

EXERCISE 6

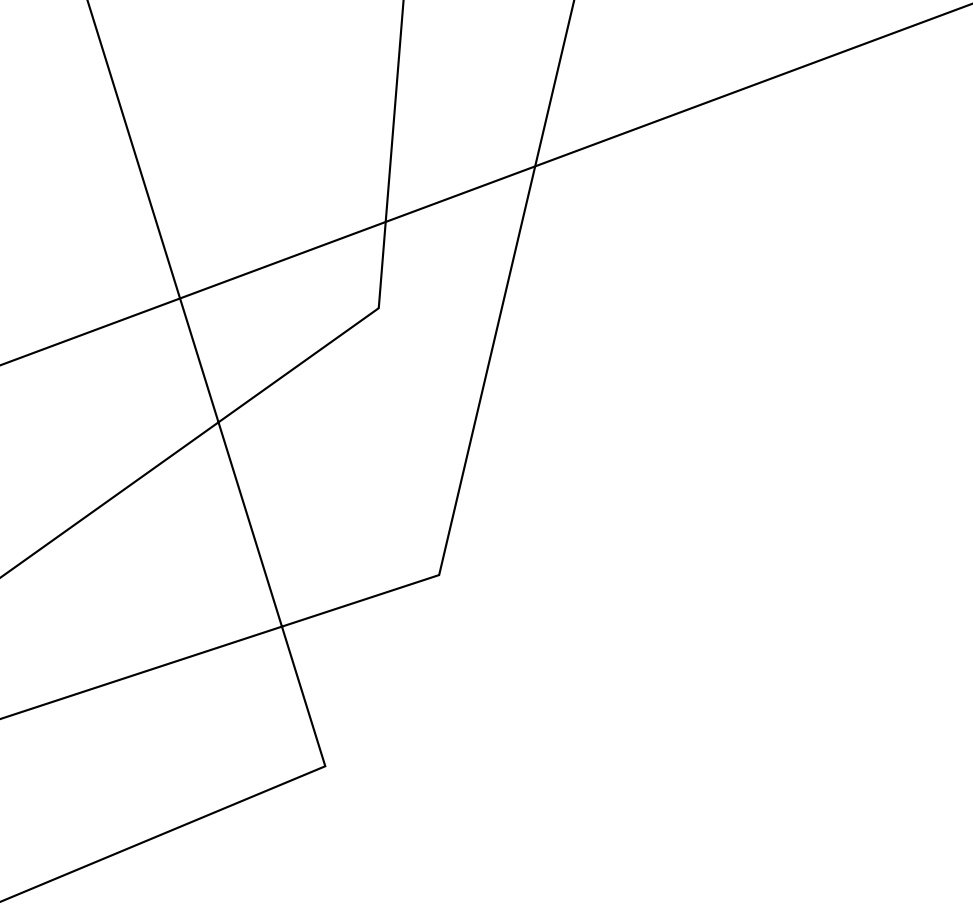
LLVM CALLS REVIEW

Write the LLVM bitcode code corresponding to this function:

```
1 int fn(int * p){  
2     p[7] = 1;  
3     p[0] = 2;  
4 }
```

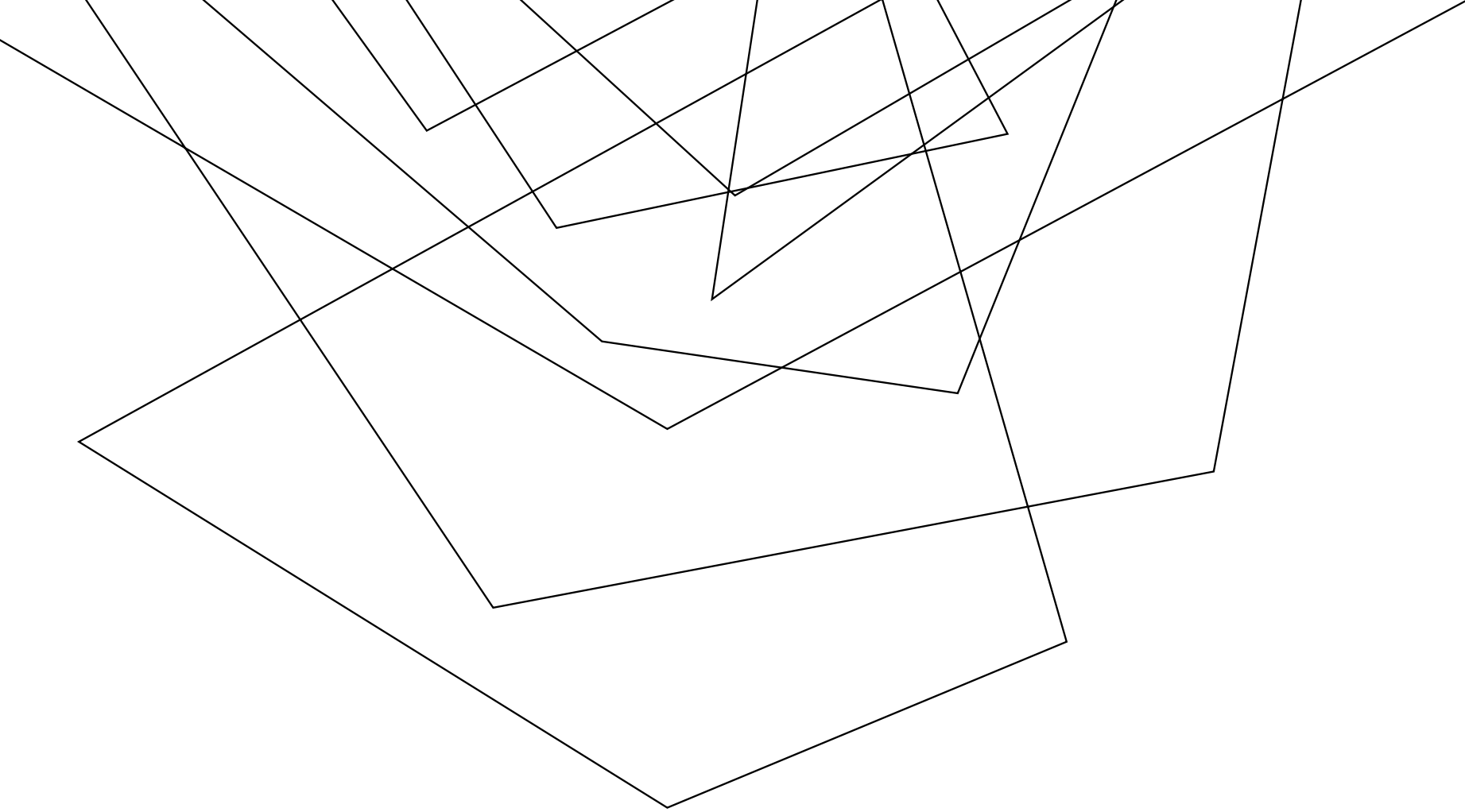
EXERCISE 6 SOLUTION

LLVM CALLS REVIEW

Abstract geometric lines in the top-left corner, consisting of several thin black lines forming a series of overlapping, irregular shapes.

P1 due tonight

**ADMINISTRIVIA
AND
ANNOUNCEMENTS**



MEMORY ATTACKS

EECS 677: Software Security Evaluation

Drew Davidson

LAST TIME: LLVM CALLS

REVIEW: LAST LECTURE

fib.ll abc
 / \
 argv[1] argv[2]

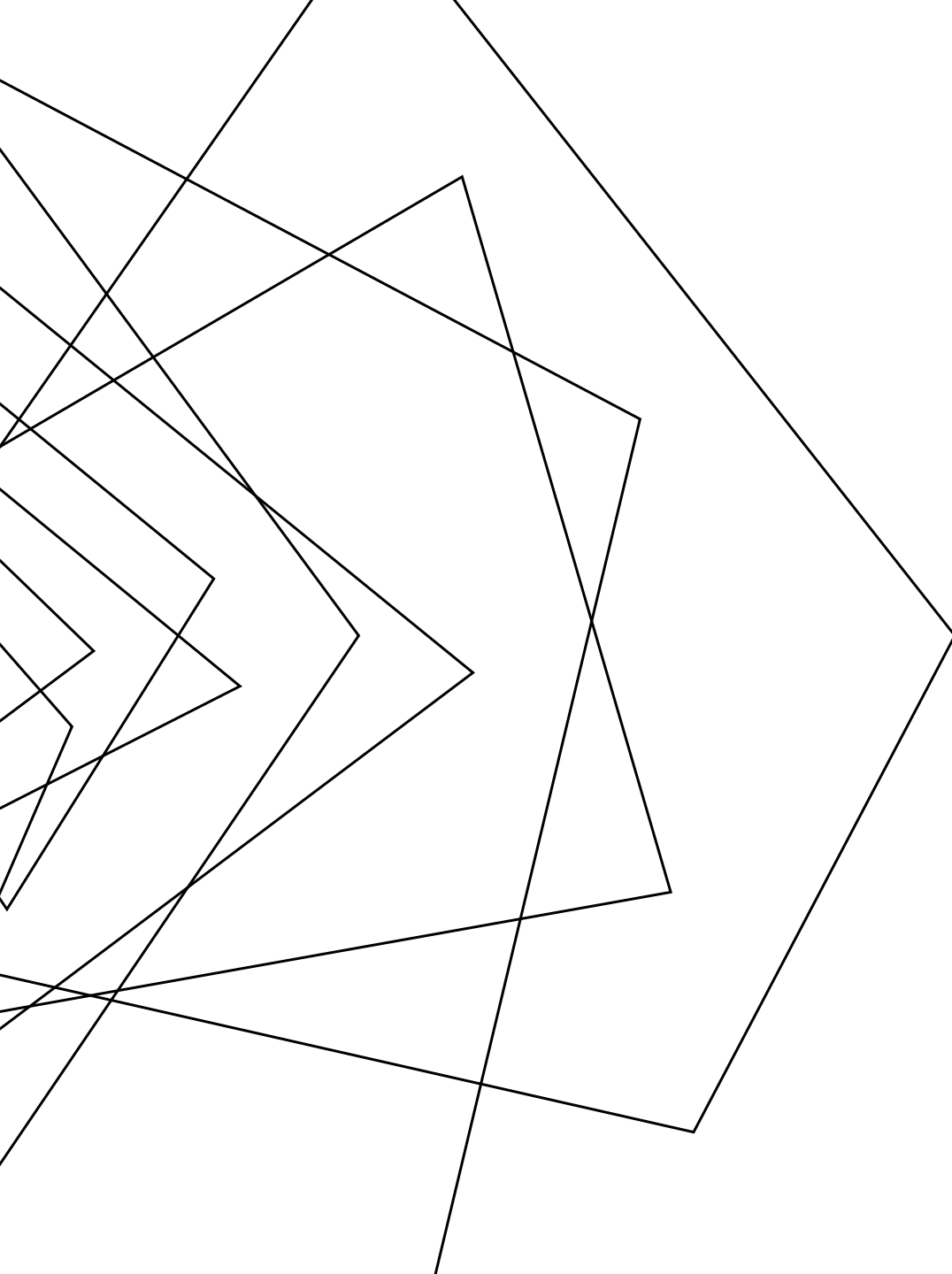
```

int main(int argc, char *argv[]) {
    return 1 + atoi(argv[1]);
}
  
```

```

1 define i32 @main(i32 %argc, i8** %argv) {
2     %idx1 = getelementptr i8*, i8** %argv, i64 1
3     %elt1 = load i8*, i8** %idx1, align 8
4     %res = call i32 @atoi(i8* %elt1)
5     %res2 = add i32 1, %res
6     ret i32 %res2
7 }
8
9 declare i32 @atoi(i8*)
  
```

||: prog.ll 123
 echo \$?
 > 124



CLASS PROGRESS

WE'VE GOT OURSELVES A WAY TO
REPRESENT PROGRAMS

WHAT ARE WE TRYING TO PREVENT?

