EXERCISE #12

REVIEW – DATAFLOW FRAMEWORKS

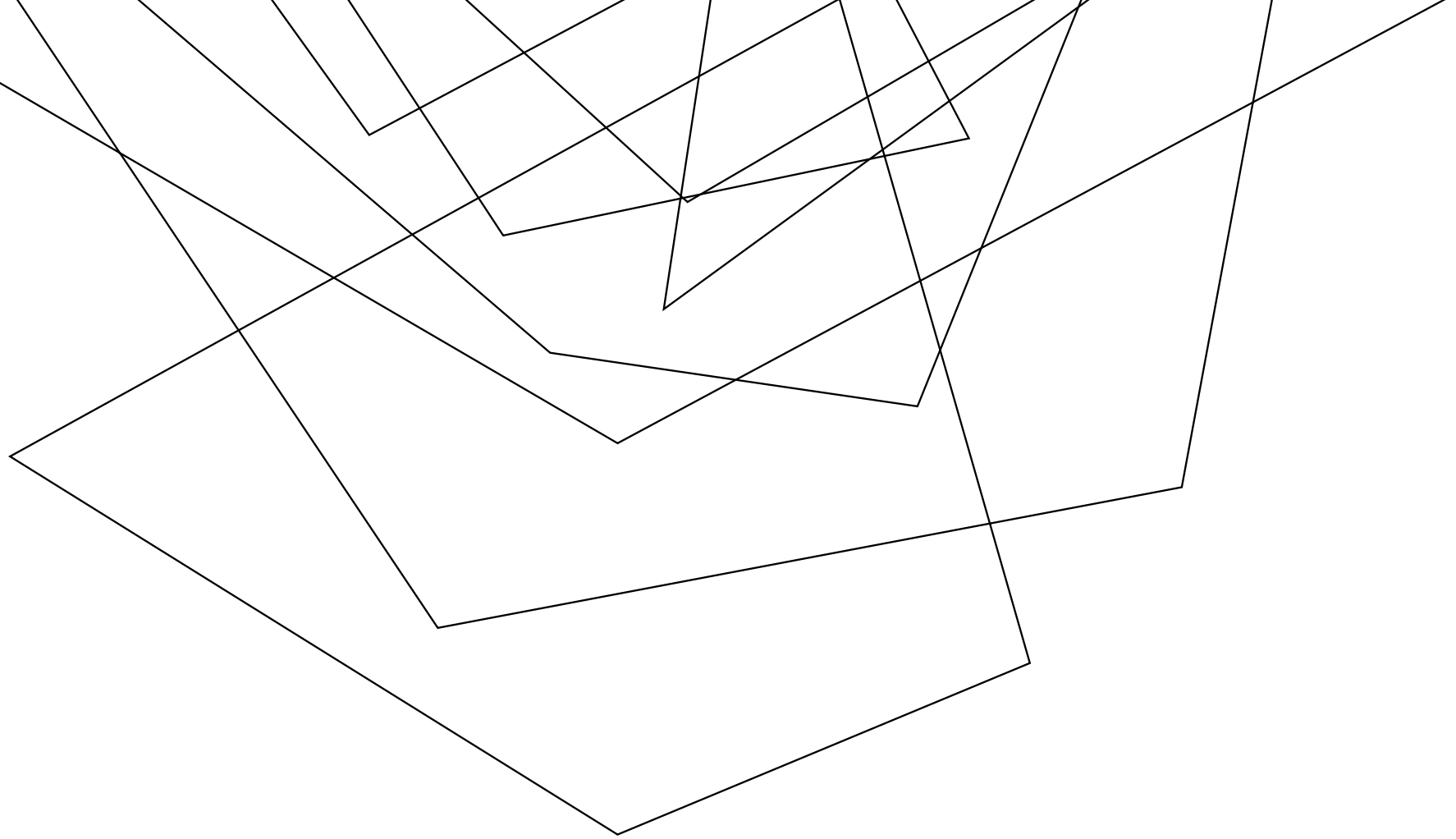
Write your name and answer the following on a piece of paper

Assume a dataflow analysis is operating over a lattice that has an infinite width, but a finite height. Is termination guaranteed?
Why or why not?

ADMINISTRIVIA AND ANNOUNCEMENTS

Note:

Exercise participation based on participation



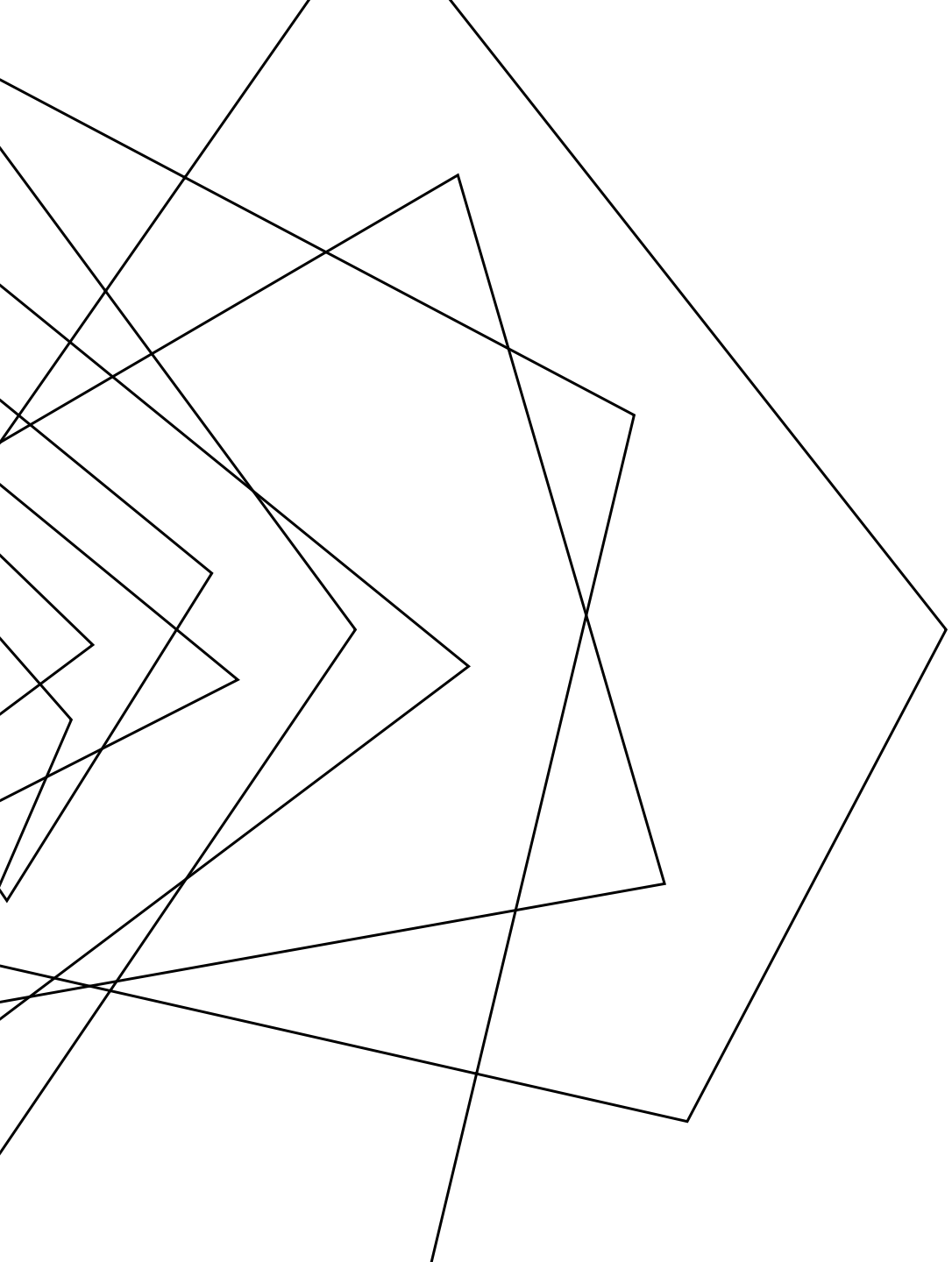
MALWARE

EECS 677: Software Security Evaluation

Drew Davidson

LAST TIME: STATIC ANALYSIS

REVIEW: LAST LECTURE



CLASS PROGRESS

WE’VE GOT OURSELVES A WAY TO REPRESENT PROGRAMS

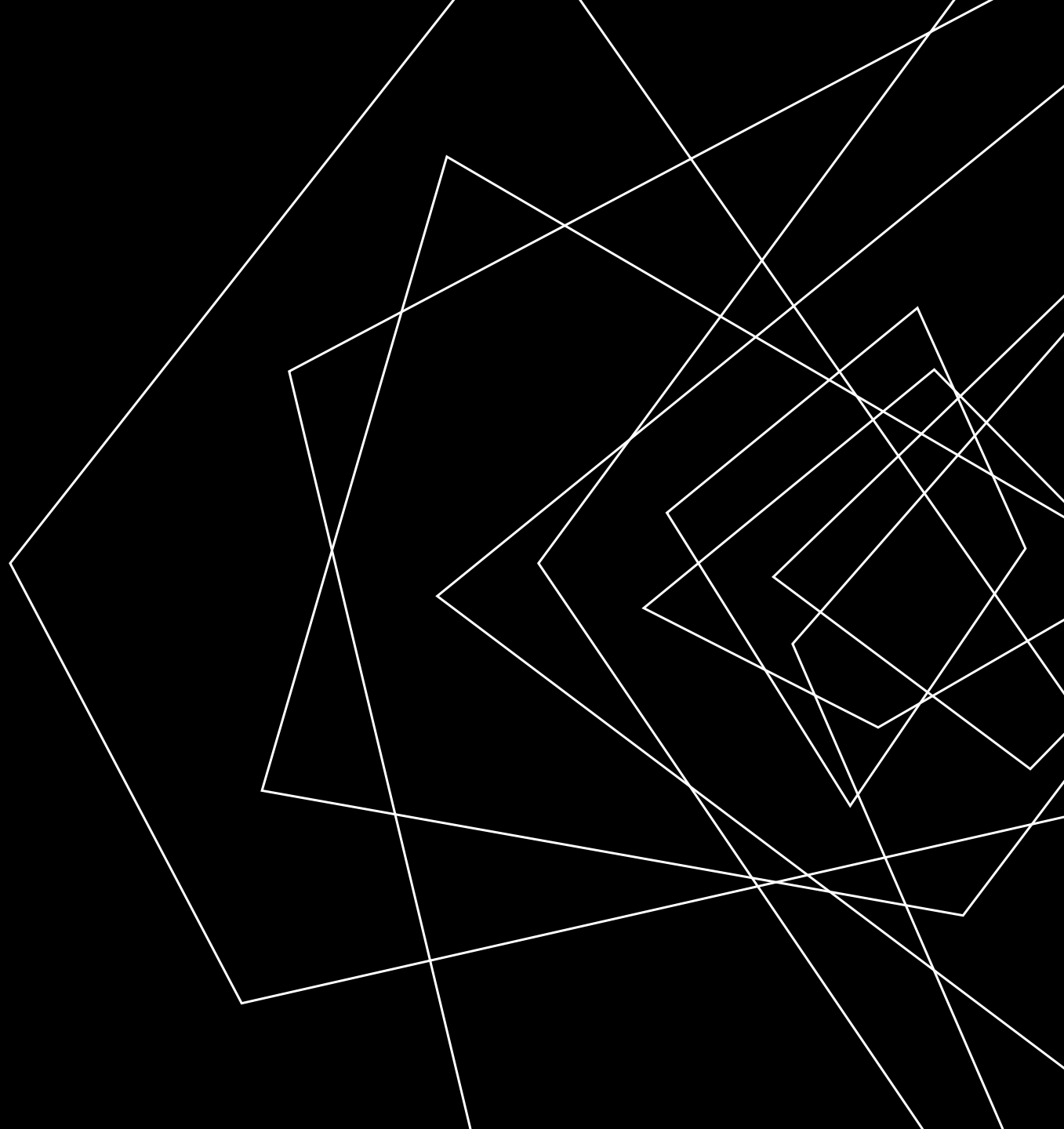
WHAT ARE WE TRYING TO PREVENT?

