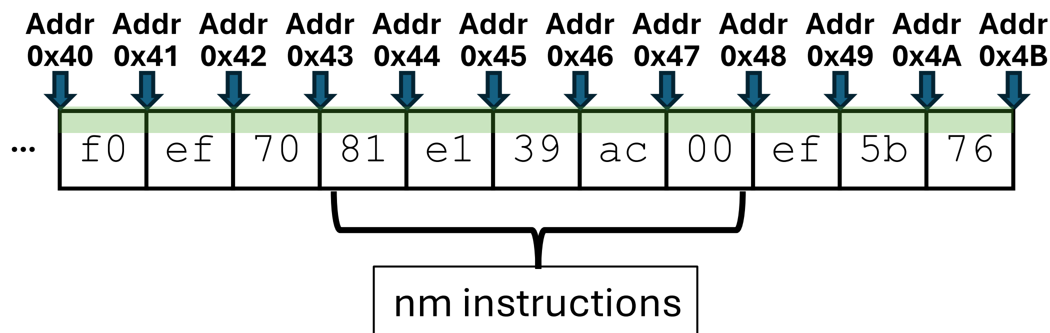


Buffer Overflows

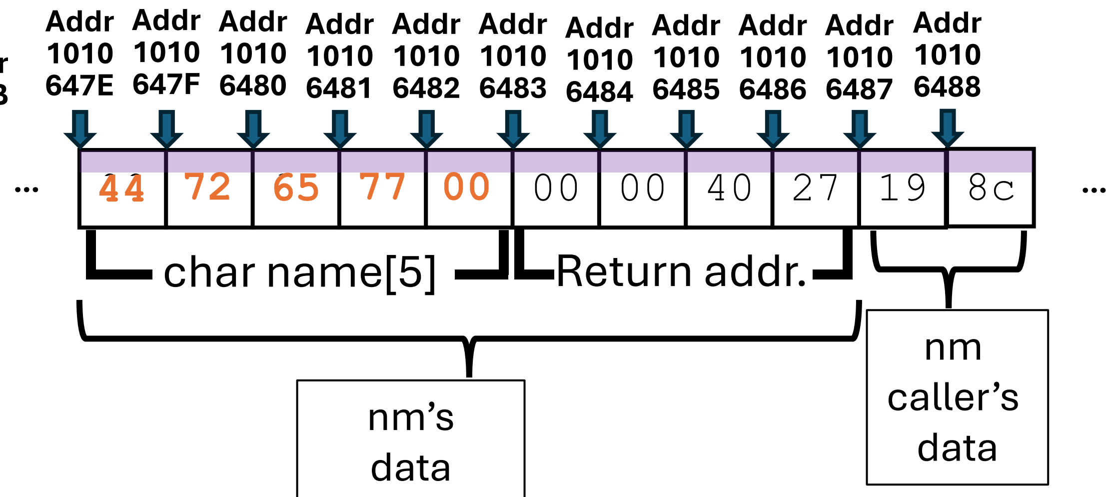
```
int nm()
{
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf("hi %s\n", name);
    return 0;
}
```