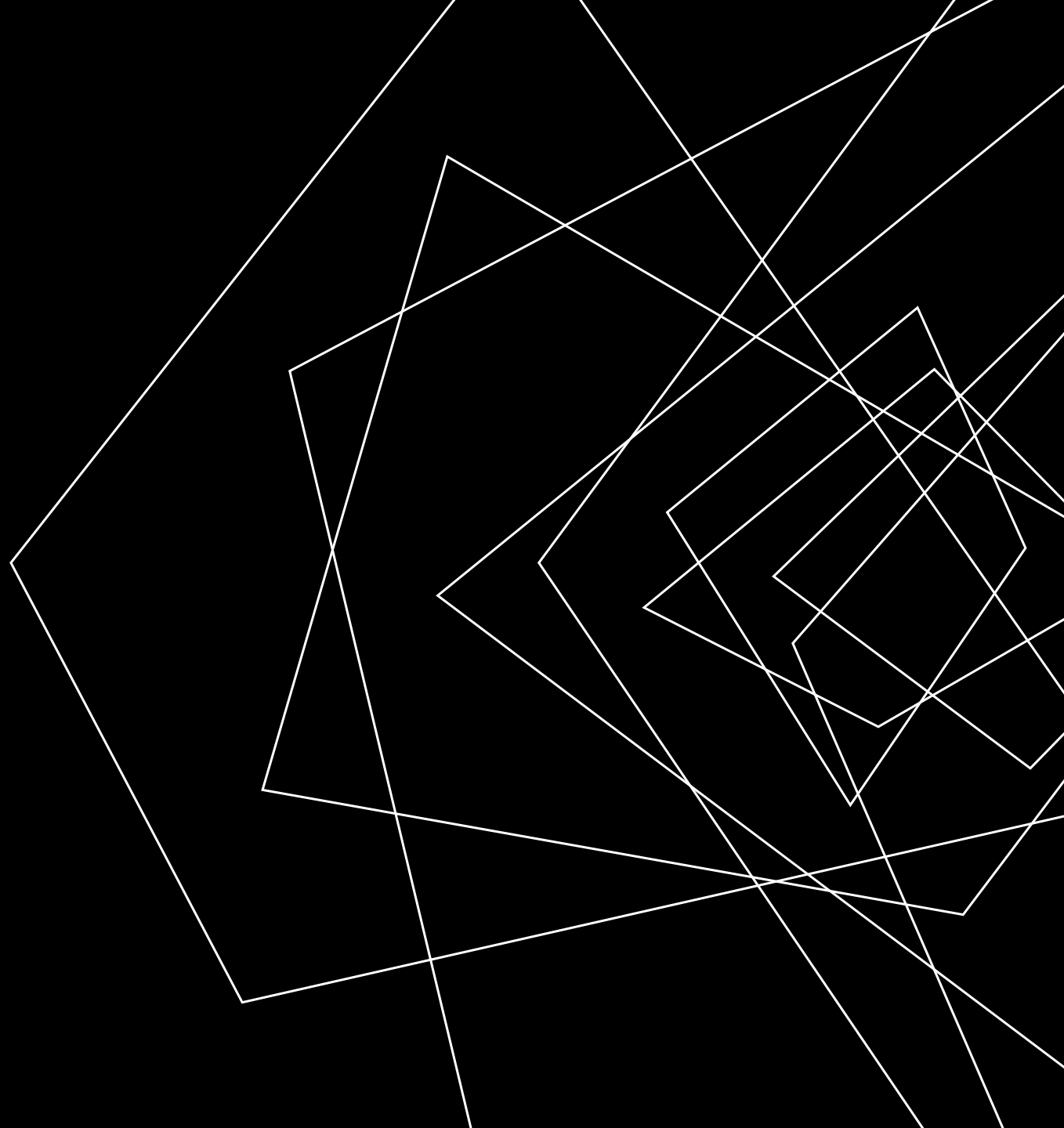
OVERVIEW

PREVENTING BAD STUFF FROM
HAPPENING IN A PROGRAM



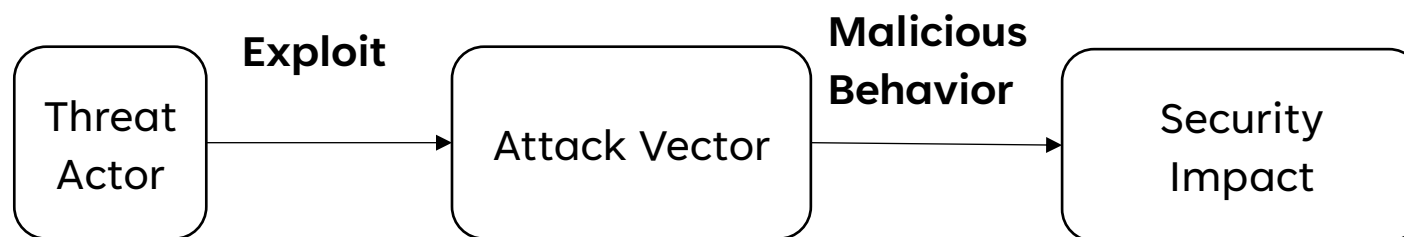
LECTURE OUTLINE

- 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

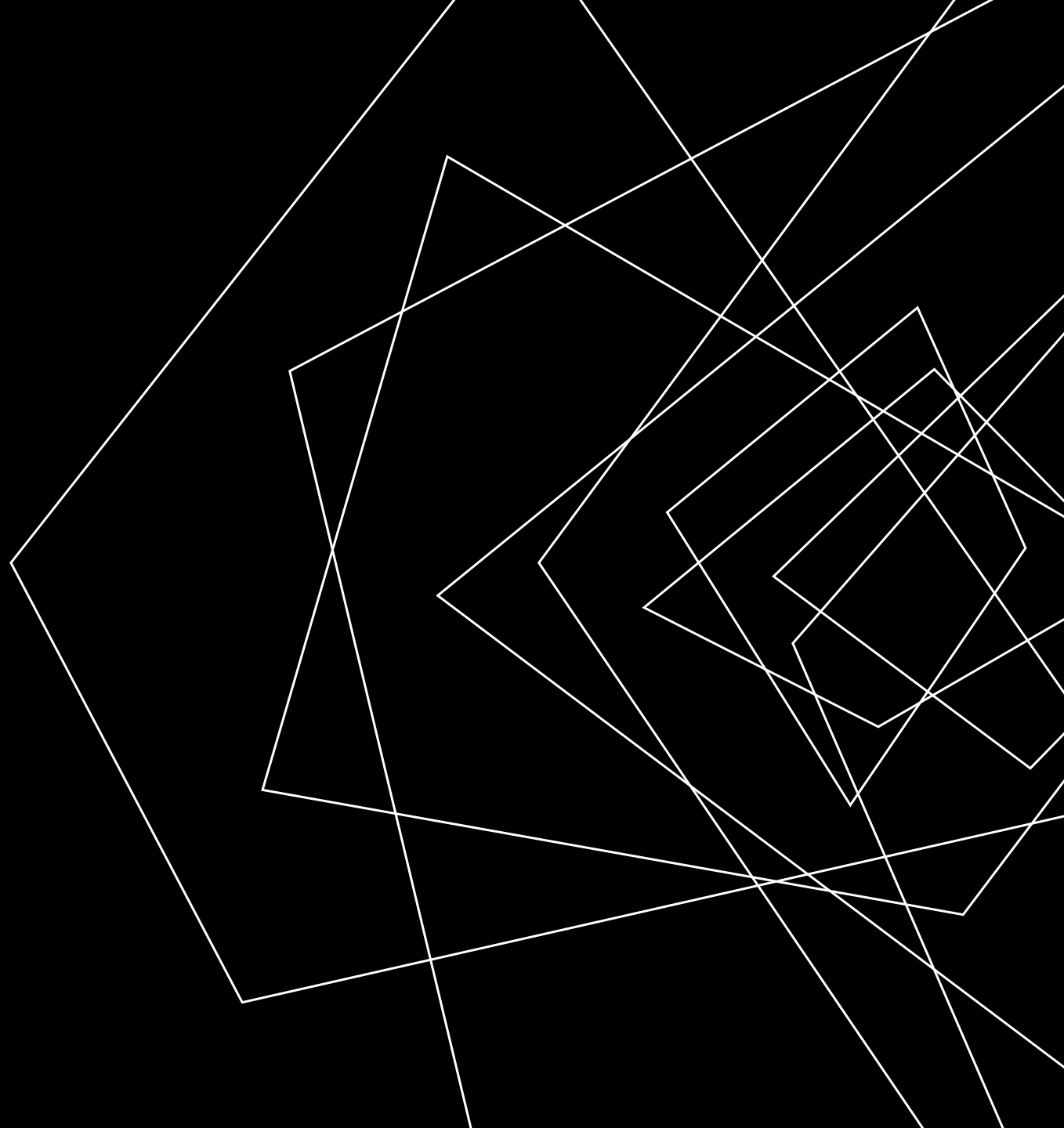
A common threat model

Goal : cause program to act in an unintended
(shut down computer)

Capabilities: interactive access to a target
program

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

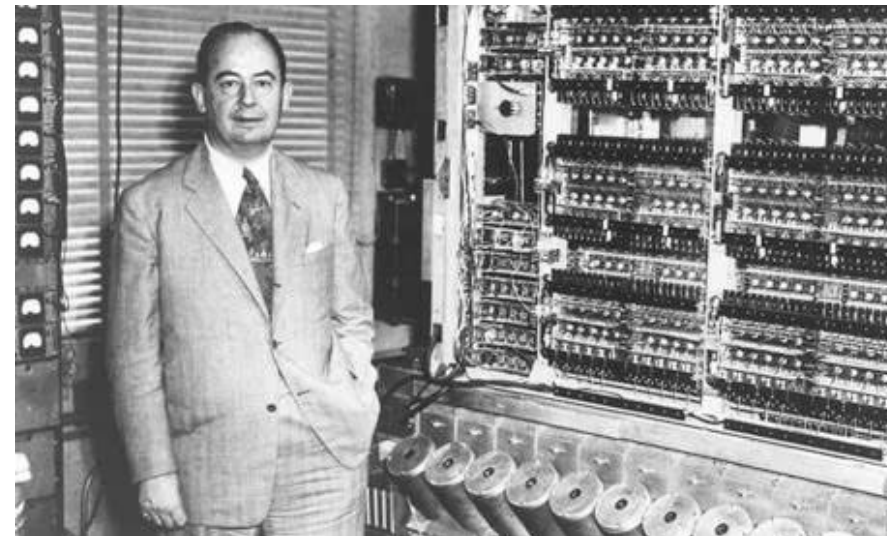
THE VON NEUMANN ARCHITECTURE

Another big idea: Code and data share memory

Good
news!

Programs can write code just
like any other form of data

Bad
news!



Code is data

PROGRAM MEMORY: A 1-D ARRAY

A HISTORY OF SUBVERSION

Code

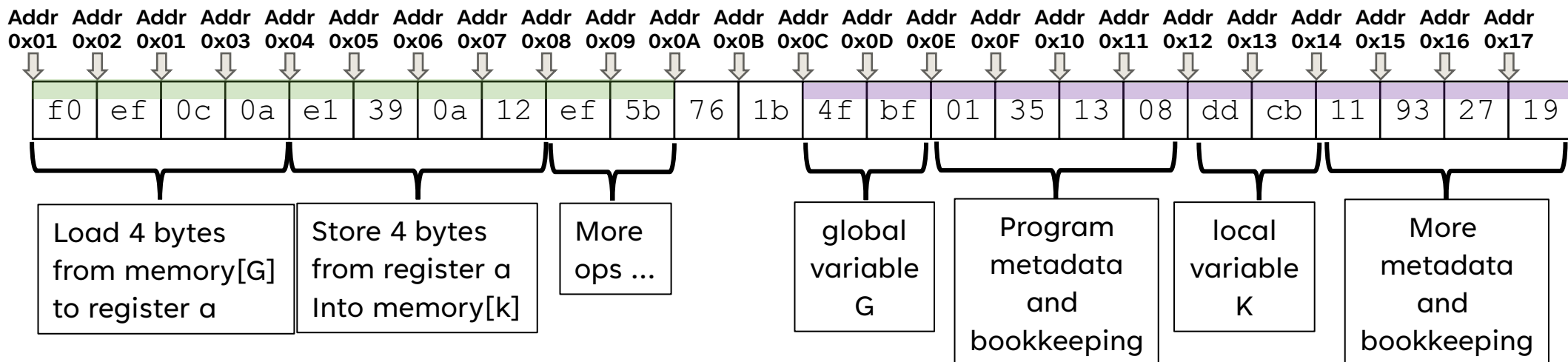
Load 2 bytes memory[G] into register a
Store register a into 4 bytes memory[K]
More ops ...