OVERVIEW

PREVENTING BAD STUFF FROM HAPPENING IN A PROGRAM

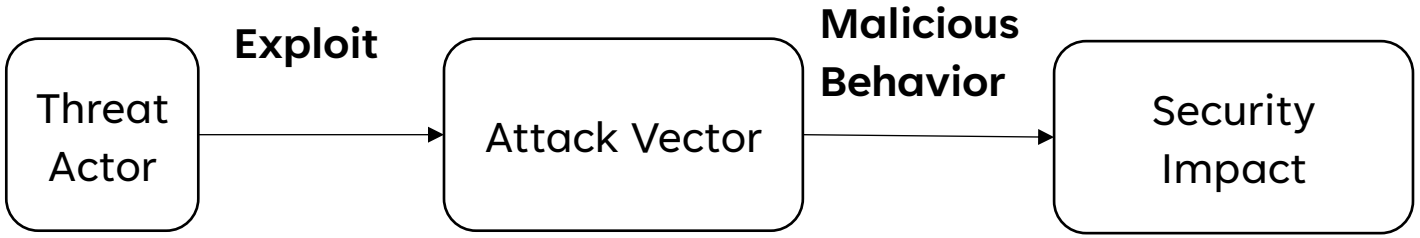
LECTURE OUTLINE

- Terms/Concepts
- Memory Layout
- Memory Attacks



A SECURITY INCIDENT WORKFLOW

MALWARE – TERMS AND CONCEPTS



ADVERSARIES

MALWARE – TERMS AND CONCEPTS

Anyone attempting to cause a security incident

THREAT MODELS

MALWARE – TERMS AND CONCEPTS

Set of adversary goals

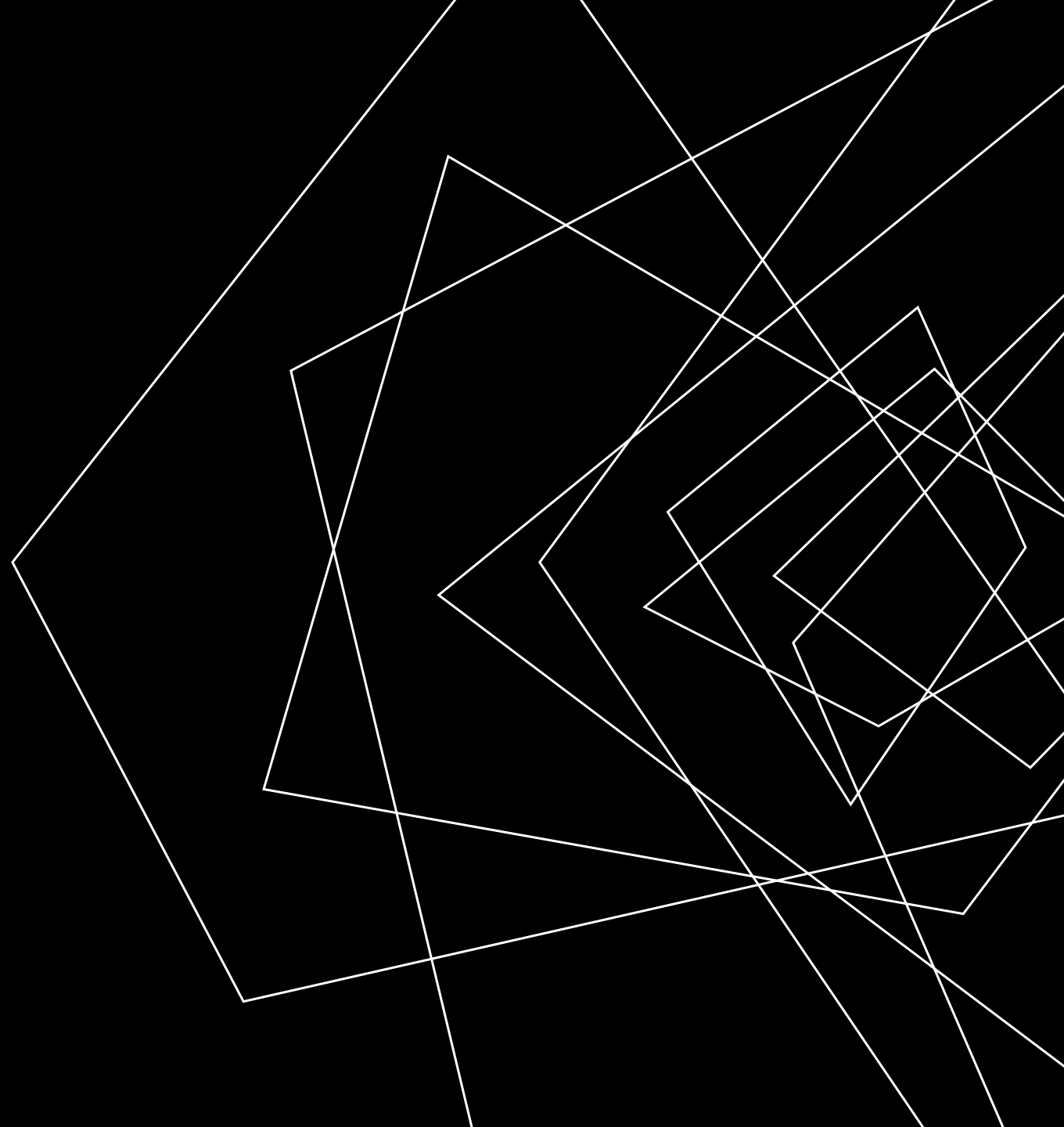
Set of adversary capabilities

SYSTEM SECURITY
MALWARE – TERMS AND CONCEPTS

A common threat model

LECTURE OUTLINE

- Terms/Concepts
- Memory Layout
- Memory Attacks



HOW DO "BAD" PROGRAMS RUN?

MALWARE – TERMS AND CONCEPTS

Reactive Concerns

Evaluating others' programs

- Social engineering
- "Flaws" in system installation policies

Proactive Concerns

Evaluating our own programs

- The program accidentally does damage
- The program contains a vulnerability

We're concerned about all these threats

Behavioral opacity

Remote Code Execution

Drive by Downloads

DATA AND CODE
A HISTORY OF COMPUTING

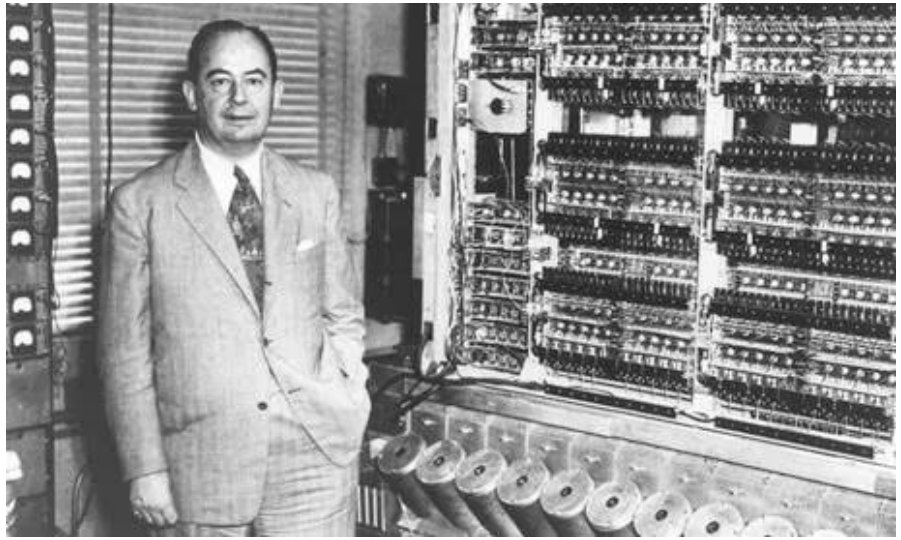
The Von Neumann architecture

Another big idea: Code and data share memory

Good news!

Programs can write code just like any other form of data

Bad news!



Code <u>is</u> data

PROGRAM MEMORY: A 1-D ARRAY

A HISTORY OF SUBVERSION

Code

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

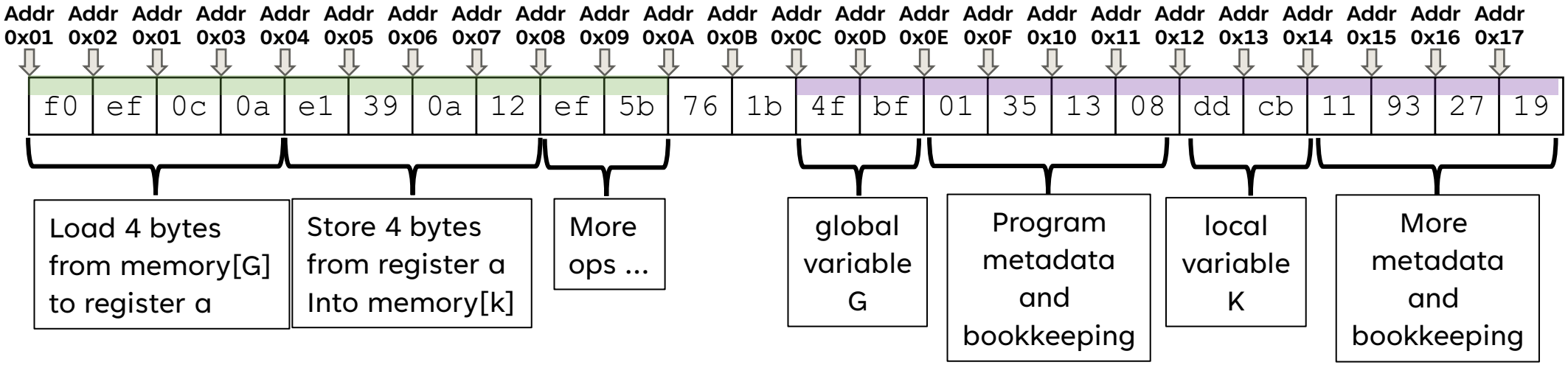
Data

Global variable G

Foo's local variable K

Program instructions (binary sequences)

Program data & metadata (binary sequences)



SIMULATING SOURCE CONSTRUCTS

A HISTORY OF SUBVERSION

Assume main calls foo
foo (recursively) calls foo

Code

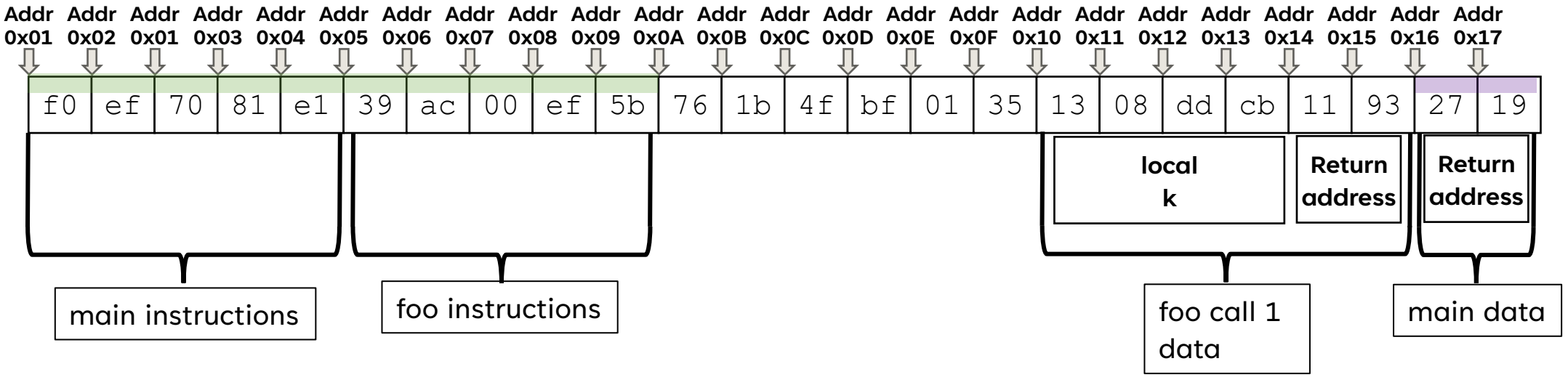
All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K

Program instructions (binary sequences)

Program data & metadata (binary sequences)



SIMULATING SOURCE CONSTRUCTS

A HISTORY OF SUBVERSION

Assume main calls foo
foo (recursively) calls foo

Code

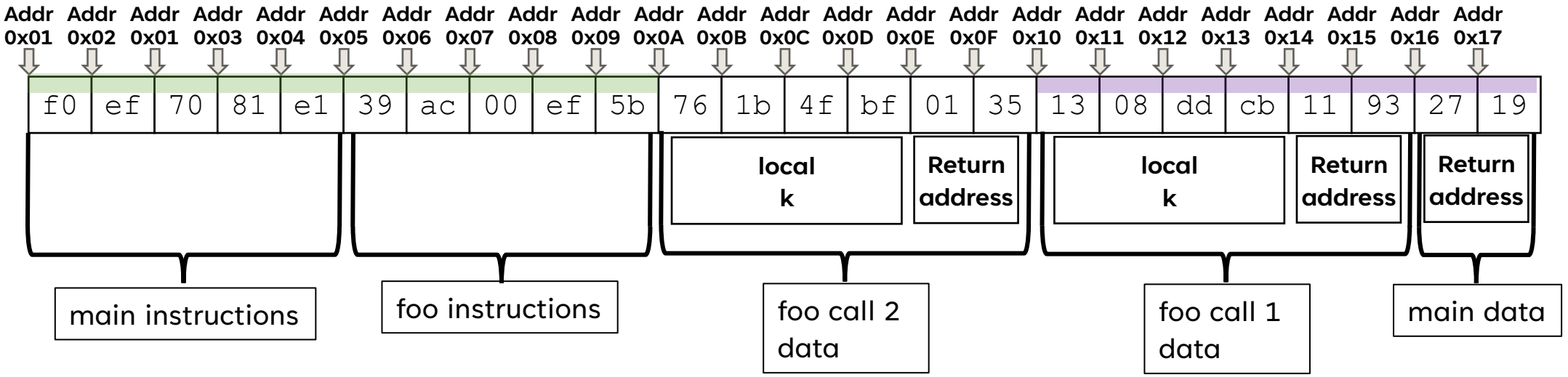
All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K

Program instructions (binary sequences)

Program data & metadata (binary sequences)