D	r	e	w	(end)
0x44	0x72	0x65	0x77	0x00

Program instructions



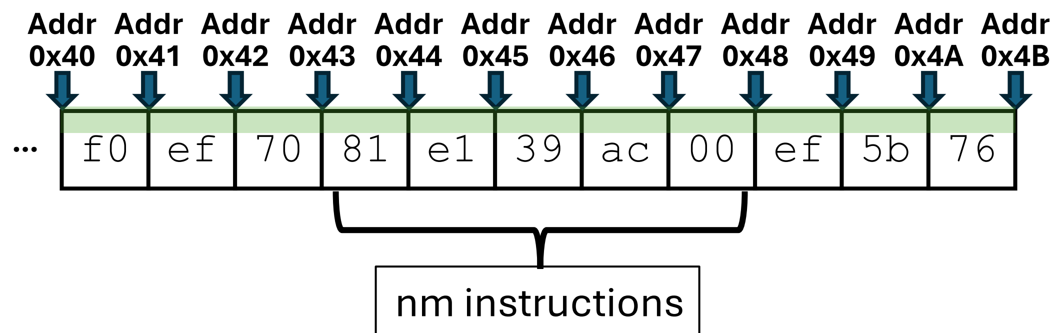
Program (meta)data

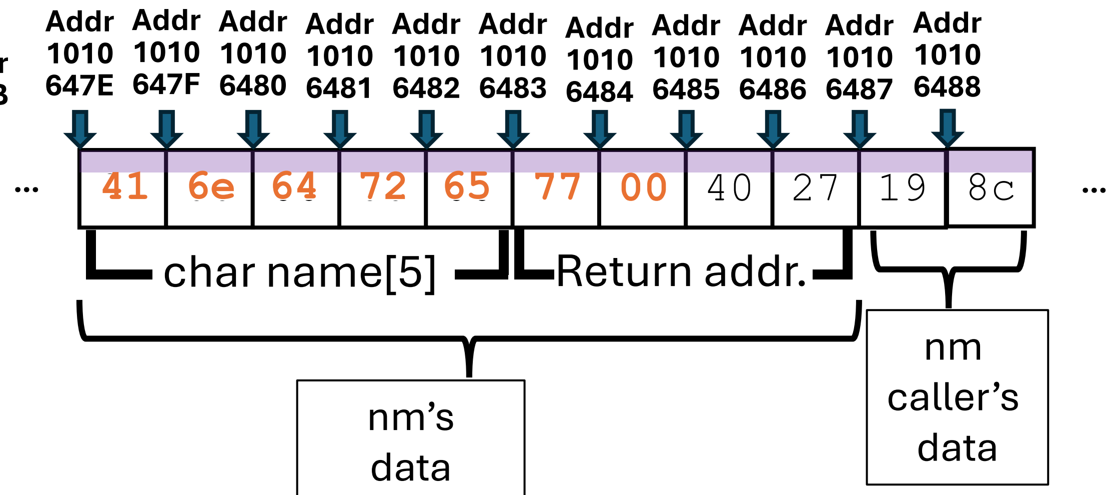


Buffer Overflows

```
int nm()
{
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf(“hi %s\n", name);
    return 0;
}
```