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

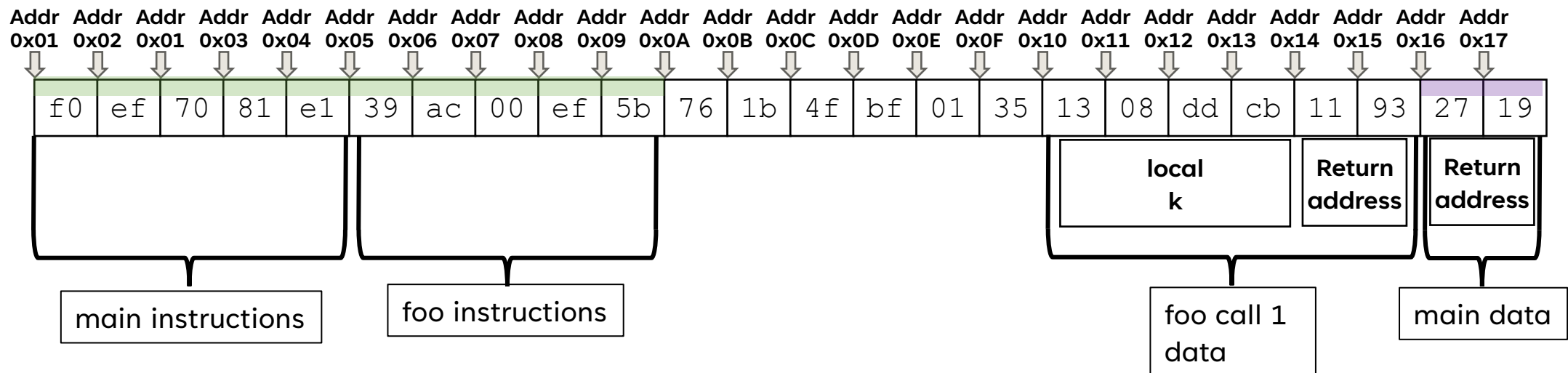
All the code in function main()
All the code in function foo()

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

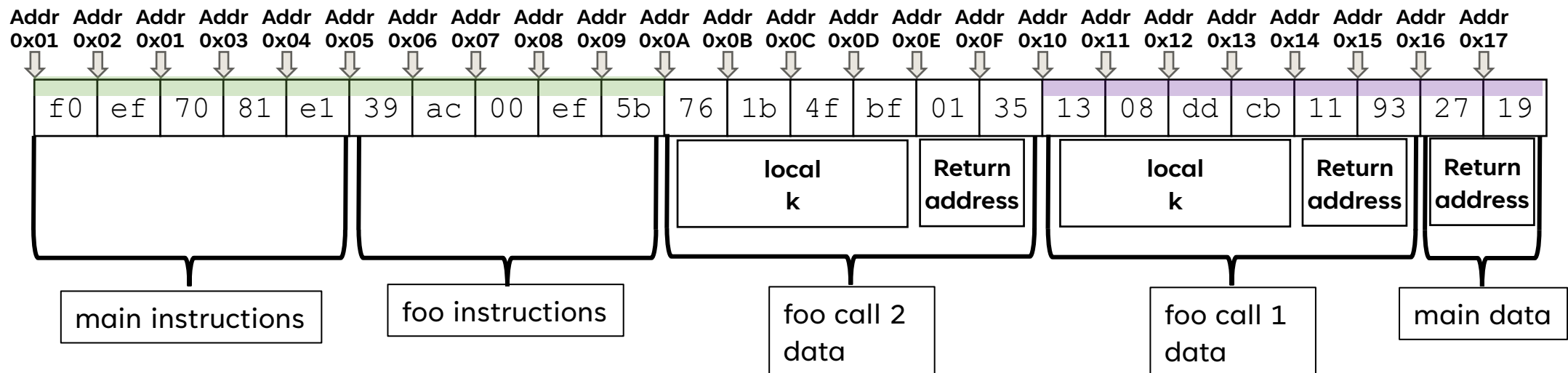
All the code in function main()
All the code in function foo()

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

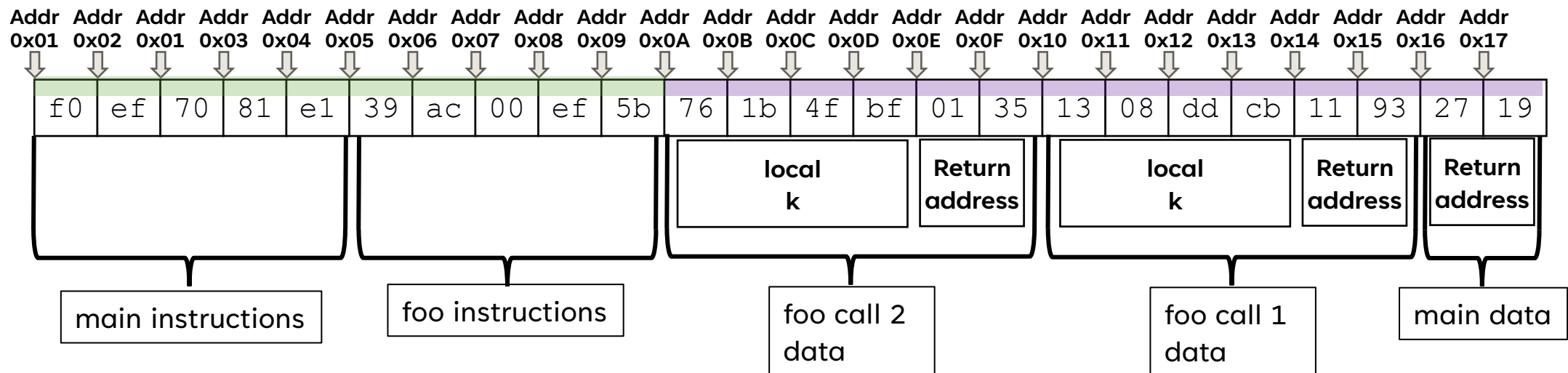
All the code in function main()
All the code in function foo()

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

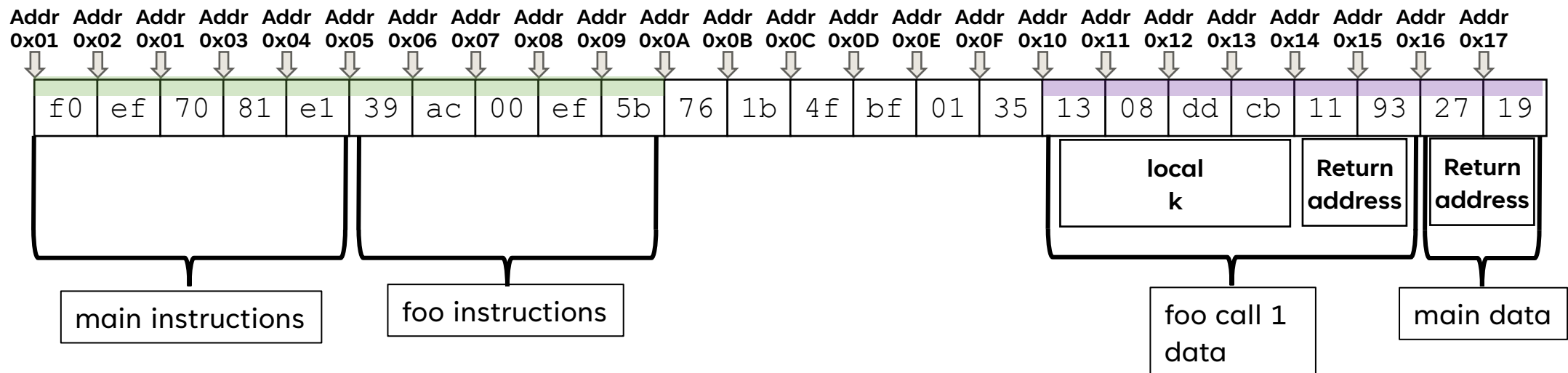
All the code in function main()
All the code in function foo()

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

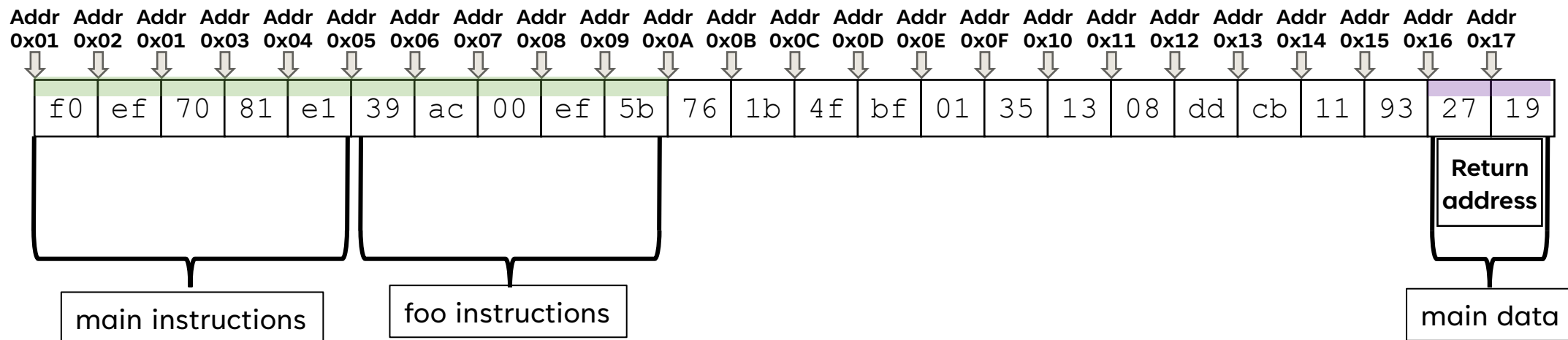
All the code in function main()
All the code in function foo()

Data

Global variable G
foo's local variable K

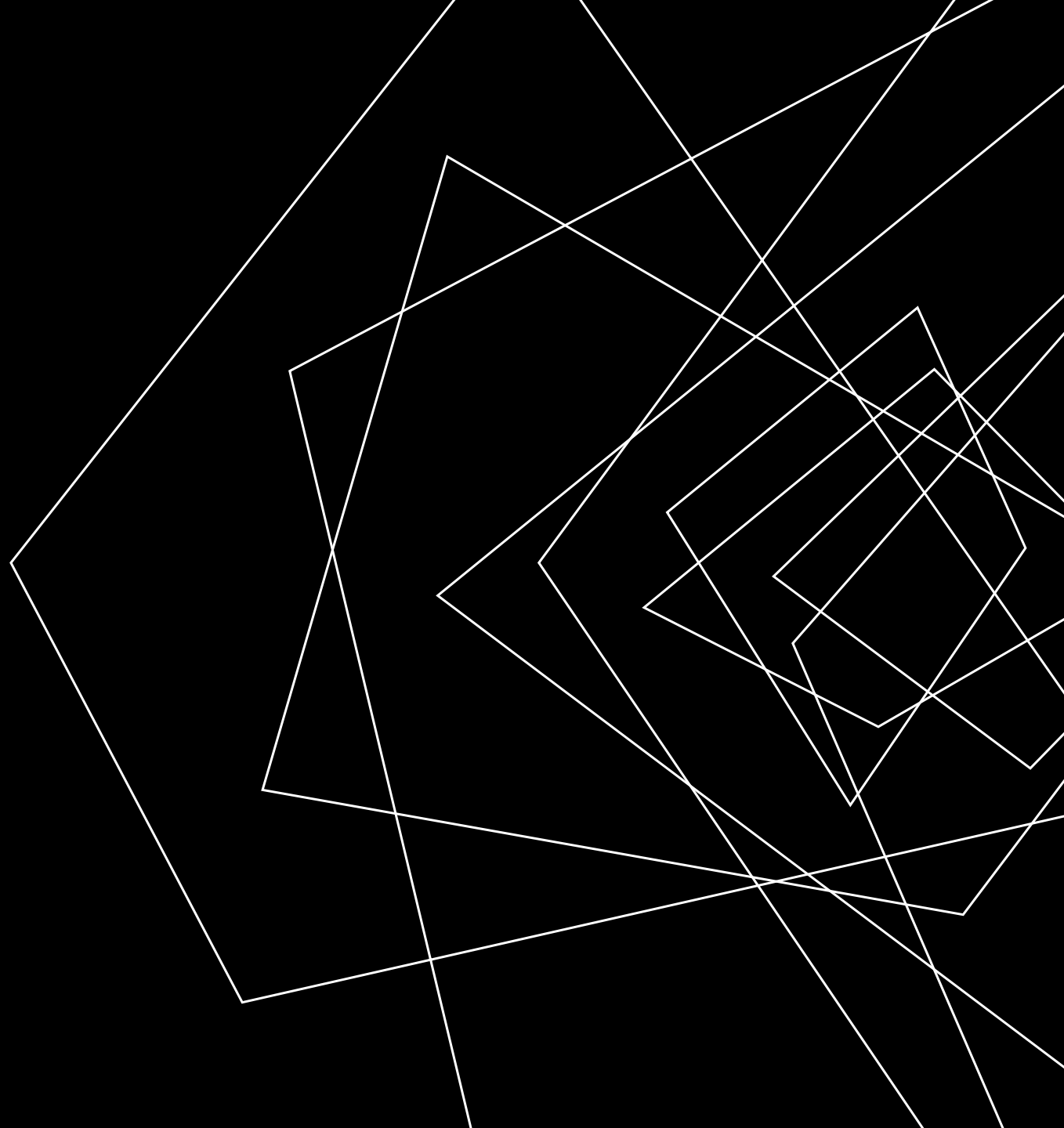
Program instructions (binary sequences)

Program data & metadata (binary sequences)



LECTURE OUTLINE

- Terms/Concepts
- Memory Layout
- Memory Attacks



BREAKING BOUNDS

A HISTORY OF SUBVERSION

(ascii hex value 0x44)