SIMULATING SOURCE CONSTRUCTS

A HISTORY OF SUBVERSION

Assume main calls foo
foo (recursively) calls foo

Code

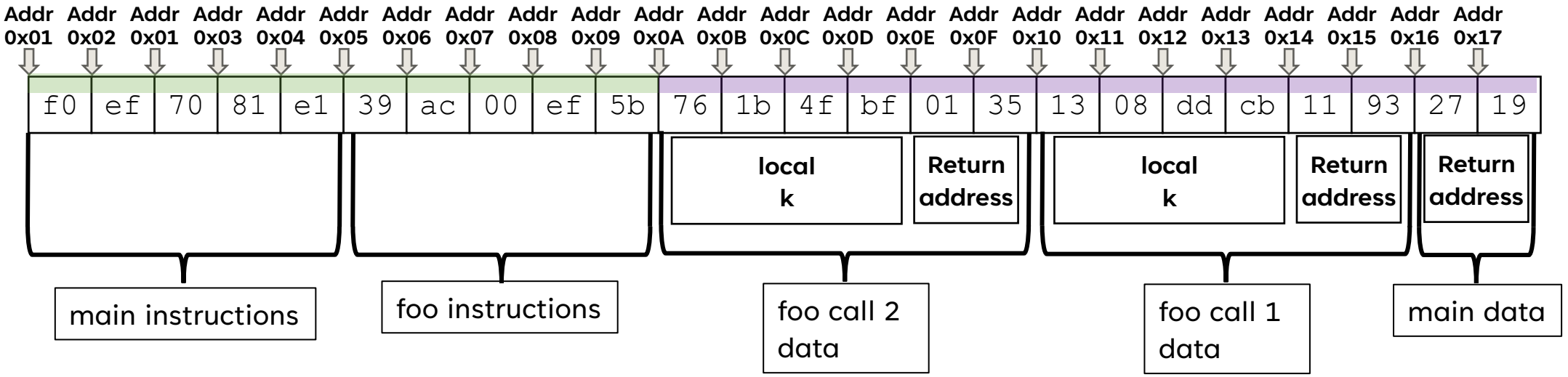
All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K

Program instructions (binary sequences)

Program data & metadata (binary sequences)



SIMULATING SOURCE CONSTRUCTS

A HISTORY OF SUBVERSION

Assume main calls foo
foo (recursively) calls foo

Code

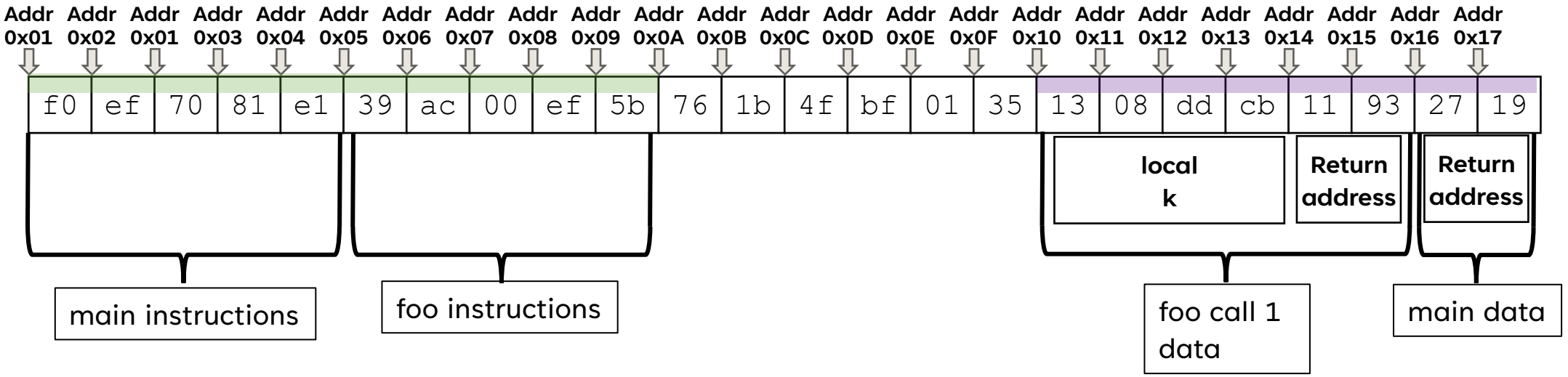
All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K

Program instructions (binary sequences)

Program data & metadata (binary sequences)



SIMULATING SOURCE CONSTRUCTS

A HISTORY OF SUBVERSION

Assume main calls foo
foo (recursively) calls foo

Code

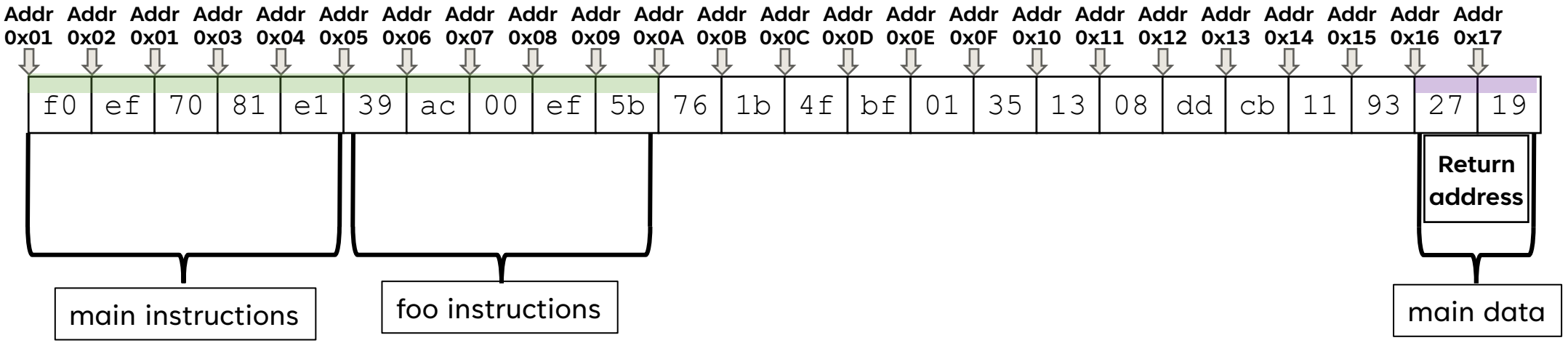
All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K

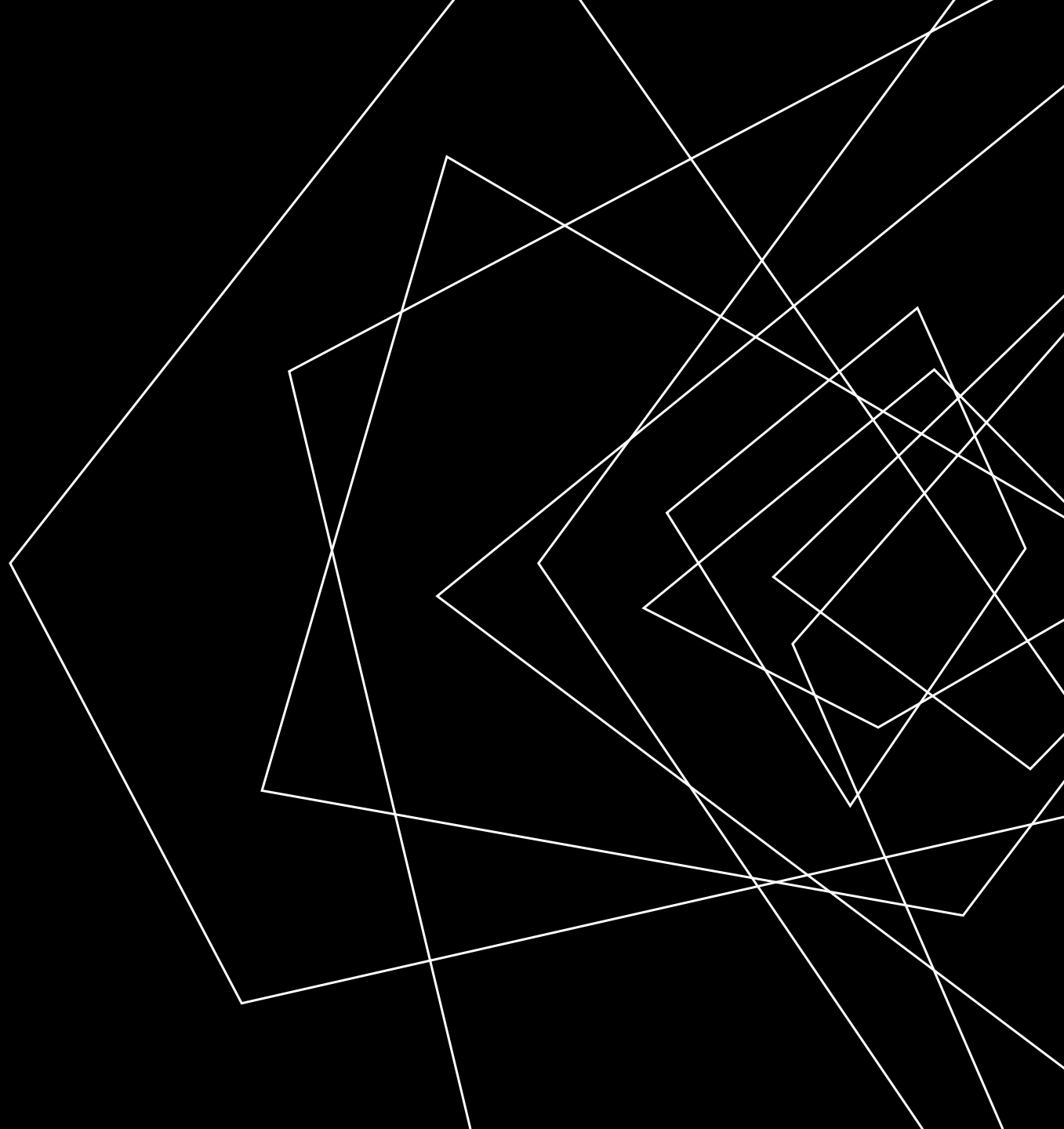
Program instructions (binary sequences)

Program data & metadata (binary sequences)



LECTURE OUTLINE

- Terms/Concepts
- Memory Layout
- Memory Attacks



BREAKING BOUNDS

A HISTORY OF SUBVERSION

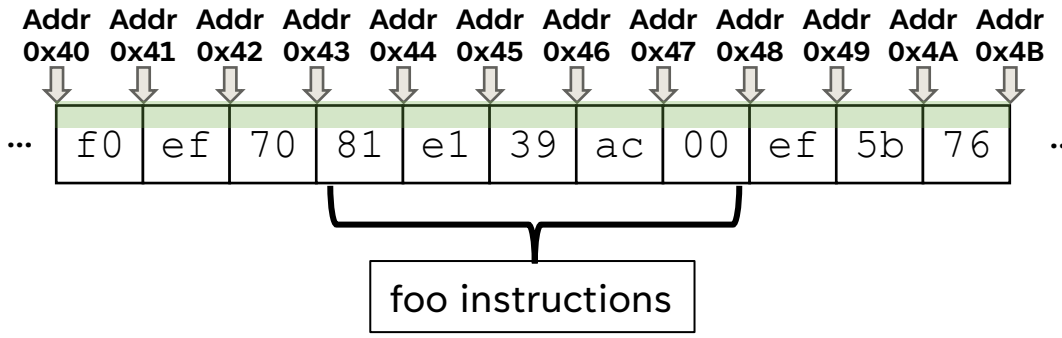
(ascii hex value 0x44)

D

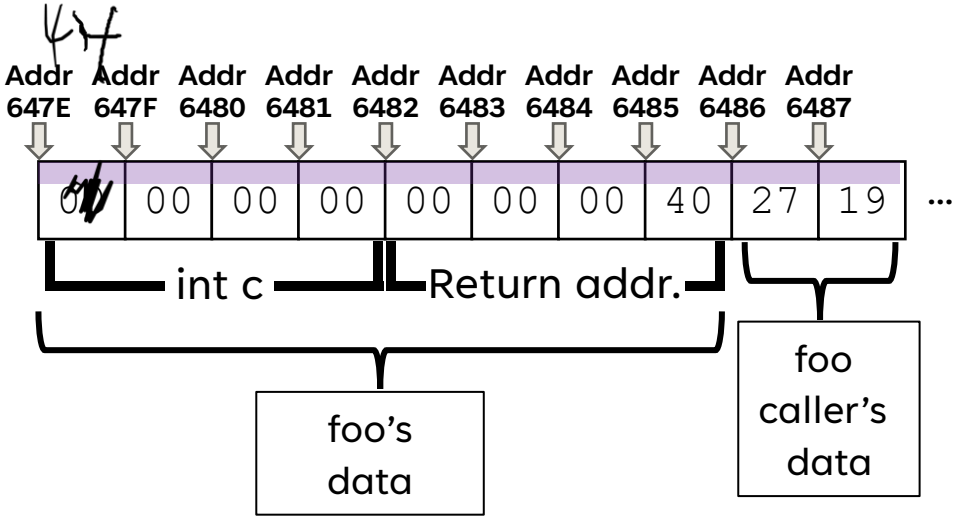
```
foo(){
  int c = getc(stdout);
  *((int *)c) = 2;
}
```

```
@stdin = external global %struct._IO_FILE*

define void @foo() {
  %stdInPtr = load %struct._IO_FILE*, %struct._IO_FILE** @stdin
  %char = call i32 @getc(%struct._IO_FILE* %stdInPtr)
  %charInt = sext i32 %char to i64
  %charPtr = inttoptr i64 %charInt to i32*
  store i32 2, i32* %charPtr
  ret void
}

declare i32 @getc(%struct._IO_FILE*)
```