A	n	d	r	e	w	(end)
0x41	0x6e	0x64	0x72	0x65	0x77	0x00

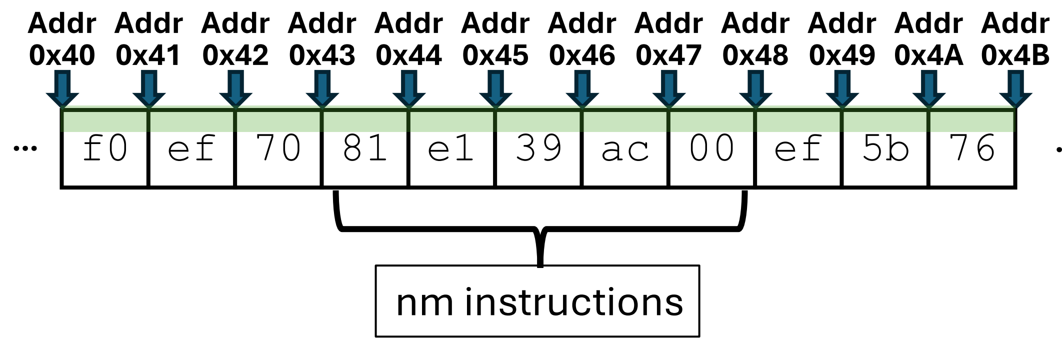


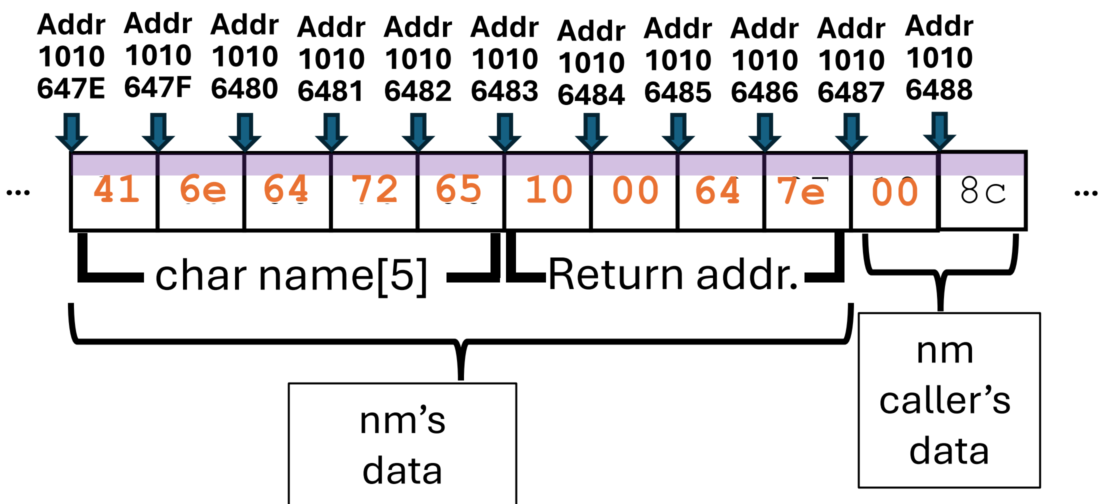


Script Injection

```
int nm()
{
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf(“hi %s\n", name);
    return 0;
}
```

A	n	d	r	e	LF	LF	d	~	(end)
0x41	0x6e	0x64	0x72	0x65	0x10	0x10	0x64	0x7e	0x00





How big of a deal is this exactly?

- … a pretty big one!
- Let’s have a chat about what we think of this paper

So… is this a big deal?

- <<This paper forever changed the cybersecurity landscape by shedding light on stack-based buffer overflows>> (Raymond A. Hagen)
- <<Aleph One’s excellent Smashing the Stack for Fun and Profit article from 1996 has long been the go-to for anyone looking to learn how buffer overflow attacks work>> (Jon Gjengset)
- << the article has become a milestone in memory vulnerability research and exploit development.>> (ARM developer hub)

Memory bugs in the mid-90s

- Lots of code written in C (interpreted languages existed, but mostly used for scripting due to overhead)
- Not much awareness about secure code development
    - Bugs, use of insecure functions such as strcpy
- No protection against buffer overflow attacks
- Many software applications sitting ducks for this type of exploits

Was it bad that this article was published?

- No! It **called attention** to the issue. People already knew about these attacks, but it was not at the forefront of the community
- By bringing this type of memory exploits in the spotlight, it spurred a **debate on their significance** and **how to contain them**
- Speaking of…

Defending against stack overflow attacks

Stack Canaries

Stack canaries

- **Stack canaries** are a type of software defense that attempts to mitigate buffer overflow attacks.
- **Idea:** Place a known value on the stack in between the local variables and the return address. If an attacker attempts to overflow the local buffer, then that overflow will also modify the canary.

2. docker

```
+pwndbg> disass *foo
Dump of assembler code for function foo:
   0x0000000000400546 <+0>:     push   rbp
   0x0000000000400547 <+1>:     mov    rbp,rsp
   0x000000000040054a <+4>:     sub    rsp,0xa0
   0x0000000000400551 <+11>:    mov    DWORD PTR [rbp-0x94],edi
   0x0000000000400557 <+17>:    mov    DWORD PTR [rbp-0x98],esi
   0x000000000040055d <+23>:    mov    DWORD PTR [rbp-0x9c],edx
   0x0000000000400563 <+29>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040056c <+38>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400570 <+42>:    xor    eax,eax
   0x0000000000400572 <+44>:    nop
   0x0000000000400573 <+45>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400577 <+49>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000400580 <+58>:    je     0x400587 <foo+65>
   0x0000000000400582 <+60>:    call   0x400420 <__stack_chk_fail@plt>
   0x0000000000400587 <+65>:    leave
   0x0000000000400588 <+66>:    ret
End of assembler dump.
+pwndbg>
```

<+4> This allocation also makes room for the canary

<+29-42> Retrieve the known canary value and place a copy on the stack.

<+45-60> Compare the stack's canary to the known value, call a special function if they don't match.