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

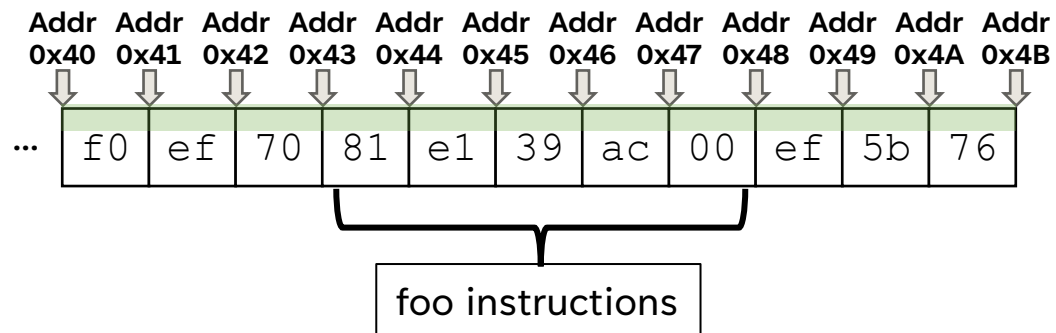
D

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

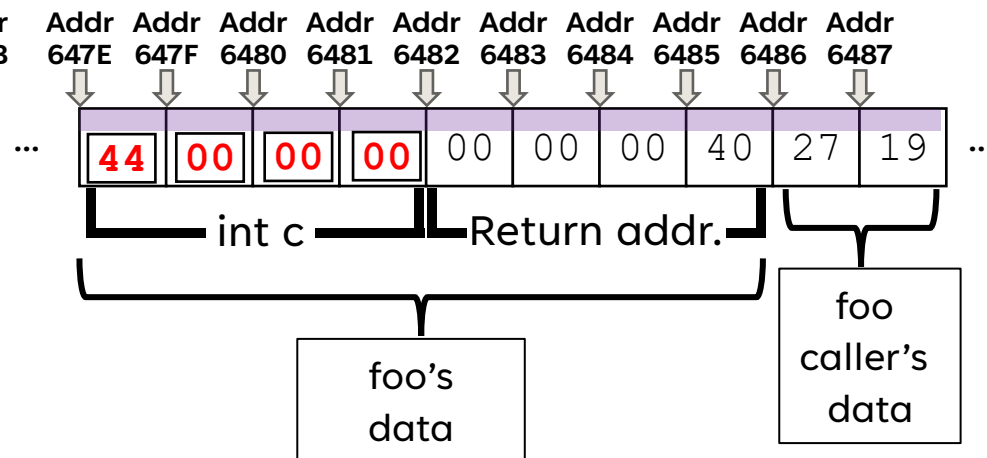
define void @foo() {
  %stdInPtr = load %struct._IO_FILE*, %struct._IO_FILE** @stdin
  %char = call i32 @getc(%struct._IO_FILE* %stdInPtr)
  %charInt = sext i32 %char to i64
  %charPtr = inttoptr i64 %charInt to i32*
  store i32 2, i32* %charPtr
  ret void
}

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

Program instructions



Program (meta)data



BREAKING BOUNDS

A HISTORY OF SUBVERSION

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

(ascii hex value 0x44)

D

Pointer to address 0x44

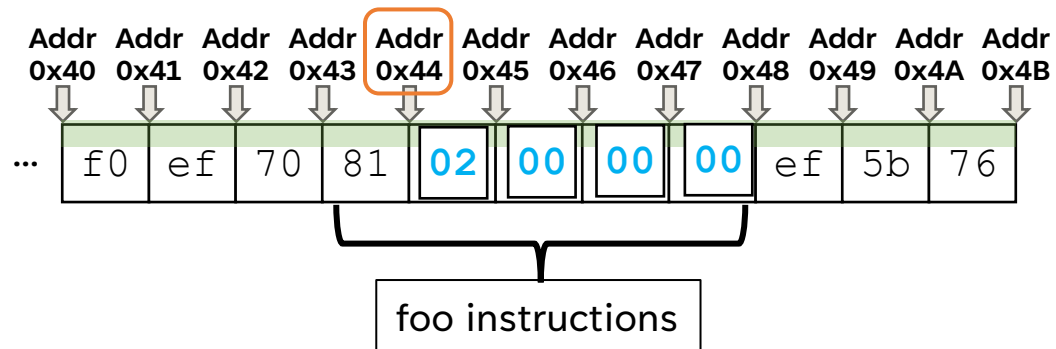
4-byte value 0x00000002

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

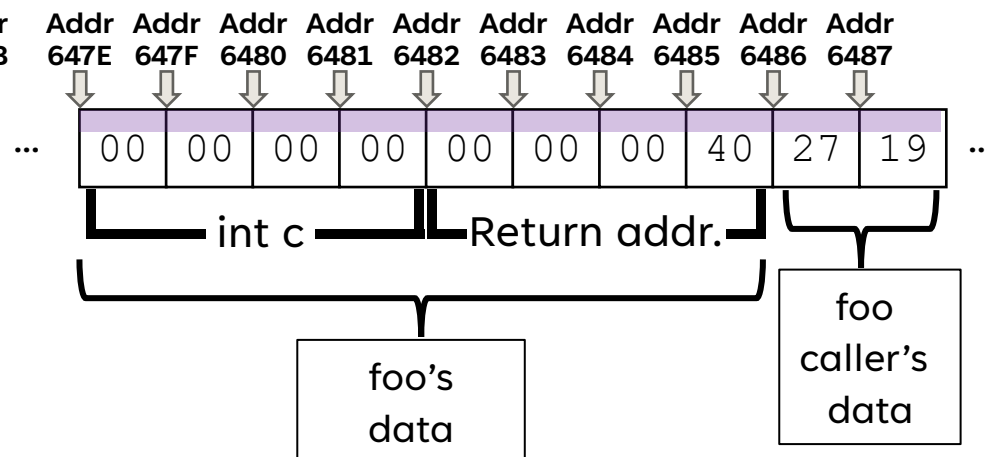
define void @foo() {
  %stdInPtr = load %struct._IO_FILE*, %struct._IO_FILE** @stdin
  %char = call i32 @getc(%struct._IO_FILE* %stdInPtr)
  %charInt = sext i32 %char to i64
  %charPtr = inttoptr i64 %charInt to i32*
  store i32 2, i32* %charPtr
  ret void
}

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

Program instructions



Program (meta)data



BREAKING BOUNDS

A HISTORY OF SUBVERSION

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

(ascii hex value 0x44)

D

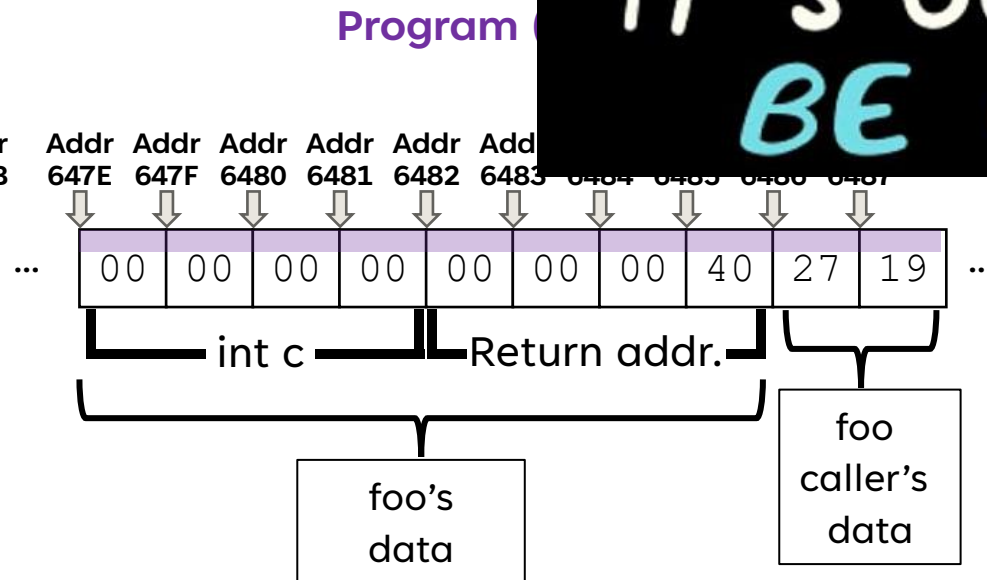
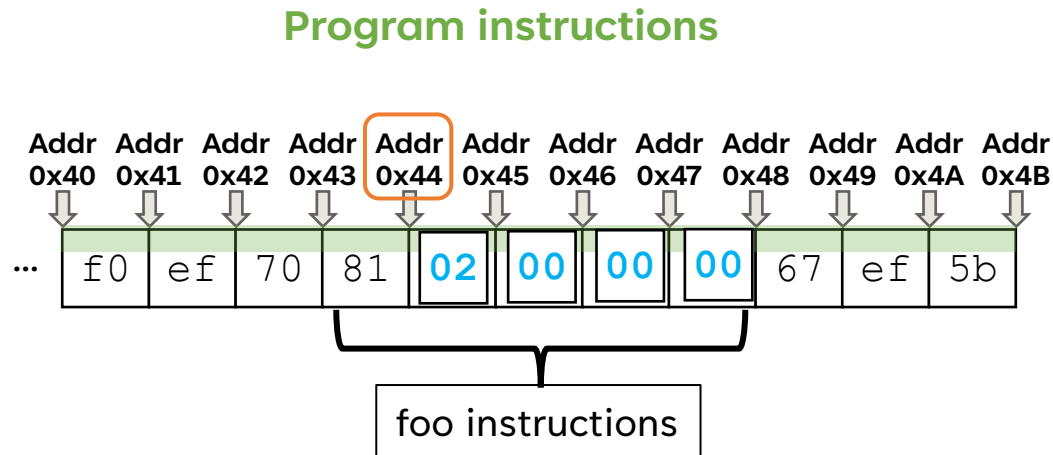
Pointer to address 0x44