Program instructions



Program (meta)data



BREAKING BOUNDS

A HISTORY OF SUBVERSION

(ascii hex value 0x44)

D

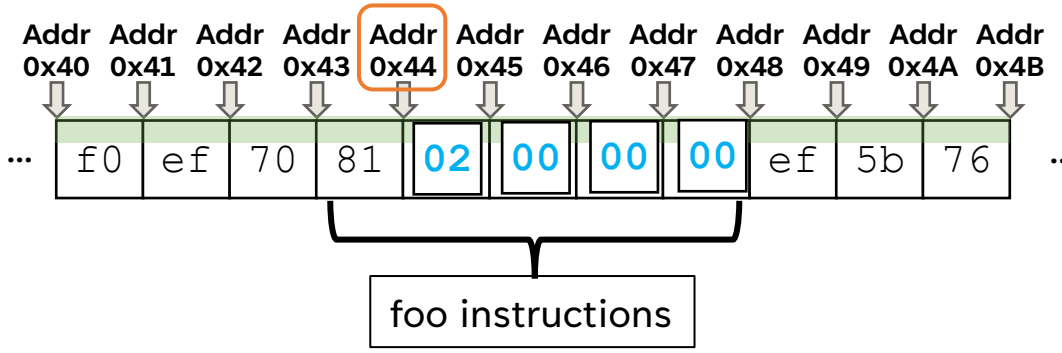
```
foo(){
    int c = getc(stdout);
*((int *)c) = 2;
}
```

Pointer to address 0x44

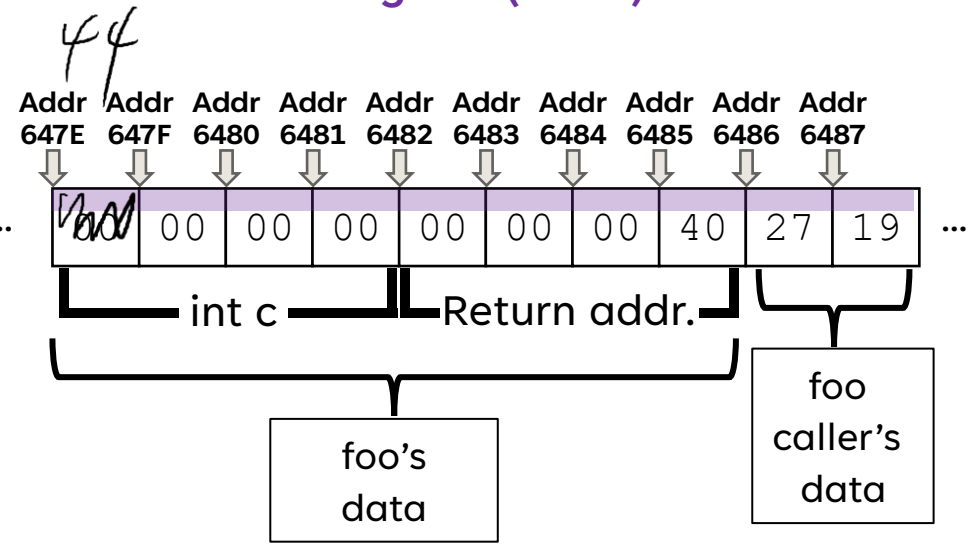
4-byte value 0x00000002

```
@stdin = external global %struct._IO_FILE*

define void @foo() {
  %stdInPtr = load %struct._IO_FILE*, %struct._IO_FILE** @stdin
  %char = call i32 @getc(%struct._IO_FILE* %stdInPtr)
  %charInt = sext i32 %char to i64
  %charPtr = inttoptr i64 %charInt to i32*
  store i32 2, i32* %charPtr
  ret void
}

declare i32 @getc(%struct._IO_FILE*)
```

Program instructions



Program (meta)data



BREAKING BOUNDS

A HISTORY OF SUBVERSION

(ascii hex value 0x44)

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

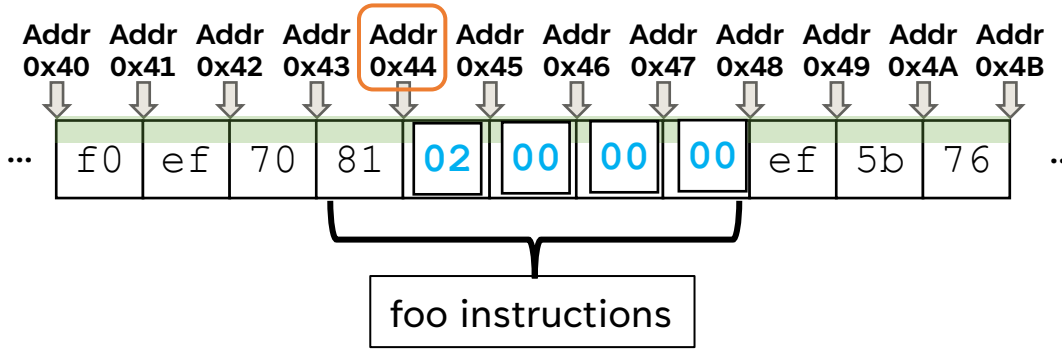
D

Pointer to address 0x44

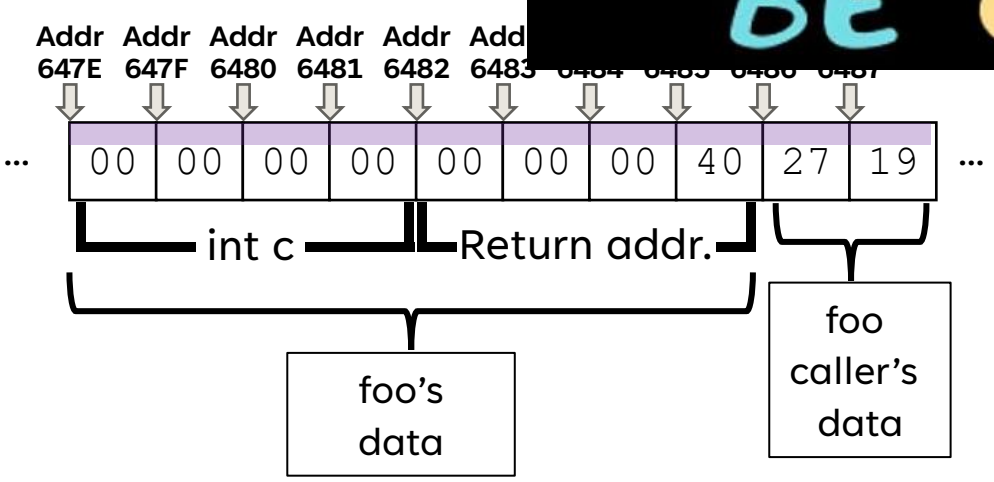
4-byte value 0x00000002

```
@stdin = external global %struct._

define void @foo() {
  %stdInPtr = load %struct._IO_FIL
  %char = call i32 @getc(%struct._
  %charInt = sext i32 %char to i64
  %charPtr = inttoptr i64 %charInt
  store i32 2, i32* %charPtr
  ret void
}

declare i32 @getc(%struct._IO_FILE
```

Program instructions



Program



BUFFER OVERFLOWS

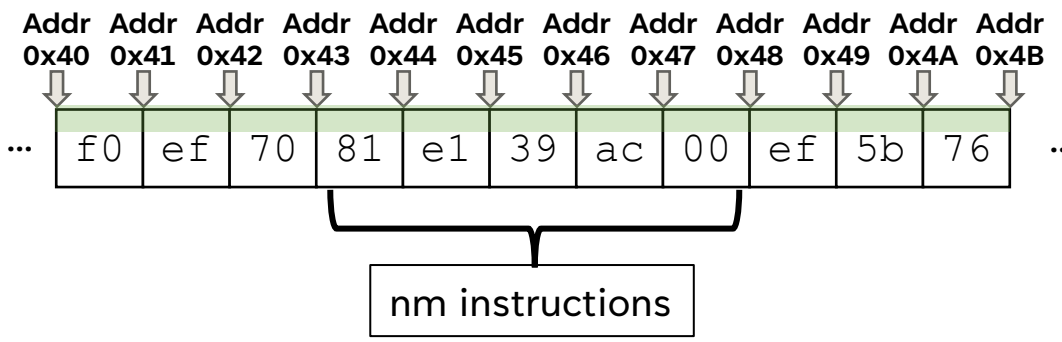
A HISTORY OF SUBVERSION

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

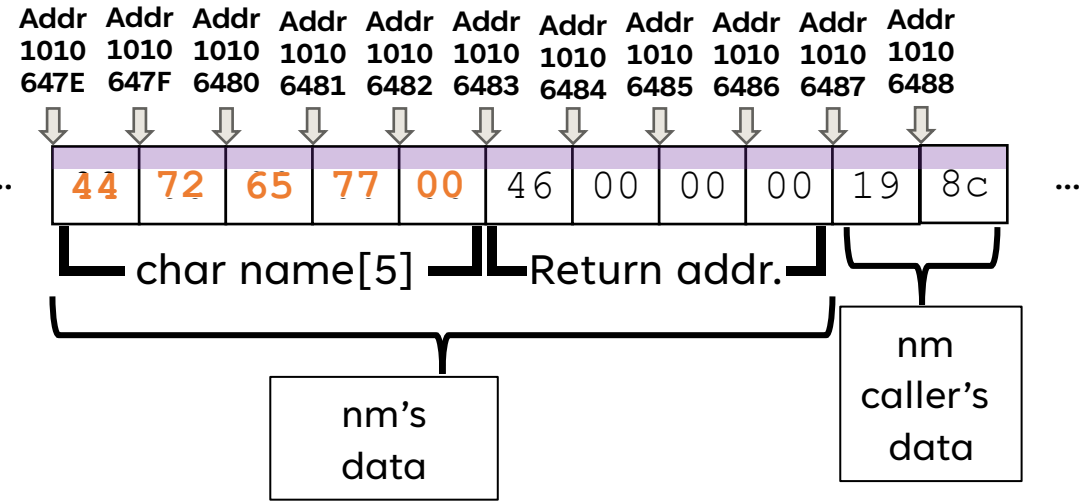
D r e W (end)

0x44 0x72 0x65 0x77 0x00

Program instructions



Program (meta)data



BUFFER OVERFLOWS

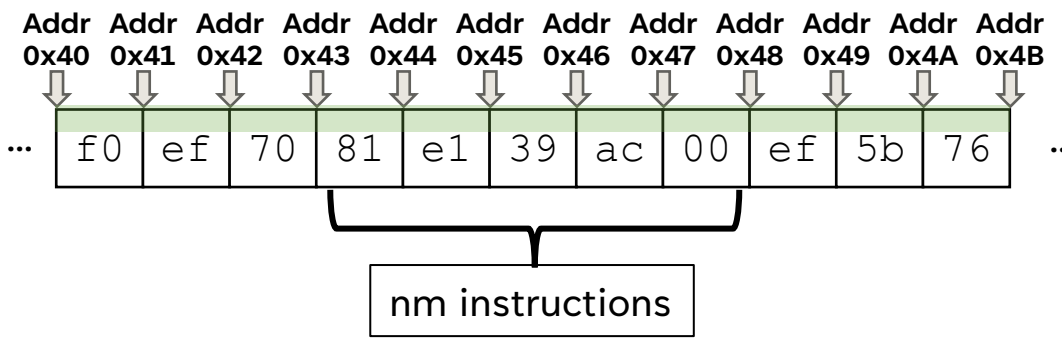
A HISTORY OF SUBVERSION

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

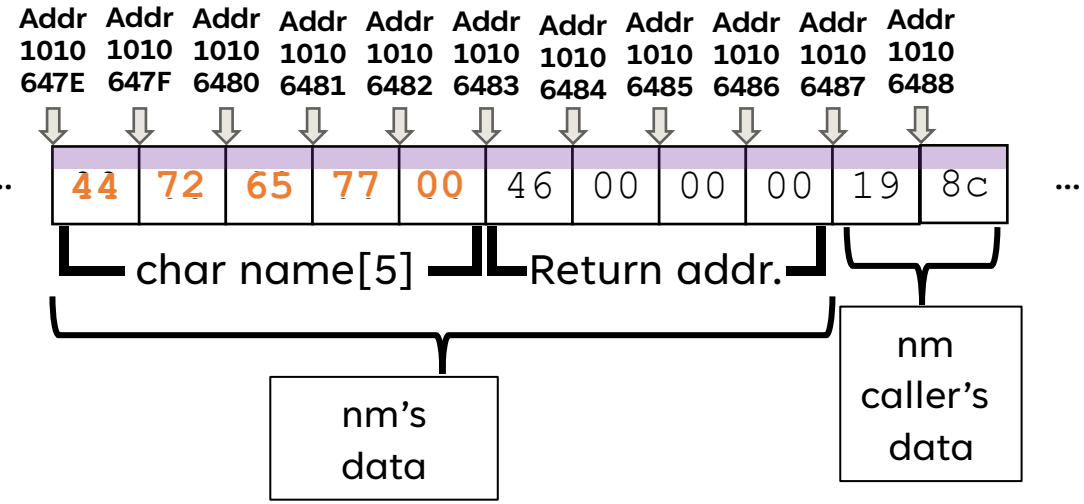
D r e W (end)