- Note: the return address won't be used if the canary check fails.

```
 RIP  0x40056c (foo+38) ◂— 0x4890c031f8458948
[──────────────────────────────────DISASM──────────────────────────────────]
 ► 0x40056c <foo+38>    mov    qword ptr [rbp - 8], rax
   0x400570 <foo+42>    xor    eax, eax
   0x400572 <foo+44>    nop
   0x400573 <foo+45>    mov    rax, qword ptr [rbp - 8]
   0x400577 <foo+49>    xor    rax, qword ptr fs:[0x28]
   0x400580 <foo+58>    je     foo+65                        <0x400587>
    ↓
   0x400587 <foo+65>    leave
   0x400588 <foo+66>    ret

   0x400589 <main>      push   rbp
   0x40058a <main+1>    mov    rbp, rsp
   0x40058d <main+4>    mov    edx, 3
[──────────────────────────────────STACK───────────────────────────────────]
00:0000│ rsp  0x7fffffffe1e0 ◂— 0x32f2f2f2f
01:0008│      0x7fffffffe1e8 ◂— 0x100000002
02:0010│      0x7fffffffe1f0 ◂— 0x0
... ↓
04:0020│      0x7fffffffe200 ◂— 0xff000000
05:0028│      0x7fffffffe208 ◂— 0x0
... ↓
[────────────────────────────────BACKTRACE─────────────────────────────────]
 ► f 0           40056c foo+38
   f 1           4005a1 main+24
   f 2     7ffff7a2d830 __libc_start_main+240
Breakpoint *foo+38
+pwndbg> p/x $rax
$3 = 0xdb4f8e8fd7a88300
+pwndbg>
```

> The canary itself is a random value.

Stack canaries

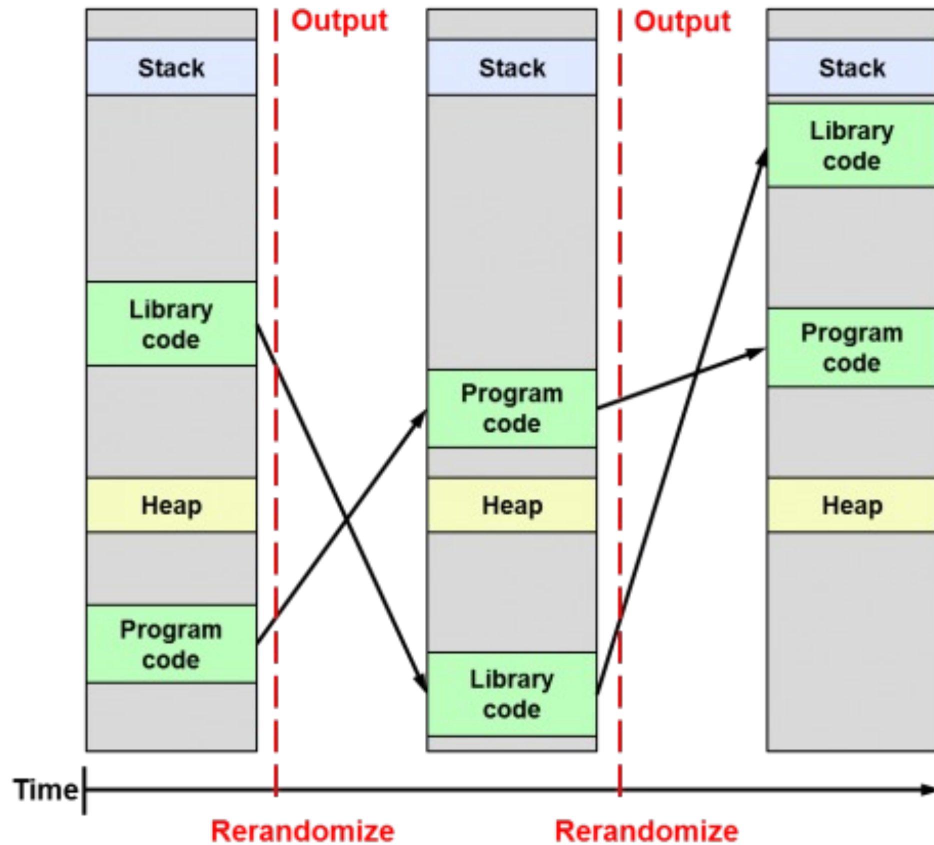
- **Stack canaries** are automatically added by modern compilers for functions with buffers that *might* overflow.
  - Is this example, the `buf` buffer is never used so it can't be overflowed, but the compiler doesn't realize that.
- As stack canaries can be added automatically by the compiler and they have low-overhead they are an attractive defense.
- However, their protections are limited; there are many known ways to bypass them.