4-byte value 0x00000002

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

define void @foo() {
  %stdInPtr = load %struct._IO_FILE, @stdin
  %char = call i32 @getc(%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*)
```



BUFFER OVERFLOWS

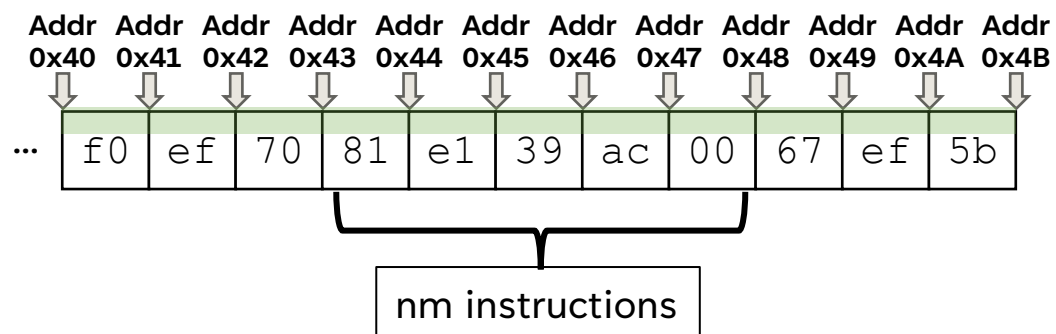
A HISTORY OF SUBVERSION

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

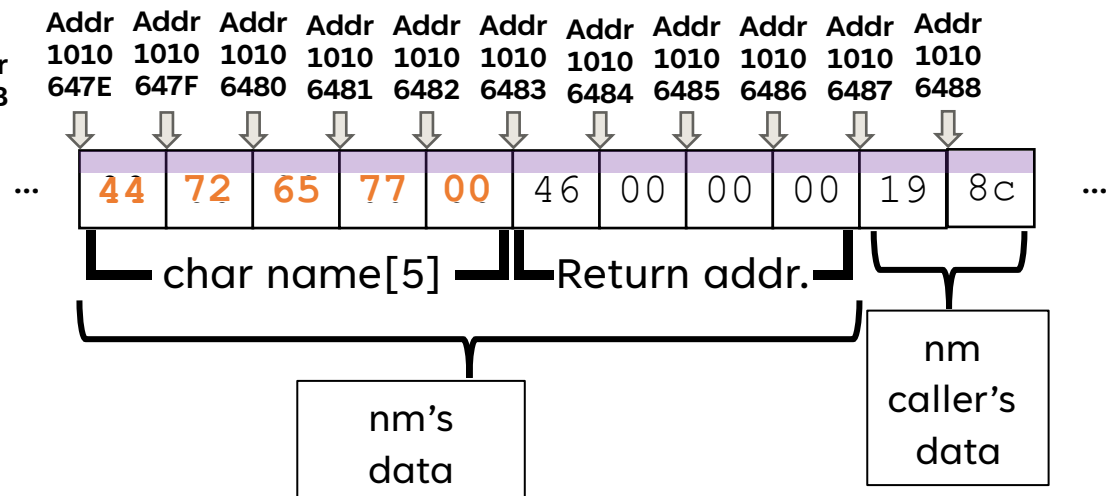
D r e W (end)

0x44 0x72 0x65 0x77 0x00

Program instructions



Program (meta)data



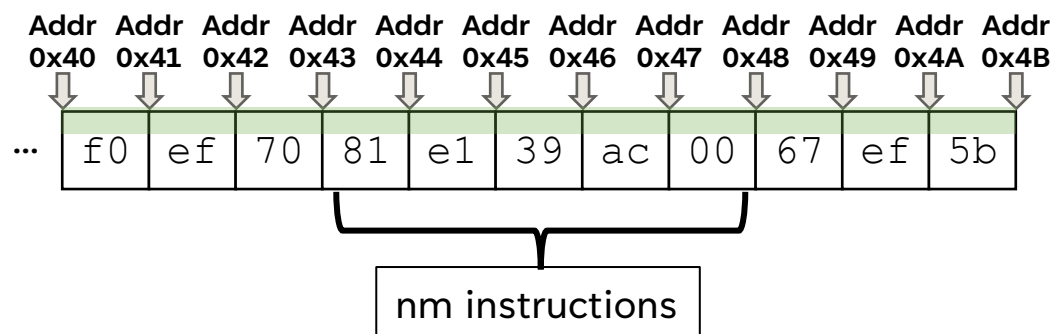
BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

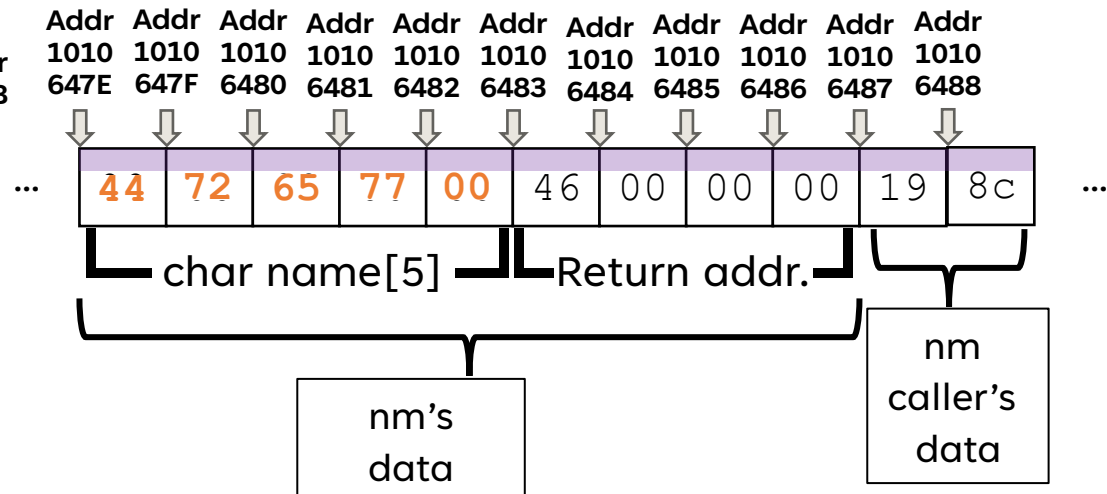
```
int nm() {
    char name[5];
    printf("Enter your name: ");
    gets(name);
    → printf("hi %s\n", name);
    return 0;
}
```

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

Program instructions



Program (meta)data



BUFFER OVERFLOWS

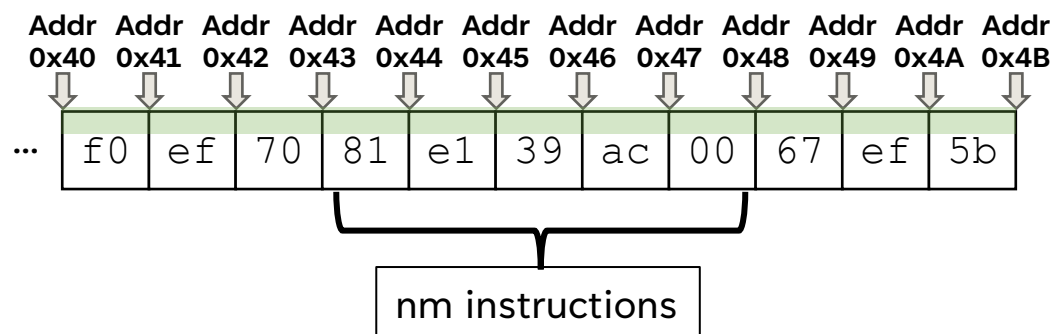
A HISTORY OF SUBVERSION

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

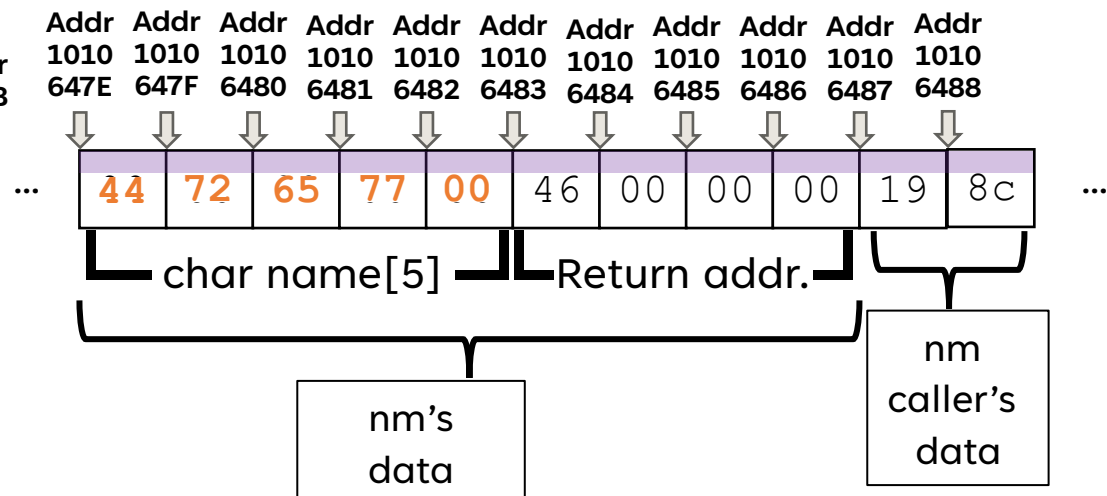
D r e W (end)
 0x44 0x72 0x65 0x77 0x00

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

Program instructions



Program (meta)data



BUFFER OVERFLOWS

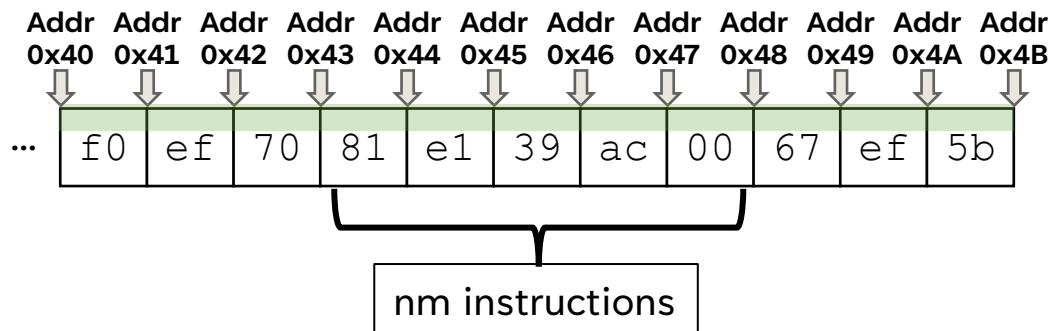
A HISTORY OF SUBVERSION

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

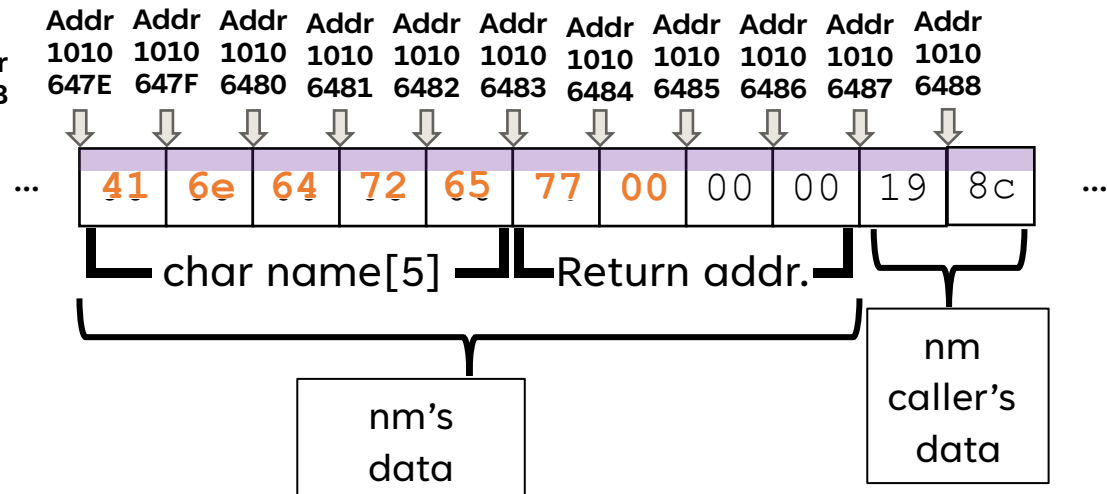
A n d r e w (end)

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

Program instructions



Program (meta)data



BUFFER OVERFLOWS

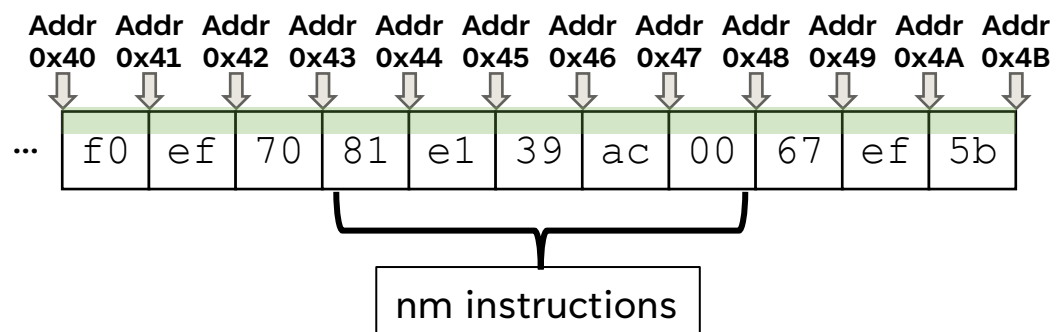
A HISTORY OF SUBVERSION

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

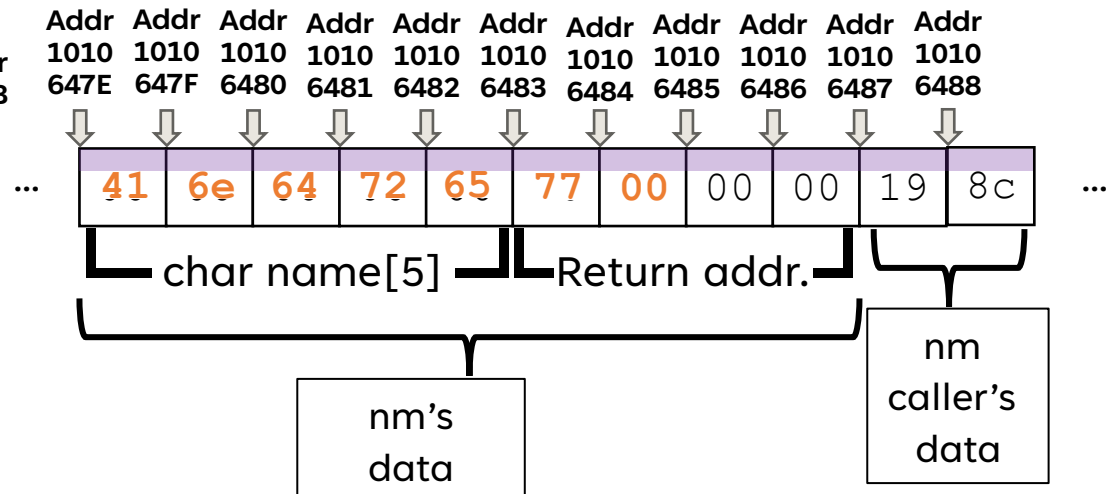
A n d r e w (end)

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

Program instructions



Program (meta)data



BUFFER OVERFLOWS

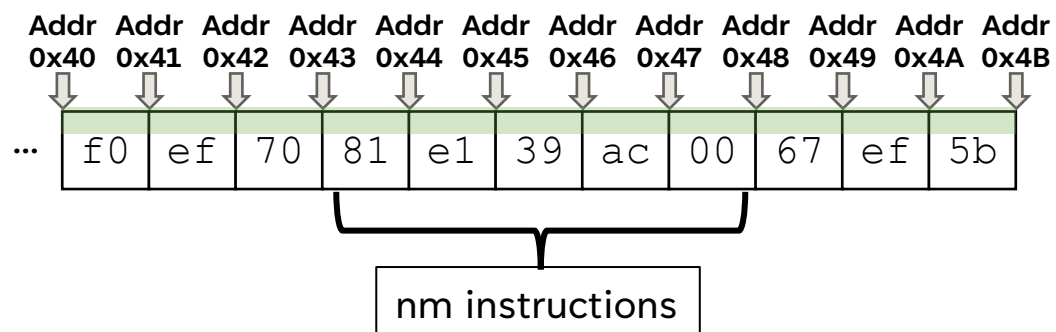
A HISTORY OF SUBVERSION

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

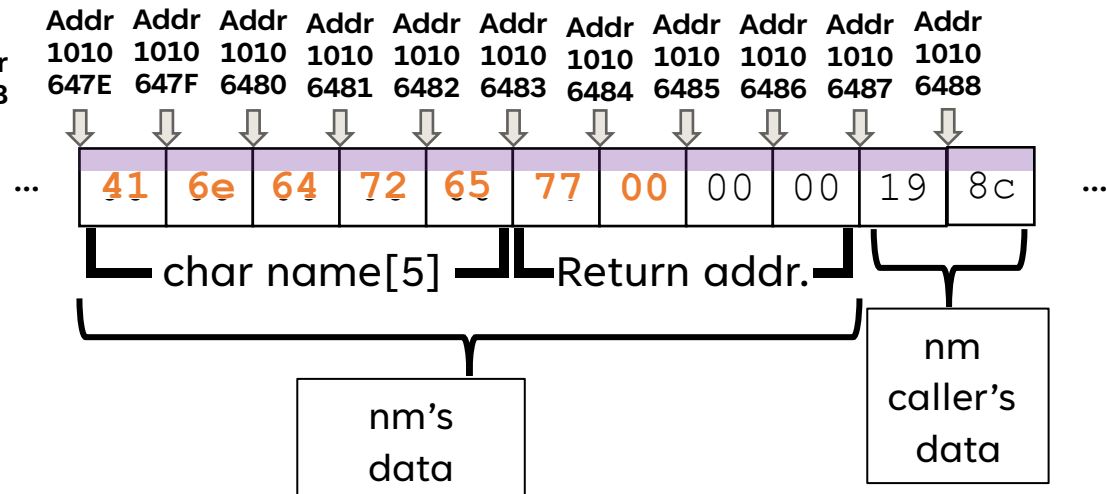
A n d r e w (end)

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

Program instructions



Program (meta)data



BUFFER OVERFLOWS

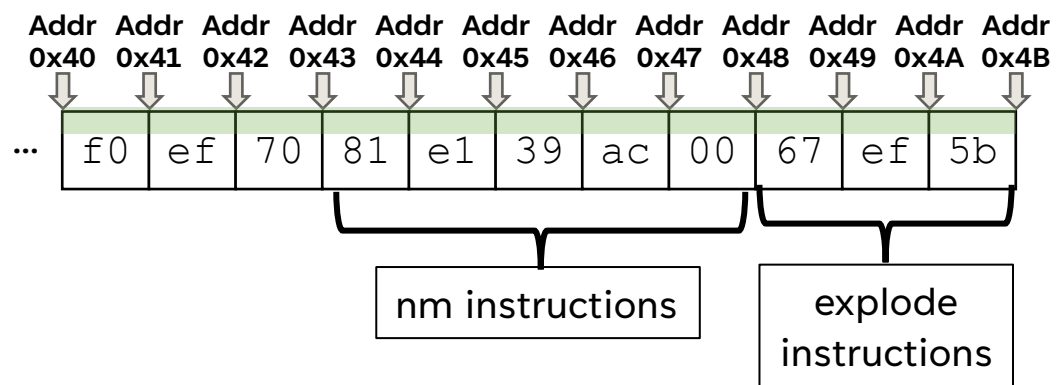
A HISTORY OF SUBVERSION

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

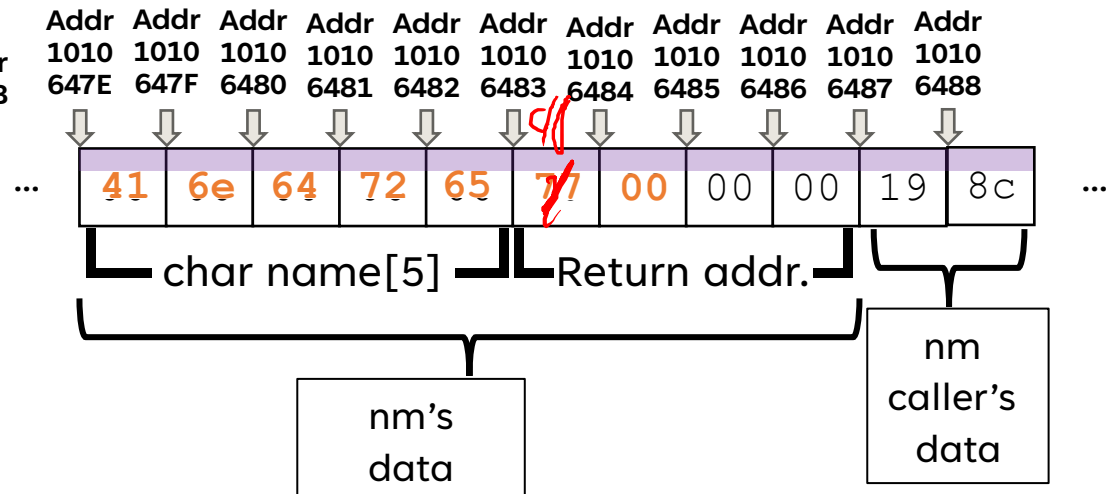
void explode() {
    ... //destroyTheComputer
}
```