0x44 0x72 0x65 0x77 0x00

Program instructions



Program (meta)data



BUFFER OVERFLOWS

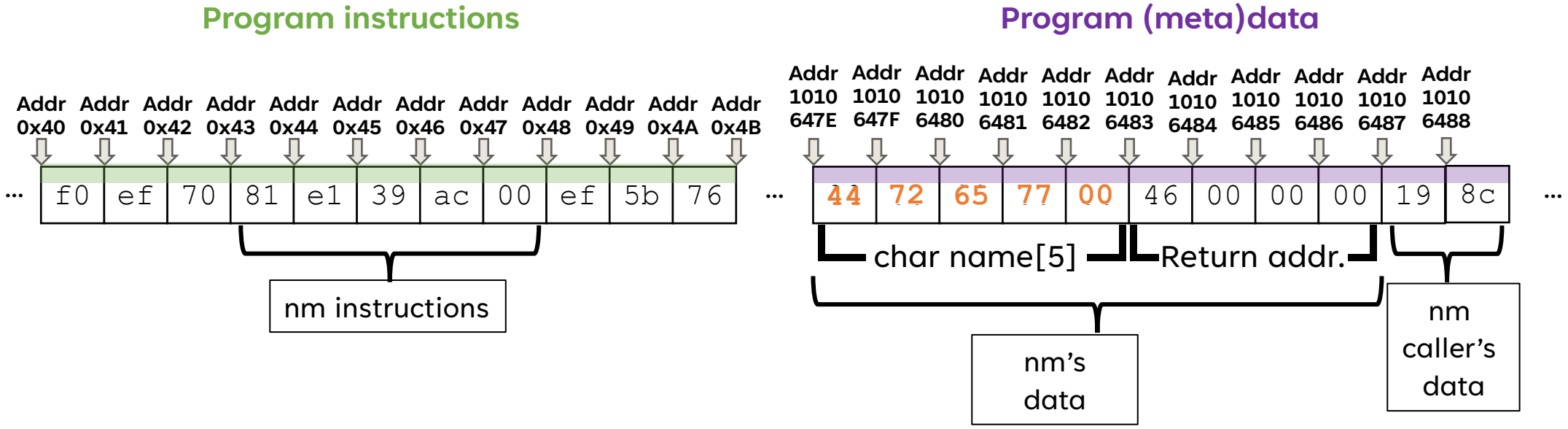
A HISTORY OF SUBVERSION

```
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

Here's the thing... Drew is just my *nickname*



BUFFER OVERFLOWS

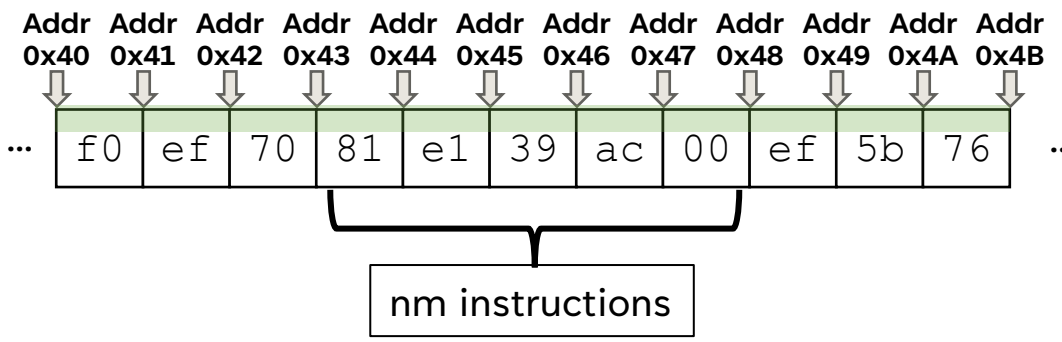
A HISTORY OF SUBVERSION

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

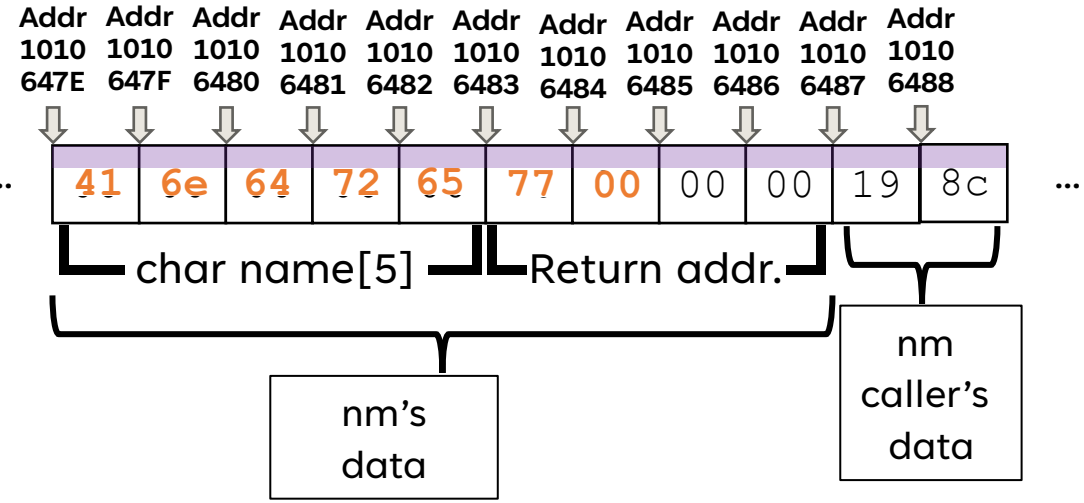
A n d r e w (end)

0x41 0x6e 0x64 0x72 0x65 0x77 0x00

Program instructions



Program (meta)data



BUFFER OVERFLOWS

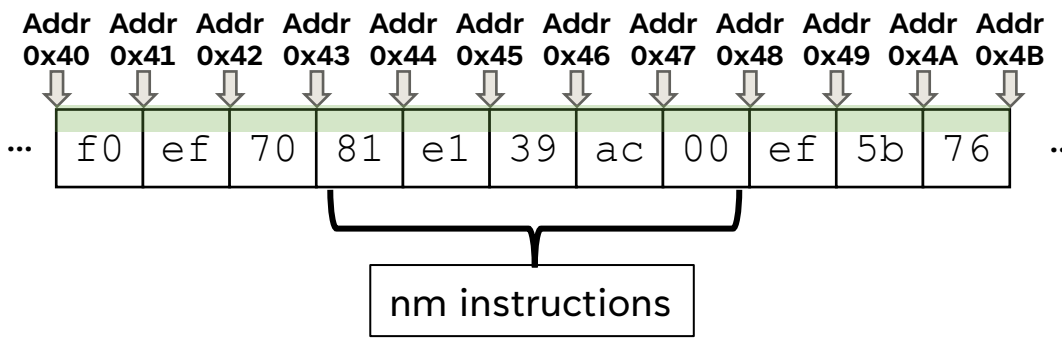
A HISTORY OF SUBVERSION

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

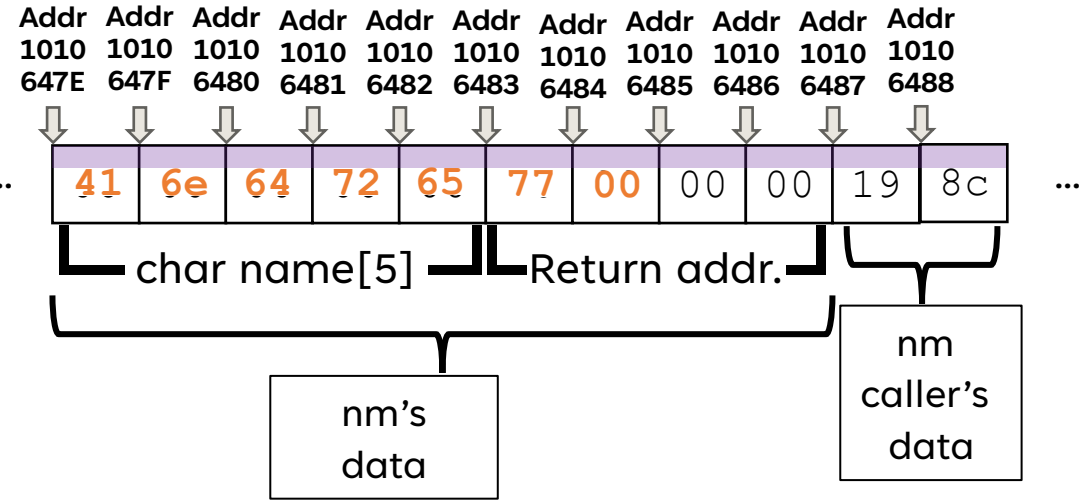
A n d r e w (end)

0x41 0x6e 0x64 0x72 0x65 0x77 0x00

Program instructions



Program (meta)data



BUFFER OVERFLOWS

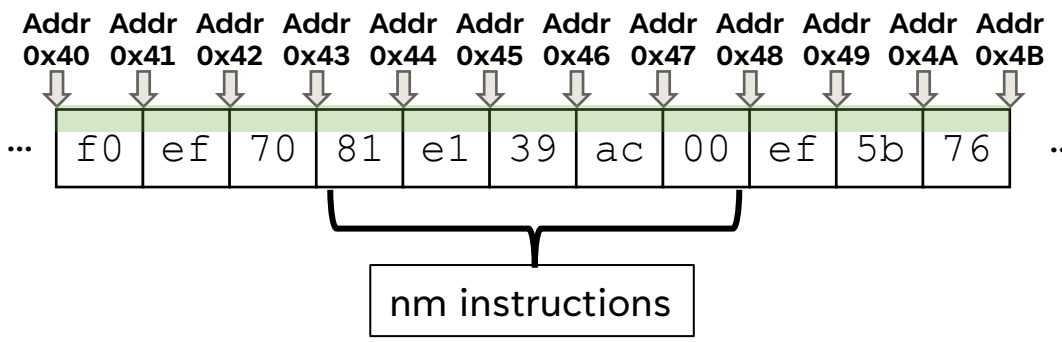
A HISTORY OF SUBVERSION

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

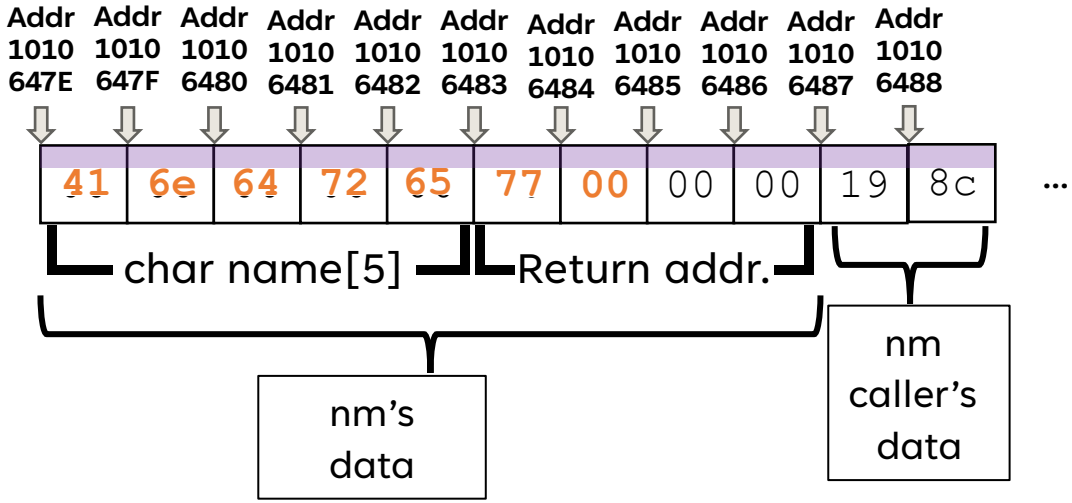
A n d r e w (end)

0x41 0x6e 0x64 0x72 0x65 0x77 0x00

Program instructions



Program (meta)data



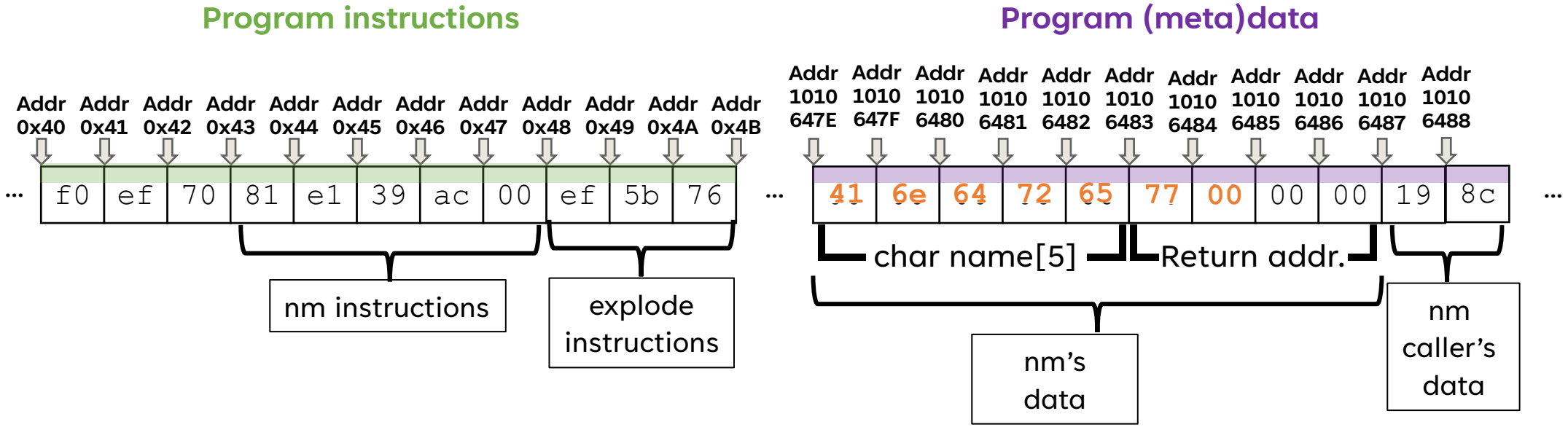
BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

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

A n d r e w (end)

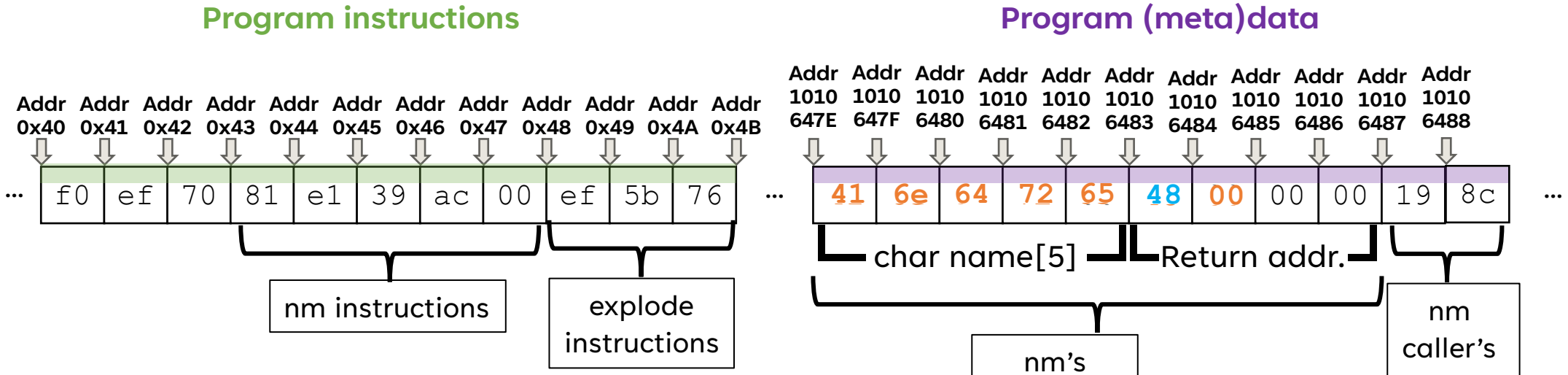
0x41 0x6e 0x64 0x72 0x65 0x77 0x00



BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

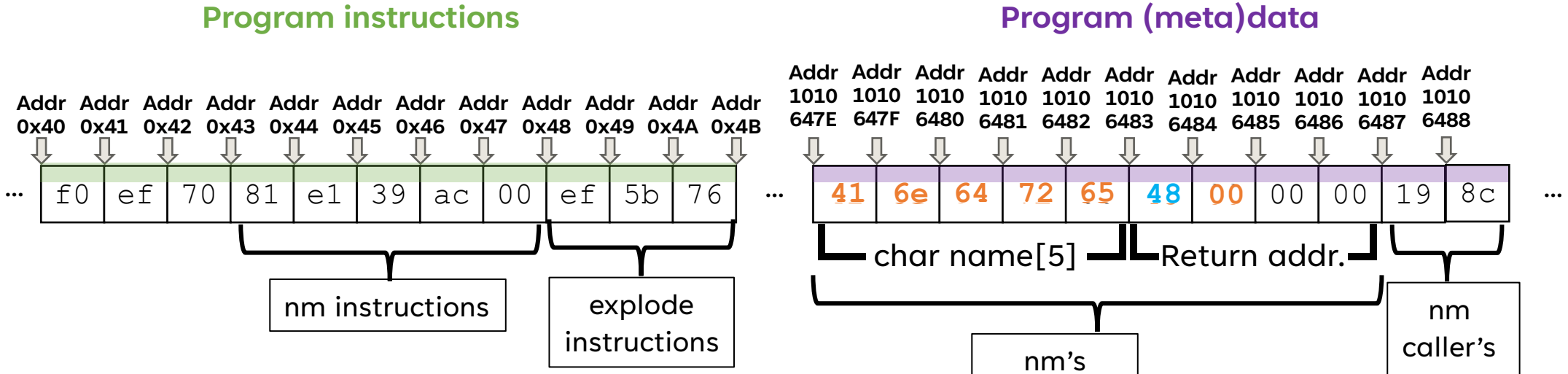
```
int nm(){
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf("hi %s\n", name);
    return 0;
}
void explode(){
    … //destroyTheComputer
}
```



BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

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



BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