ASLR

What is ASLR about?

- Observation: stack smashing relies on having an **approximate idea** of **where stuff is located** in memory
- The idea is to make these locations **unpredictable**
- The program is still going to run correctly, but…
- … the **location** of libraries, stack, heap, etc., is going to be **randomized**
- (not that useful for our simple example, but actual exploits rely on that knowledge to work)

How does ASLR look?



https://www.daniloaz.com/en/differences-between-aslr-kaslr-and-karl/

In different program runs, the layout of the virtual address space will look different

Works (?) for the kernel too

- **KASLR** is designed to prevent similar attacks in **kernel space**
- **... why is this a problem?**
- There is still some **debate** on its **usefulness**
  - Kernel has more **constraints** than user-space programs
  - Certain addresses are **hardware-specific** and cannot be **trivially randomized**

If interested

SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon

Hyerean Jang
Korea University
Seoul, Republic of Korea
hr_jang@korea.ac.kr

Taehun Kim
Korea University
Seoul, Republic of Korea
taehunk@korea.ac.kr

Youngjoo Shin
Korea University
Seoul, Republic of Korea
syoungjoo@korea.ac.kr

Abstract

Apple silicon is the proprietary ARM-based processor that powers the mainstream of Apple devices. The move to this proprietary architecture presents unique challenges in addressing security issues, requiring huge research efforts into the security of Apple silicon-based systems. In this paper, we study the security of KASLR, the randomization-based kernel hardening technique, on the state-of-the-art macOS system equipped with Apple silicon processors. Because KASLR has been subject to many microarchitectural side-channel attacks, the latest operating systems, including macOS, use kernel isolation, which separates the kernel page table from the userspace table. Kernel isolation in macOS provides a barrier to KASLR break attacks. To overcome this, we exploit speculative execution in system calls. By using Spectre-type gadgets in system calls, an unprivileged attacker can cause translations of the attacker's chosen kernel addresses, causing the TLB to change according to the validity of the address. This allows the construction of an attack primitive that breaks KASLR bypassing kernel isolation. Since the TLB is used as a side-channel source, we reverse-engineer the hidden internals of the TLB on various M-series processors using a hardware performance monitoring unit. Based on our attack primitive, we implement SysBumps, the first KASLR break attack on macOS for Apple silicon. Throughout evaluation, we show that SysBumps can effectively break KASLR across different M-series processors and macOS versions. We also discuss possible mitigations against the proposed attack.

ACM Reference Format:
Hyerean Jang, Taehun Kim, and Youngjoo Shin. 2024. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690189

1 Introduction

Apple recently began a transition from Intel-based processors to Apple silicon, its custom-designed, proprietary ARM-based processors for its products. While the move to this ARM-based architecture increases the performance and efficiency, the inherent nature of the proprietary processor creates challenges in addressing security issues within the products. However, despite its importance, there are only a few studies on the security of Apple silicon products [32, 49, 61] compared to studies on other commodity processors [23, 25, 31, 34, 40], requiring huge research efforts into the security of Apple silicon-based systems.

In line with this, this paper studies the security of the KASLR[1] implementation on the latest Apple silicon-based macOS system. KASLR is a primary kernel hardening technique to mitigate memory corruption vulnerabilities in the kernel by randomizing the layout of the kernel address space [52]. Since its introduction, KASLR implementations have been subject to microarchitectural side-channel attacks [2, 10, 11, 23, 28, 35, 39, 40, 42, 63]. That is, using side-channel techniques on caching hardware such as TLB[2], unprivileged attack-

Presented at ACM CCS 2024