A n d r e ~~W~~ (end)
 0x41 0x6e 0x64 0x72 0x65 ~~0x77~~ 0x00
 0x4f

Program instructions



Program (meta)data



BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

```

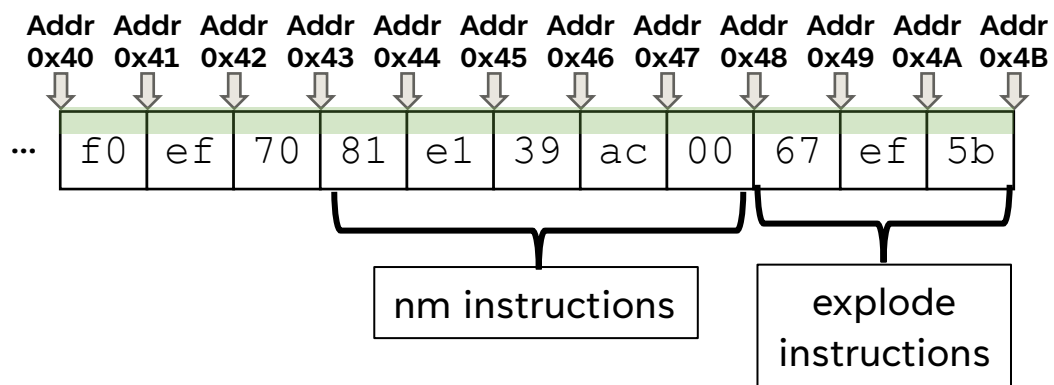
int nm() {
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf("hi %s\n", name);
    → return 0;
}

void explode() {
    ... //destroyTheComputer
}

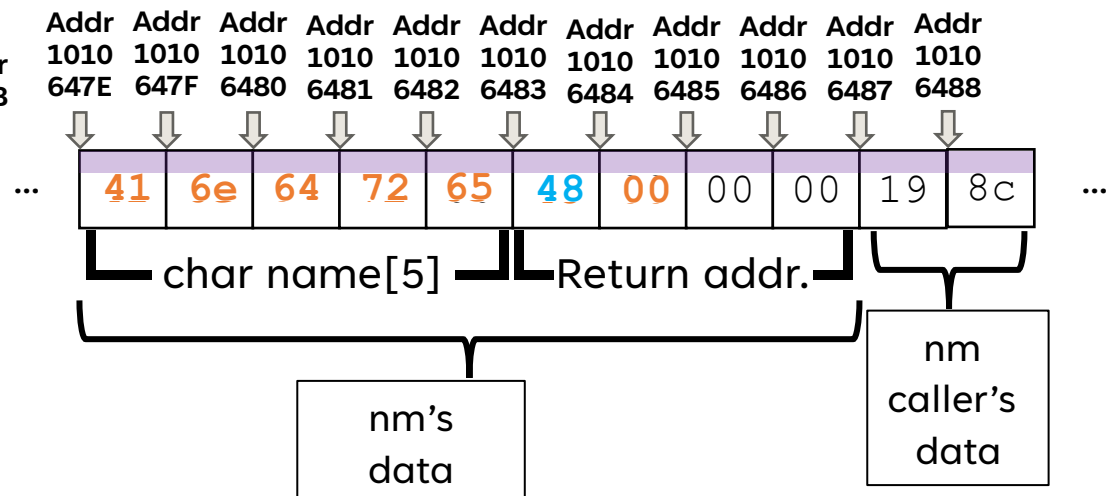
```

A n d r e ~~W~~ (end)
 0x41 0x6e 0x64 0x72 0x65 ~~0x77~~ 0x00
 0x48

Program instructions



Program (meta)data

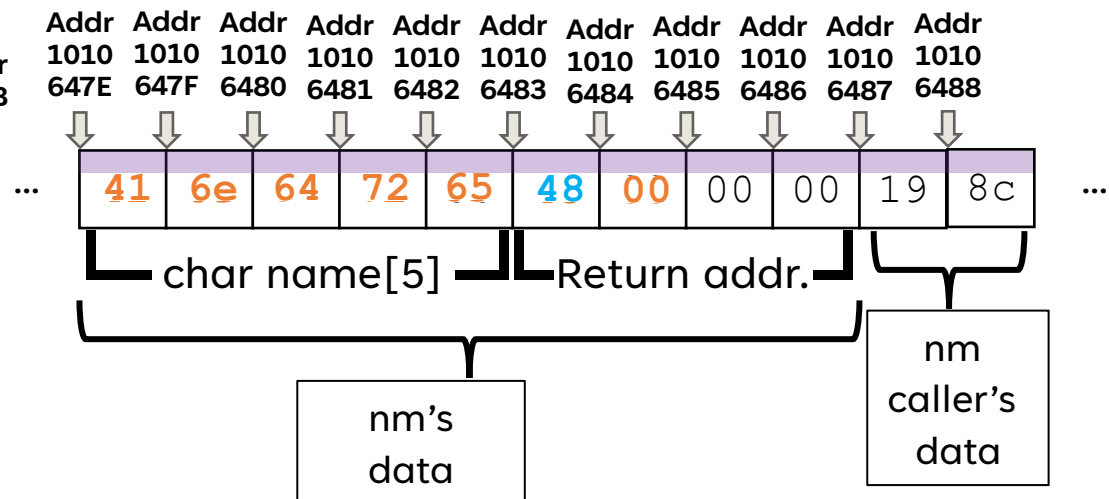


A HISTORY OF SUBVERSION

```
➡ void explode() {
    ... //destroyTheComputer
}
```

H

Program (meta)data



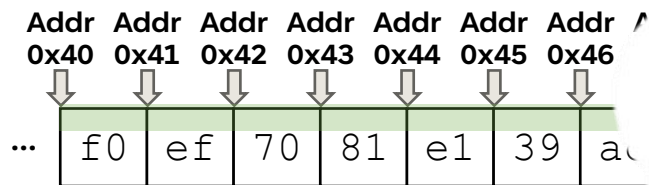
BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

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

→ void explode() {
 ... //destroyTheComputer
}

Program instructions



nm instructions

explode
instructions

nm's
data

nm
caller's
data

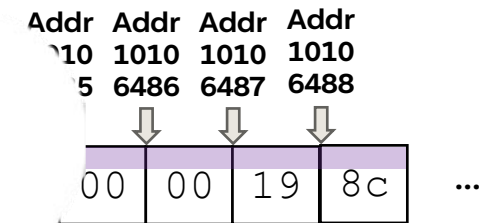
IS IT TIME



TO PANIC ?

H
 e W (end)
 0x65 0x77 0x00
 0x48

meta) data



addr.

BUFFER OVERFLOWS: IMPACT

A HISTORY OF SUBVERSION

explodeTheComputer isn't going to be in most programs
i.e. no good program ever includes 67 followed by ef 5b



Credit: KC Green

SCRIPT INJECTION

A HISTORY OF SUBVERSION

explodeTheComputer isn't going to be in most programs
i.e. no good program ever includes 67 followed by ef 5b

Can we inject our own payload?