```
int nm(){
    char name[5];
    printf("Enter
    gets(name);
    printf("hi %s\
    return 0;
}
void explode(){
    … //destroyTheCompu
}
```

H

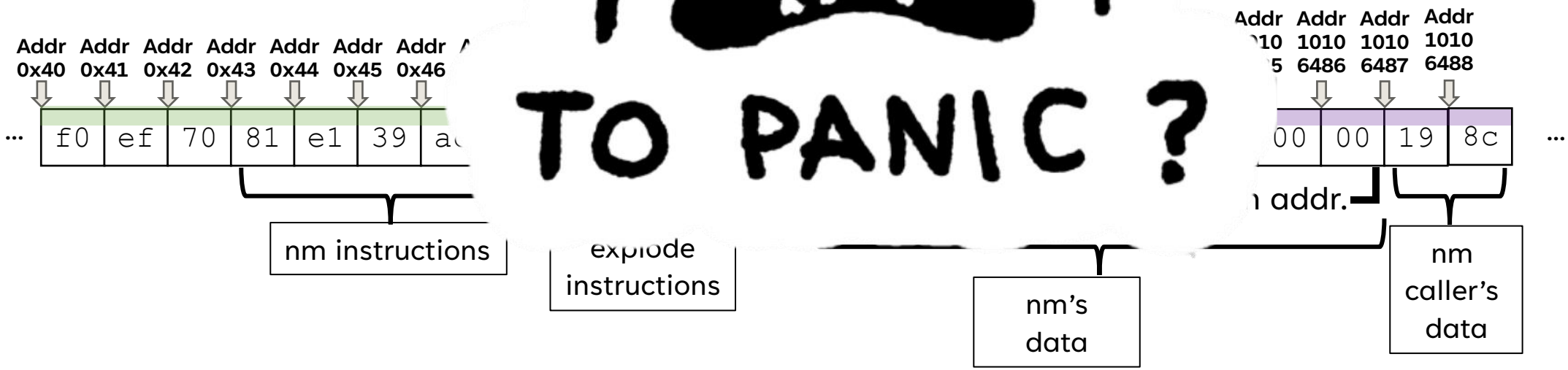
e W (end)

0x65 0x77 0x00

0x48

Program instructions

(meta)data

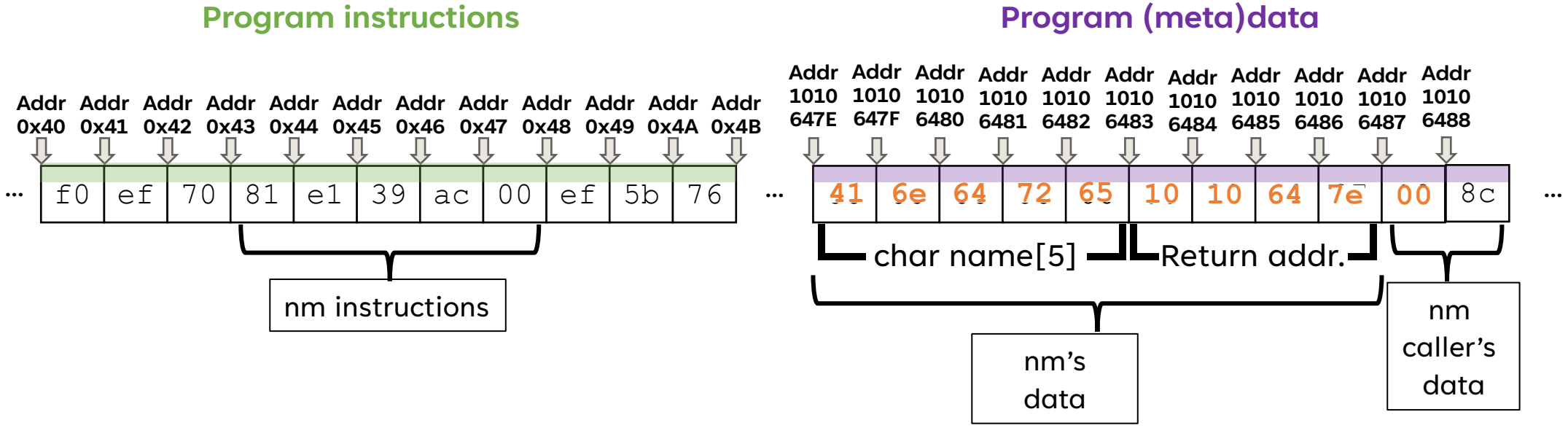


SCRIPT INJECTION

A HISTORY OF SUBVERSION