- 1) Put 67 then ef 5b into memory
- 2) Jump to that memory location

SCRIPT INJECTION

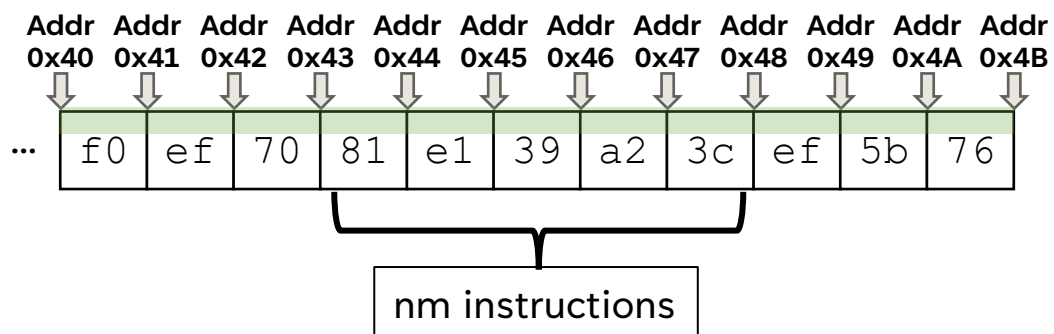
A HISTORY OF SUBVERSION

- 1) Put 67 then ef 5b into memory
- 2) Jump to that memory location

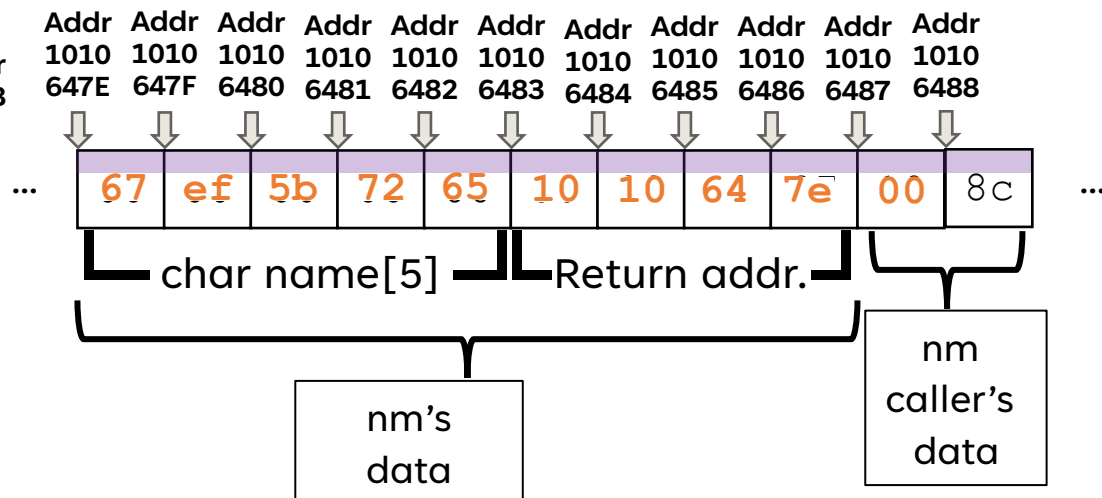
```
int nm() {
    char name[5];
    printf("Enter your name: ");
    → gets(name);
    printf("hi %s\n", name);
    return 0;
}
```

g i [r e LF LF sp B (end)
 0x67 0xef 0x5b 0x72 0x65 0x10 0x10 0x64 0x7e 0x00

Program instructions



Program (meta)data



SCRIPT INJECTION

A HISTORY OF SUBVERSION

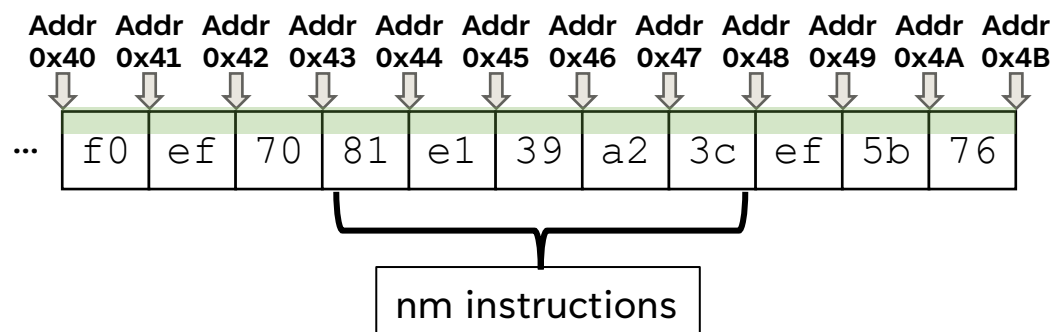
```
int nm() {
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf("hi %s\n", name);
    → return 0;
}
```

- 1) Put 67 then ef 5b into memory
- 2) Jump to that memory location

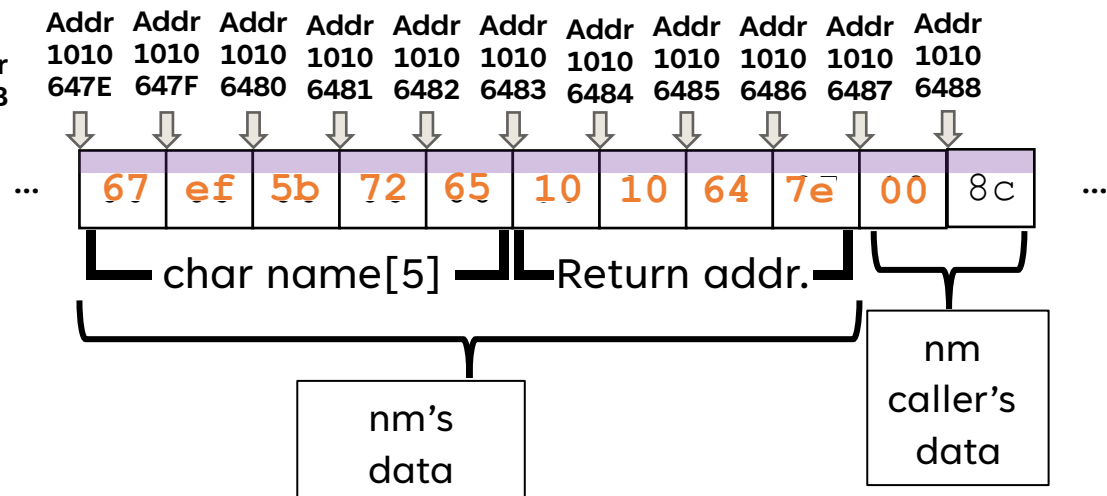
g	i	[	r	e	LF	LF	sp	B	(end)
0x67	0xef	0x5b	0x72	0x65	0x10	0x10	0x64	0x7e	0x00

*jump to the "return" address
(which we overwrote with this
address) ...*

Program instructions



Program (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;
}
```

- 1) Put 67 then ef 5b into memory
- 2) Jump to that memory location

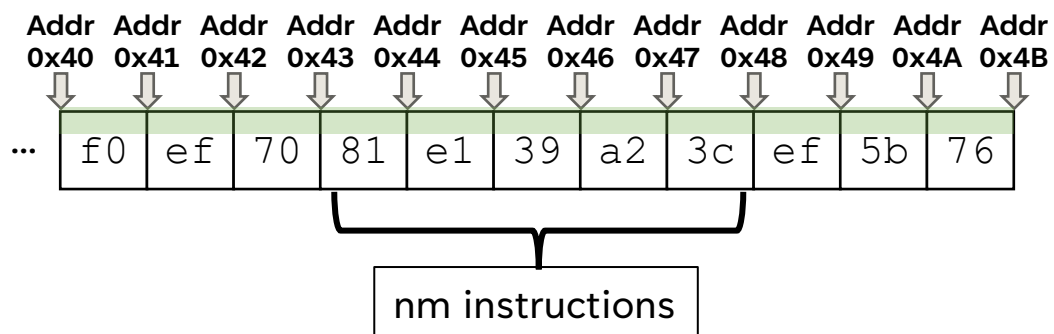
g i [r e
0x67 0xef 0x5b 0x72 0x65

*... start executing the bytes
we put in memory!*

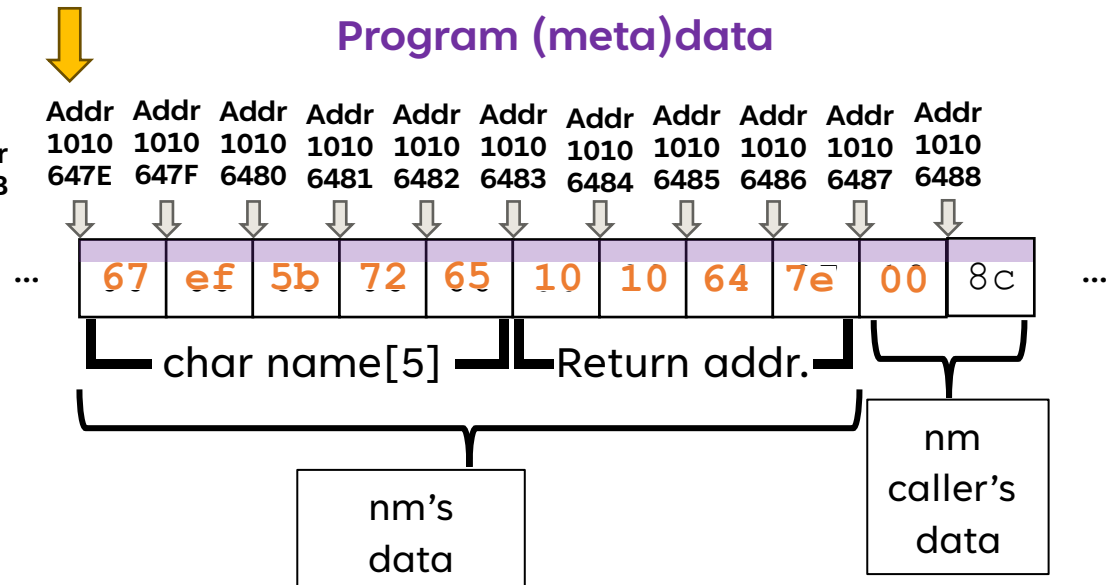
LF LF sp B (end)
0x10 0x10 0x64 0x7e 0x00

*jump to the "return" address
(which we overwrote with this
address) ...*

Program instructions



Program (meta)data



SCRIPT INJECTION

A HISTORY OF SUBVERSION

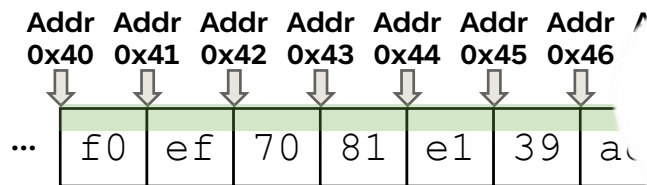
```
int nm() {
    char name[5];
    printf("Enter\n");
    gets(name);
    printf("hi %s\n");
    return 0;
}
```

IS IT TIME



TO PANIC ?

Program instructions



nm instructions

LF LF d ~ (end)
0x10 0x10 0x64 0x7e 0x00

these

meta)data

Addr	Addr	Addr	Addr
1010	1010	1010	1010
5	6486	6487	6488

64	7e	00	8c
----	----	----	----

addr.

nm's
data

nm
caller's
data

SCRIPT INJECTION: IMPACT

A HISTORY OF SUBVERSION

explodeTheComputer isn't going to be in most programs
i.e. no good program ever includes 67 followed by ef 5b

Can we inject our own payload?

- 1) Put 67 then ef 5b into memory
- 2) Jump to that memory location



Credit: KC Green

SCRIPT INJECTION: THE FIX

A HISTORY OF SUBVERSION

explodeTheComputer isn't going to be in most programs
i.e. no good program ever includes 67 followed by ef 5b

Can we inject our own payload?

- 1) Put 67 then ef 5b into memory
- 2) Jump to that memory location

Don't allow execution of user-supplied 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;
}
```

1) Put 67 then ef 5b into memory

2) Jump to that memory location

g i [r e
0x67 0xef 0x5b 0x72 0x65

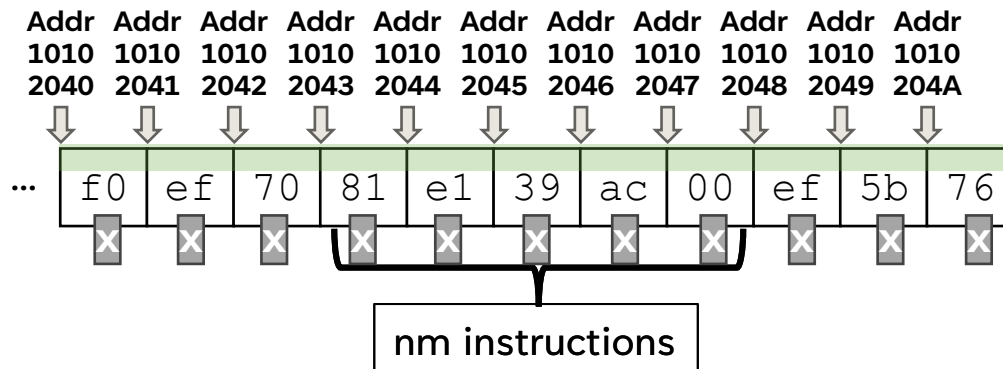
*... start executing the bytes
we put in memory!*