```
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



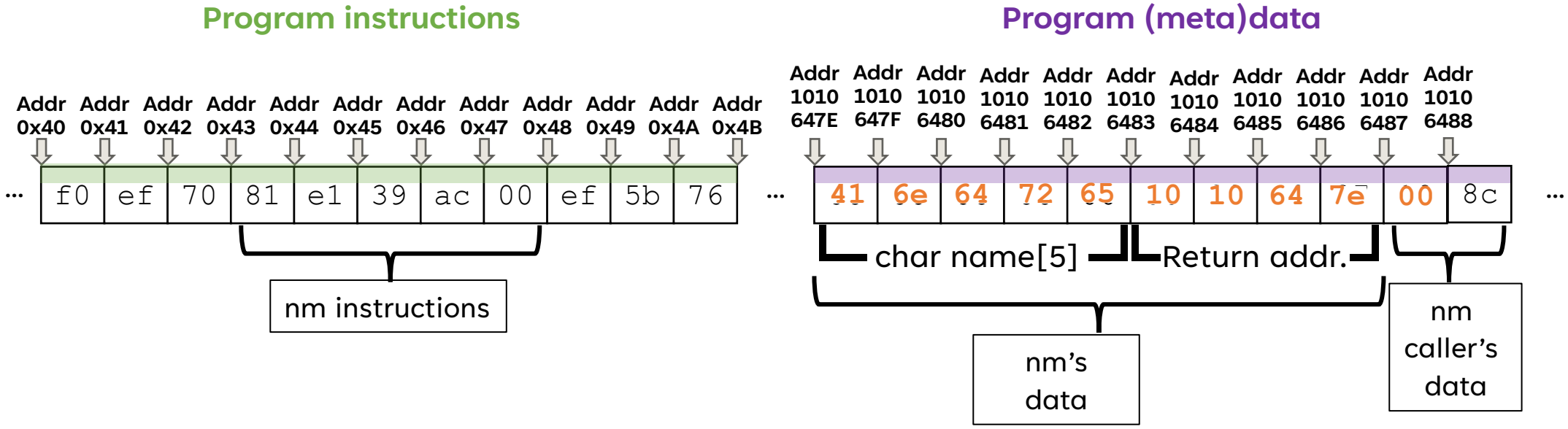
SCRIPT INJECTION

A HISTORY OF SUBVERSION

```
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



SCRIPT INJECTION

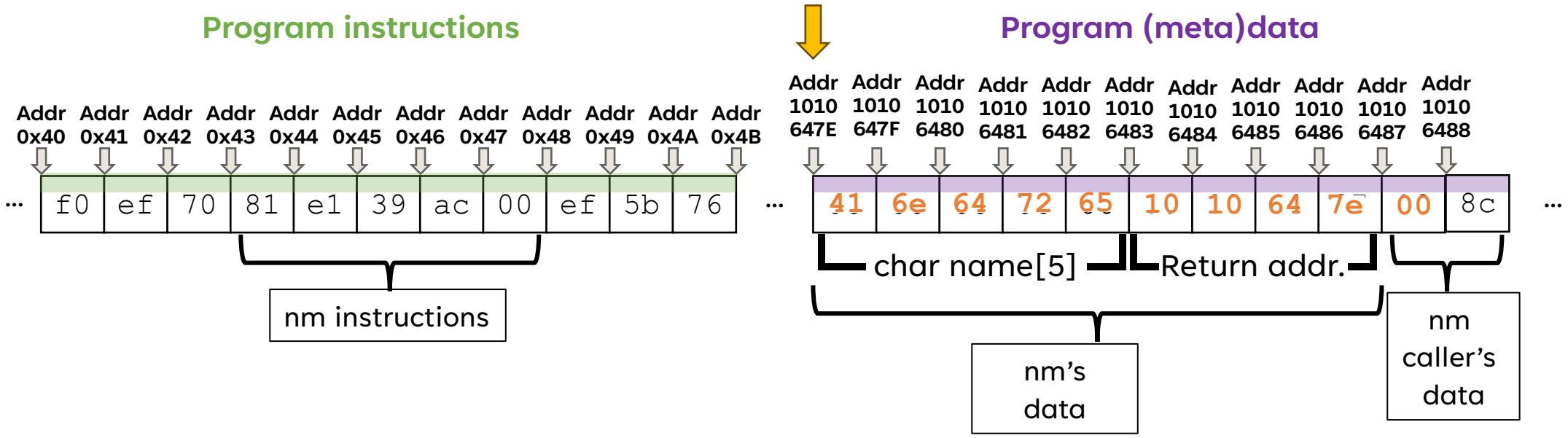
A HISTORY OF SUBVERSION

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

change this input…

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

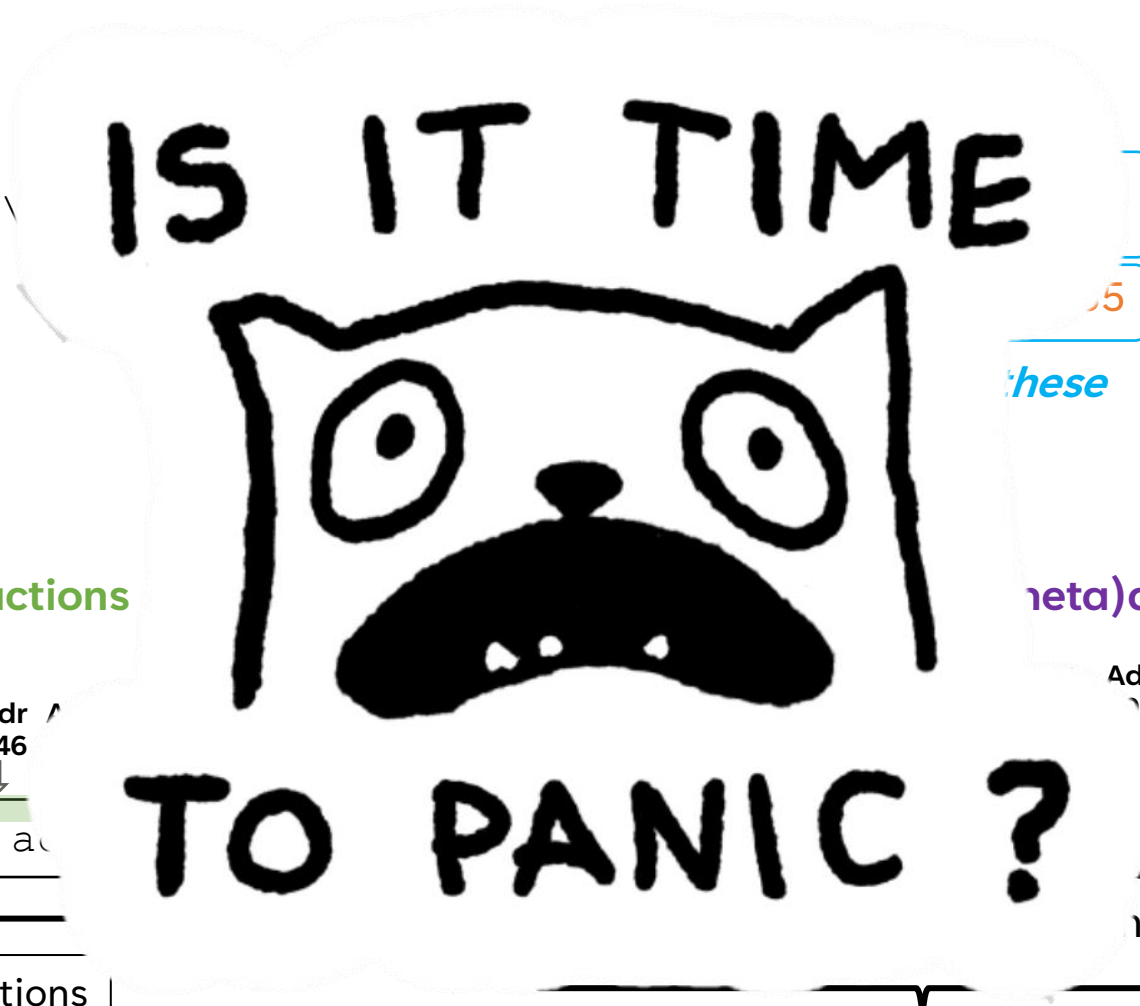
…execute whatever these Instructions are



SCRIPT INJECTION

A HISTORY OF SUBVERSION

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



LF LF d ~ (end)

5 0x10 0x10 0x64 0x7e 0x00

these

Program instructions

(meta)data

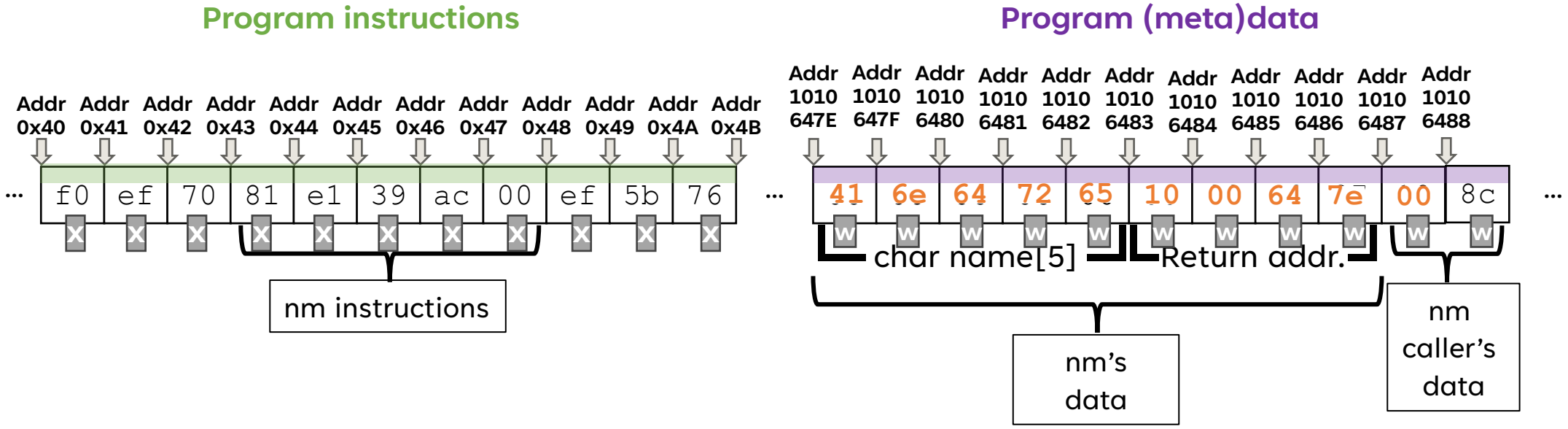
SCRIPT INJECTION: THE FIX

A HISTORY OF SUBVERSION

```
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

W⊗X: hardware extension, assure every byte is EITHER executable OR writable (NEVER both)



SCRIPT INJECTION: THE FIX

A HISTORY OF SUBVERSION

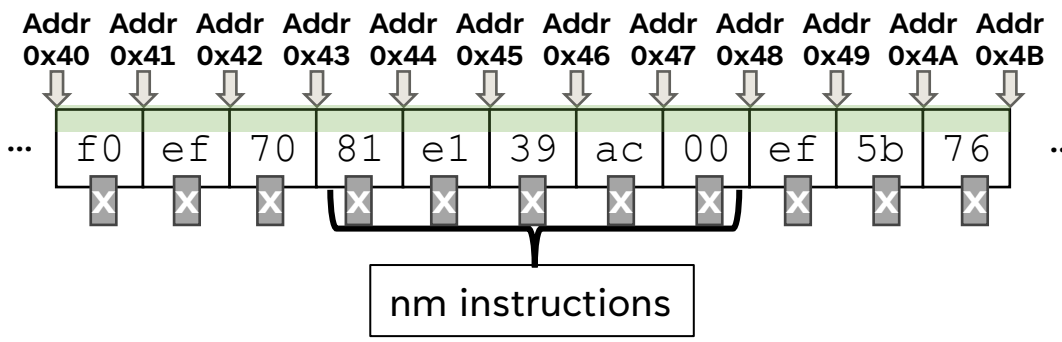
```
int nm(){
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf("hi %s\n", name);
    return 0;
}
```

A n d r e LF LF d ~ (end)

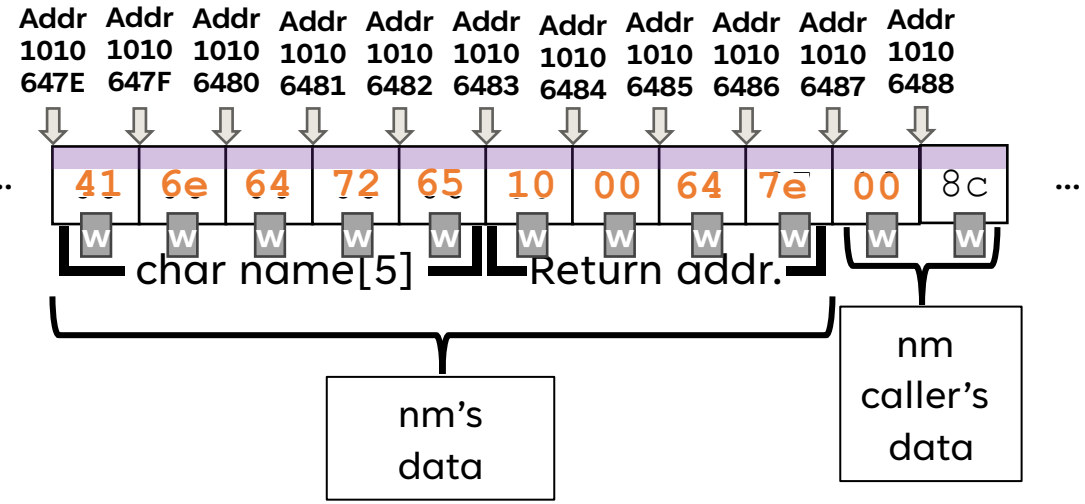
0x41 0x6e 0x64 0x72 0x65 0x10 0x10 0x64 0x7e 0x00

W⊗X: hardware extension, assure every byte is EITHER executable OR writable (NEVER both)

Program instructions



Program (meta)data



SCRIPT INJECTION: THE FIX

A HISTORY OF SUBVERSION

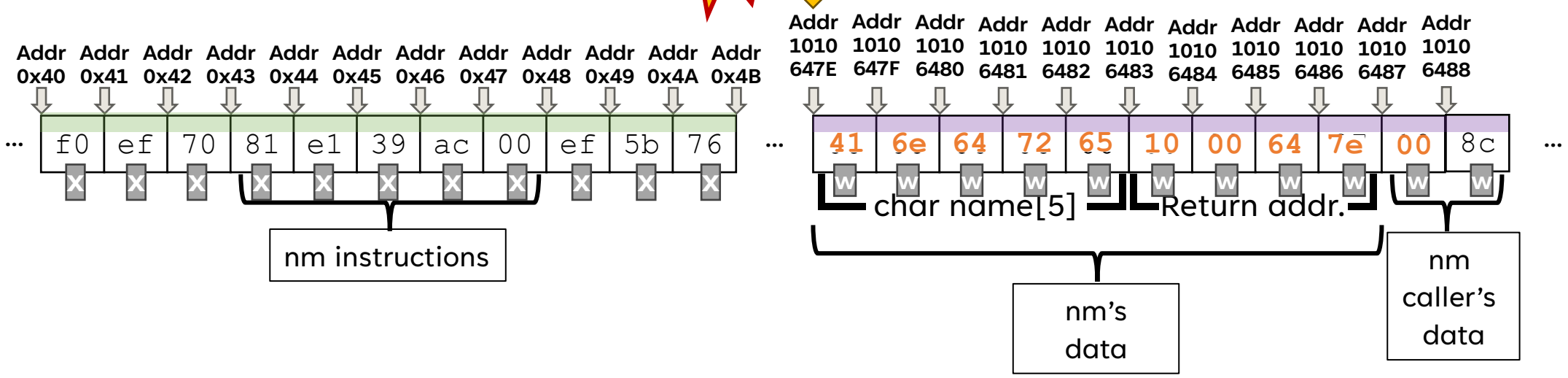
```
int nm(){
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf("hi %s\n", name);
    return 0;
}
```

A n d r e LF LF d ~ (end)

0x41 0x6e 0x64 0x72 0x65 0x10 0x10 0x64 0x7e 0x00

W⊗X: hardware extension, assure every byte is EITHER executable OR writable (NEVER both)

Crash

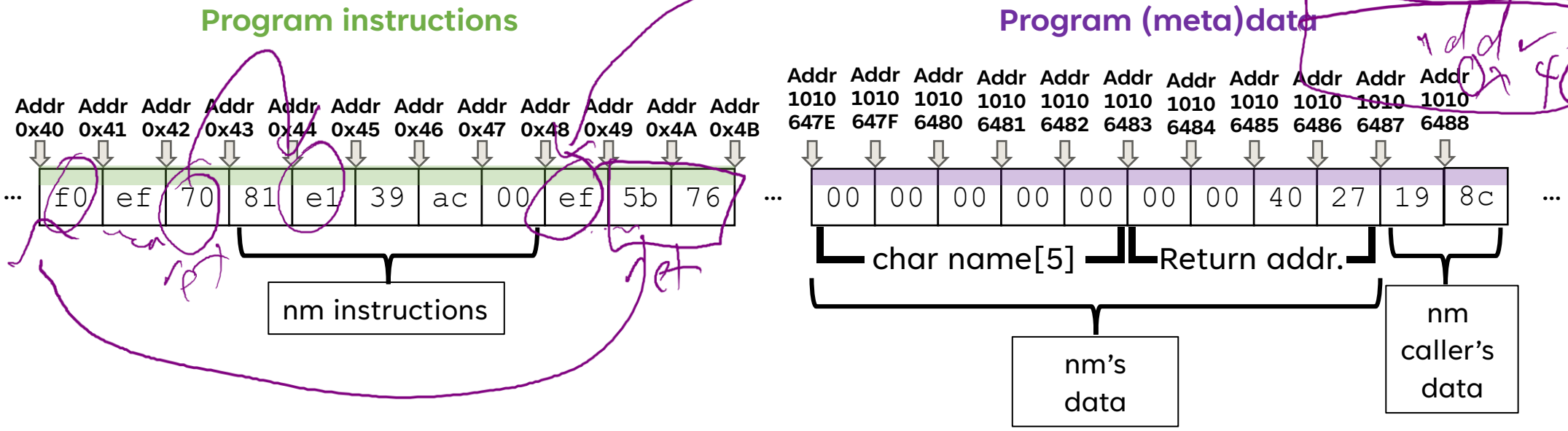


ROP

A HISTORY OF SUBVERSION

```
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



ROP CHALLENGES

A HISTORY OF SUBVERSION

THE PRACTICALITY OF THIS ATTACK MAY SEEM LIMITED

Are (sub)sequences present in process code to do the attack?

Are the (sub)sequences placed in predictable positions?

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

2014 IEEE Symposium on Security and Privacy

Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University

***Abstract*—We show that it is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary or source code, against services that restart after a crash. This makes it possible to hack proprietary closed-binary services, or open-source servers manually compiled and installed from source where the binary remains unknown to the attacker. Traditional techniques are usually paired against a particular binary and distribution where the hacker knows the location of useful gadgets for Return Oriented Programming (ROP). Our Blind ROP (BROP) attack instead remotely finds enough ROP gadgets to perform a `write` system call and transfers the vulnerable binary over the network, after which an exploit can be completed using known techniques. This is accomplished by leaking a single bit of information based on whether a process crashed or not when given a particular input string. BROP requires a stack vulnerability and a service that restarts after a crash. We implemented Braille, a fully automated exploit that yielded a shell in under 4,000 requests (20 minutes) against a contemporary nginx vulnerability, yaSSL + MySQL, and a toy proprietary server written by a colleague. The attack works against modern 64-bit Linux with address space layout randomization (ASLR), no-execute page protection (NX) and stack canaries.**

I. Introduction

Attackers have been highly successful in building exploits with varying degrees of information on the target. Open-source software is most within reach since attackers can audit the code to find vulnerabilities. Hacking closed-source software is also possible for more motivated attackers through the use of fuzz testing and reverse engineering. In an effort to understand an attacker's limits, we pose the following question: *is it possible for attackers to extend their reach and create exploits for proprietary services when neither the source nor binary code is available?* At first sight this goal may seem unattainable because today's exploits rely on having a copy of the target binary for use in Return Oriented Programming (ROP) [1]. ROP is necessary because, on modern systems, non-executable (NX) memory protection has largely prevented code injection attacks.

To answer this question we start with the simplest possible vulnerability: stack buffer overflows. Unfortunately these are still present today in popular software (*e.g.,* nginx CVE-2013-2028 [2]). One can only speculate that bugs such as these go unnoticed in proprietary software, where the source (and binary) has not been under the heavy scrutiny of the public and security specialists. However, it is certainly possible for an attacker to use fuzz testing to find potential bugs through known or reverse engineered service interfaces. Alternatively, attackers can target known vulnerabilities in popular open-source libraries (*e.g.,* SSL or a PNG parser) that may be used by proprietary services. The challenge is developing a methodology for exploiting these vulnerabilities when information about the target binary is limited.

One advantage attackers often have is that many servers restart their worker processes after a crash for robustness. Notable examples include Apache, nginx, Samba and OpenSSH. Wrapper scripts like `mysqld_safe.sh` or daemons like `systemd` provide this functionality even if it is not baked into the application. Load balancers are also increasingly common and often distribute connections to large numbers of identically configured hosts executing identical program binaries. Thus, there are many situations where an attacker has potentially infinite tries (until detected) to build an exploit.

We present a new attack, Blind Return Oriented Programming (BROP), that takes advantage of these situations to build exploits for proprietary services for which both the binary and source are unknown. The BROP attack assumes a server application with a stack vulnerability and one that is restarted after a crash. The attack works against modern 64-bit Linux with ASLR (Address Space Layout Randomization), non-executable (NX) memory, and stack canaries enabled. While this covers a large number of servers, we can not currently target Windows systems because we have yet to adapt the attack to the Windows ABI. The attack is enabled by two new techniques:

1) Generalized stack reading: this generalizes a known technique, used to leak canaries, to also leak saved return addresses in order to defeat ASLR on 64-bit even when Position Independent Executables (PIE) are used.
2) Blind ROP: this technique remotely locates ROP gadgets.

Both techniques share the idea of using a single stack vulnerability to leak information based on whether a server process crashes or not. The stack reading technique overwrites the stack byte-by-byte with possible guess values, until the correct one is found and the server does not crash, effectively reading (by overwriting) the stack. The Blind ROP attack remotely finds enough gadgets to perform the `write` system call, after which the server's binary can be transferred from memory to the attacker's socket. At this point, canaries, ASLR and NX have been defeated and the exploit can proceed using known techniques.

The BROP attack enables robust, general-purpose exploits for three new scenarios:

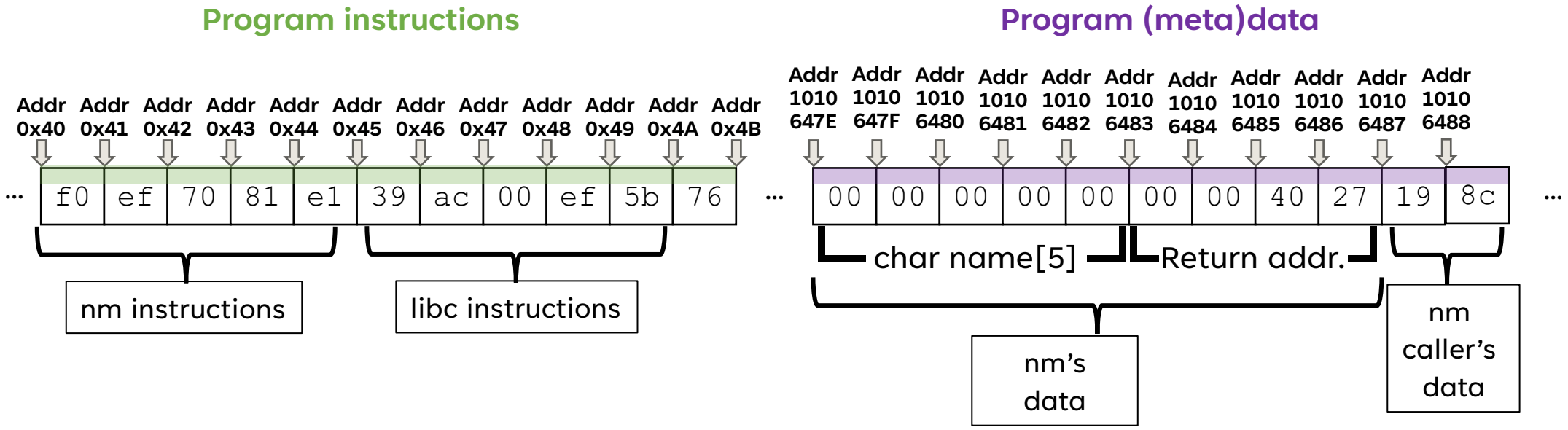
1) Hacking proprietary closed-binary services. One may notice a crash when using a remote service or discover one through remote fuzz testing.
2) Hacking a vulnerability in an open-source library thought to be used in a proprietary closed-binary service. A popular SSL library for example may have

DOI 10.1109/SP.2014.22

RETURN-INTO-LIBC

A HISTORY OF SUBVERSION

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




(yes, time to panic)

BEYOND ROP AS RCE

A HISTORY OF SUBVERSION

Return oriented Programming constitutes a form of Remote code execution

It’s not the *only* form of remote code execution

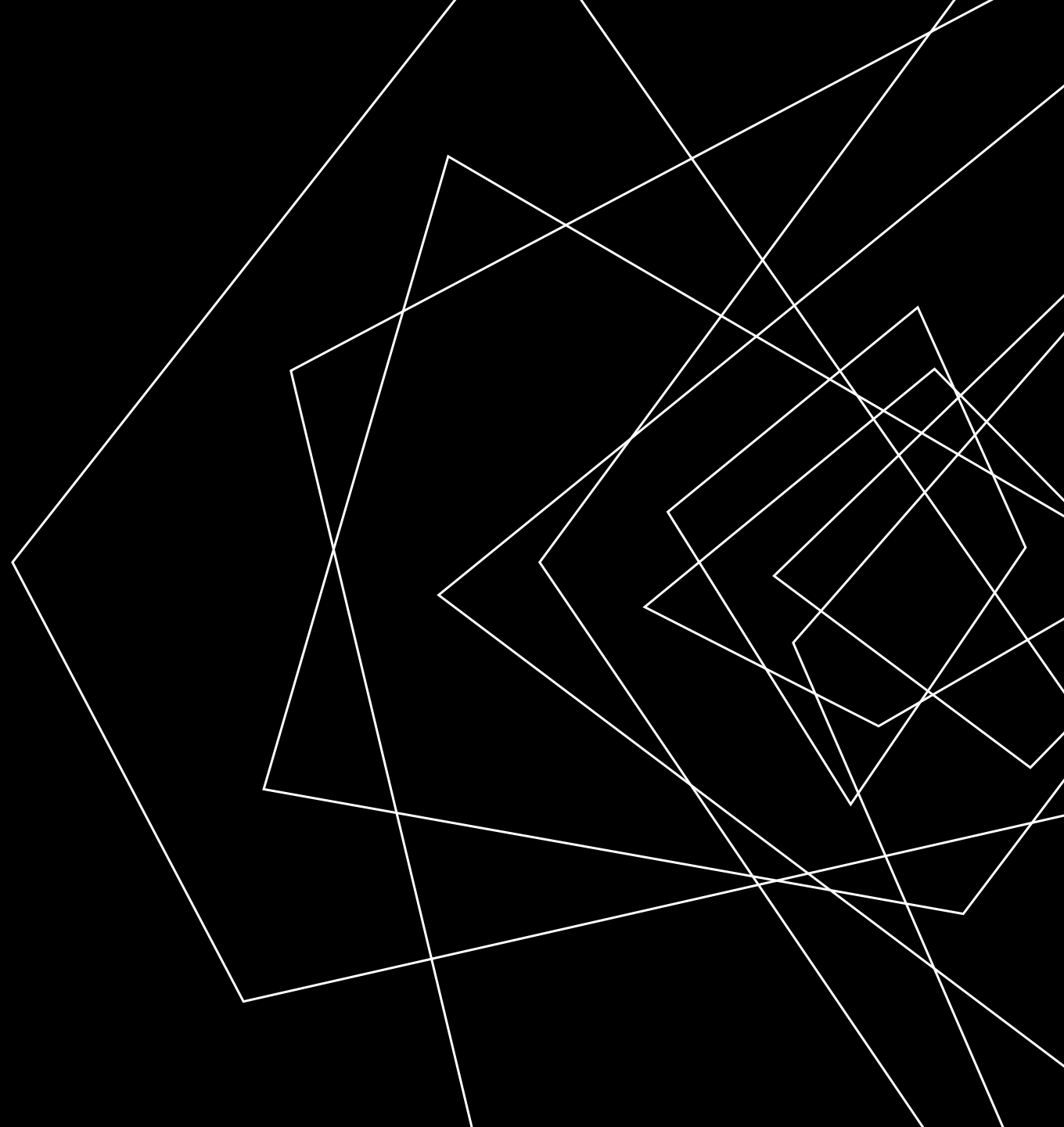
Web programming

THINKING ABOUT DEFENSES

A HISTORY OF SUBVERSION

CAN WE DETECT PROGRAMS WITH OVERFLOW POTENTIAL?

END LECTURE

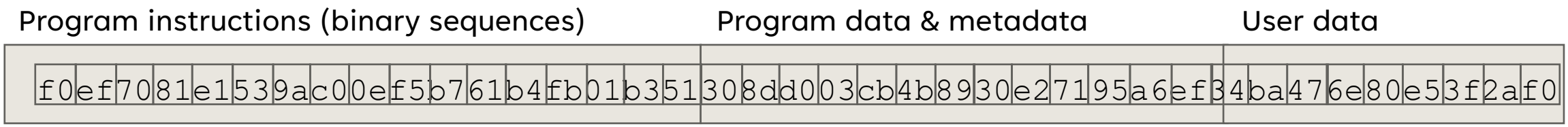


STACK CANARIES

A HISTORY OF SUBVERSION

STACK CANARIES

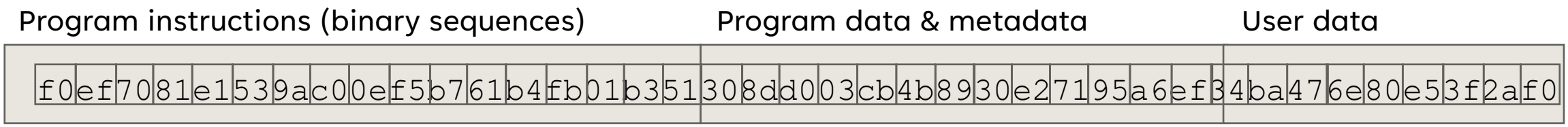
A HISTORY OF SUBVERSION



ASLR
A HISTORY OF SUBVERSION

ASLR

A HISTORY OF SUBVERSION



CFI
A HISTORY OF SUBVERSION