LF LF sp B (end)
0x10 0x10 0x64 0x7e 0x00

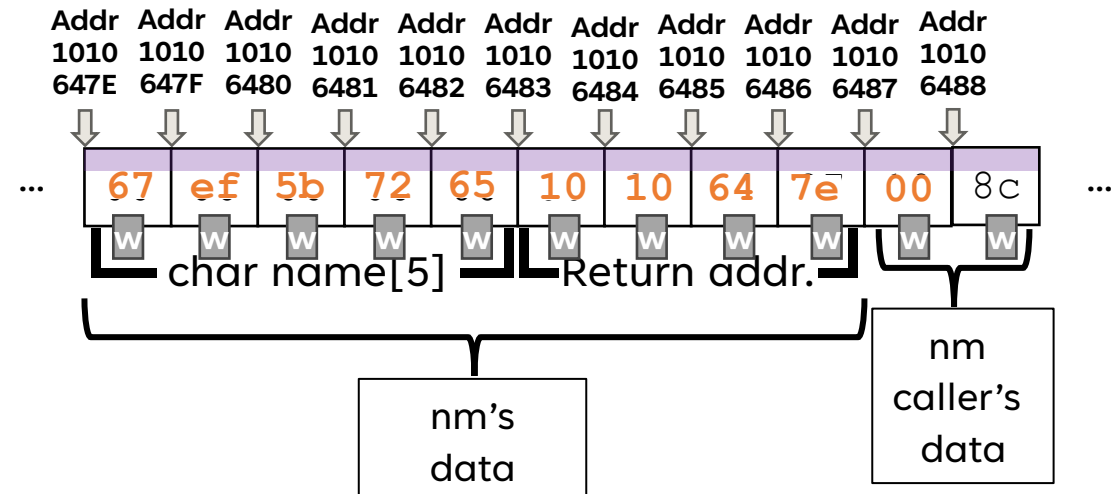
*jump to the "return" address
(which we overwrote with this
address) ...*

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;
}
```

1) Put 67 then ef 5b into memory

2) Jump to that memory location

g i [r e
0x67 0xef 0x5b 0x72 0x65

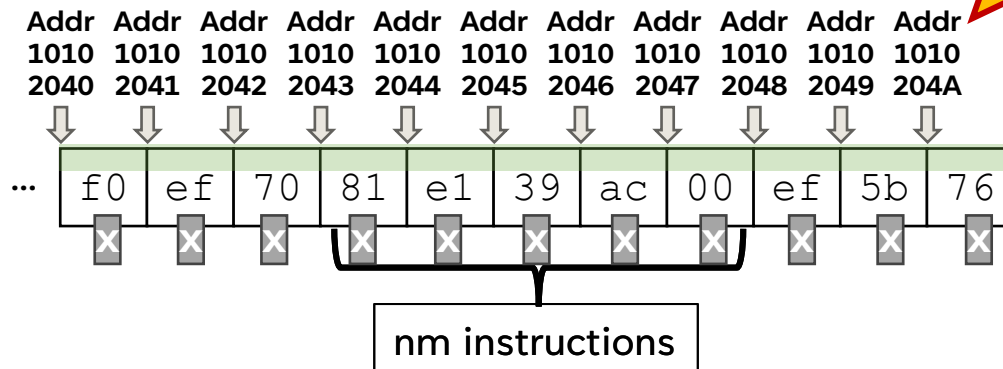
*... start executing the bytes
we put in memory!*

LF LF sp B (end)
0x10 0x10 0x64 0x7e 0x00

*jump to the "return" address
(which we overwrote with this
address) ...*

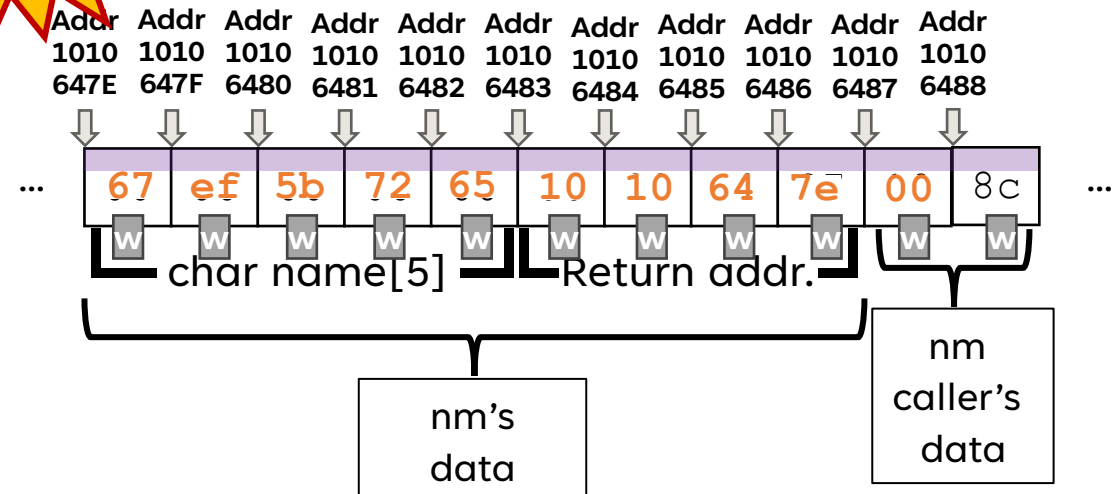
W⊗X: hardware extension, assure every byte is EITHER executable OR writable (NEVER both)

Program instructions



Crash

Program (meta)data



W⊗X: IMPACT

A HISTORY OF SUBVERSION

 explodeTheComputer isn't going to be in most programs
i.e. no good program ever includes 67 followed by ef 5b

 Can we inject our own payload?

1) Put 67 ef 5b into memory

2) Jump to that memory location

Find existing bytes of payload instructions

... followed by a return instruction (c3)

Return-oriented programming

ROP

A HISTORY OF SUBVERSION

explodeTheComputer isn't going to be in most programs
i.e. no good program ever includes 67 followed by ef 5b

Can we inject our own payload?

- 1) Put 67 ef 5b into memory
- 2) Jump to that memory location

Find existing bytes of payload instructions
... followed by a return instruction (c3)

Return-oriented programming

ROP

A HISTORY OF SUBVERSION

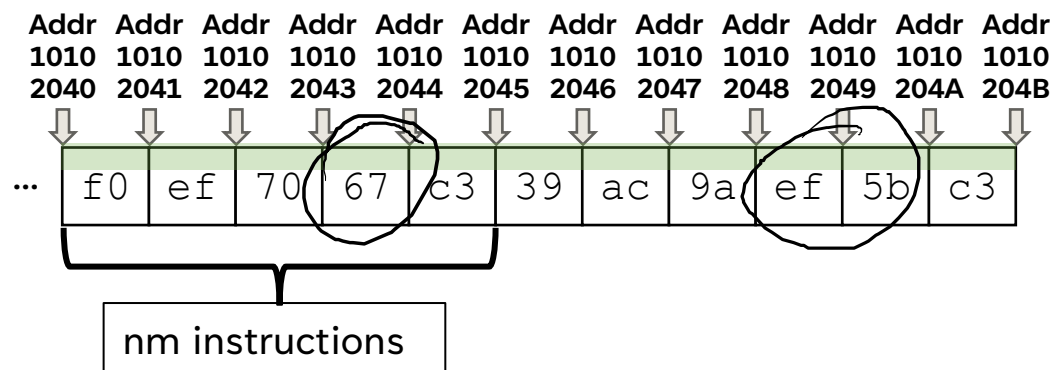
Find existing bytes of payload instructions

... followed by a return instruction (c3)

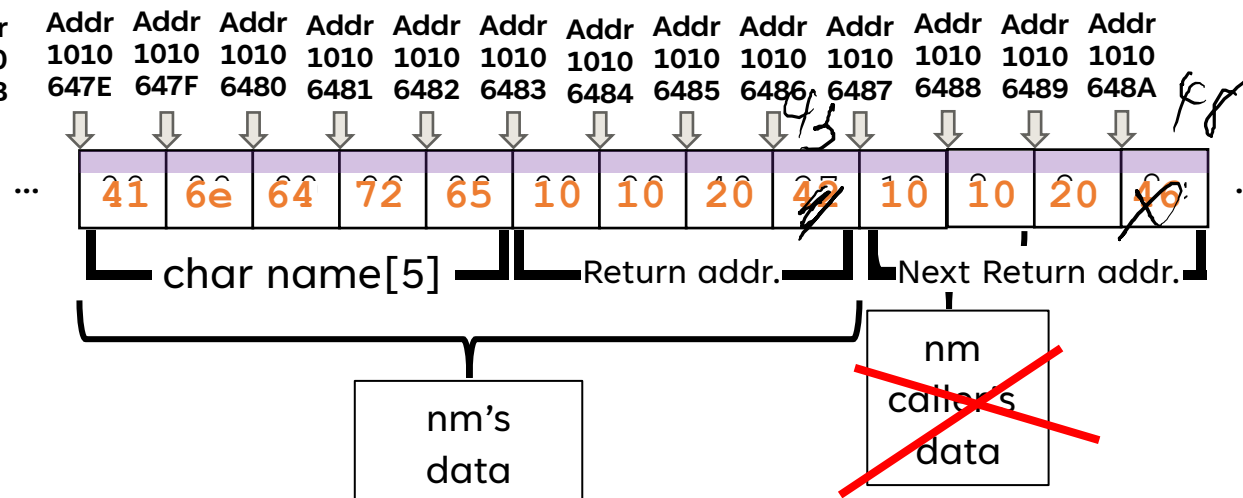
```
int nm() {
    char name[5];
    printf("Your name: ");
    → gets(name);
    printf("hi %s\n", name);
    return 0;
}
```

A n d r e LF LF sp B LF LF sp . (end)
 0x41 0x6e 0x64 0x72 0x65 0x10 0x10 0x20 0x42 0x10 0x10 0x20 0x46 0x00
 f3 f8

Program instructions



Program (meta)data



ROP: IMPACT

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

we find in a specific distribution are that, because of the proper- est, in any sufficiently large body e will feature sequences that al- ilar gadgets. (This claim is our ree major contributions:

at algorithm for analyzing libe to n sequences that can be used in

vered from a particular version be gadgets that allow arbitrary cing many techniques that lay hat we call, facetiously, *return-*

we provide strong evidence for late for how one might explore mine whether they provide fur-

es several smaller contributions. ented shellcode and show how it e a study of the provenance of on of libe we study, and consider ould be eliminated by compiler y our attack techniques fit within nto-libc techniques.

Attacks and Defenses

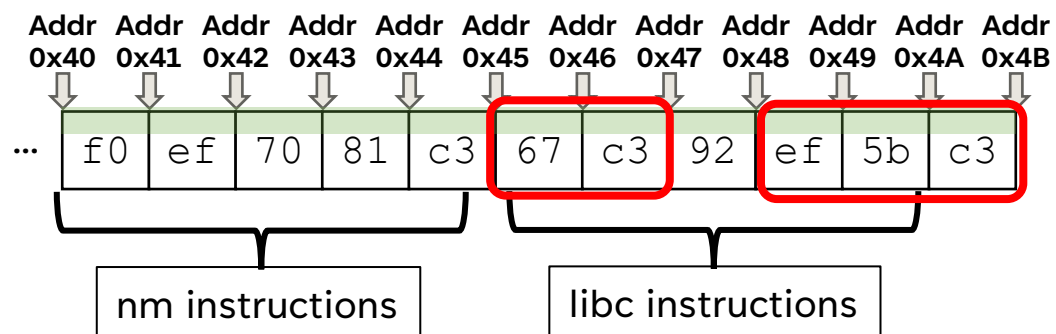
o has discovered a vulnerability s to exploit it. Exploitation, in e subverts the program's control tions of his choice with its cre- nability in this context is the e [1], though many other classes considered, such as buffer over- 3], integer overflows [34, 11, 4], ilities [25, 10]. In each case, the w tasks: he must find some way trol flow from its normal course, ram to act in the manner of his k-smashing attacks, an attacker overwriting a return address on e to code of his choosing rather made the call. (Though even in an be used, such as frame-pointer letes the second task by inject- age; the modified return address

RETURN-INTO-LIBC

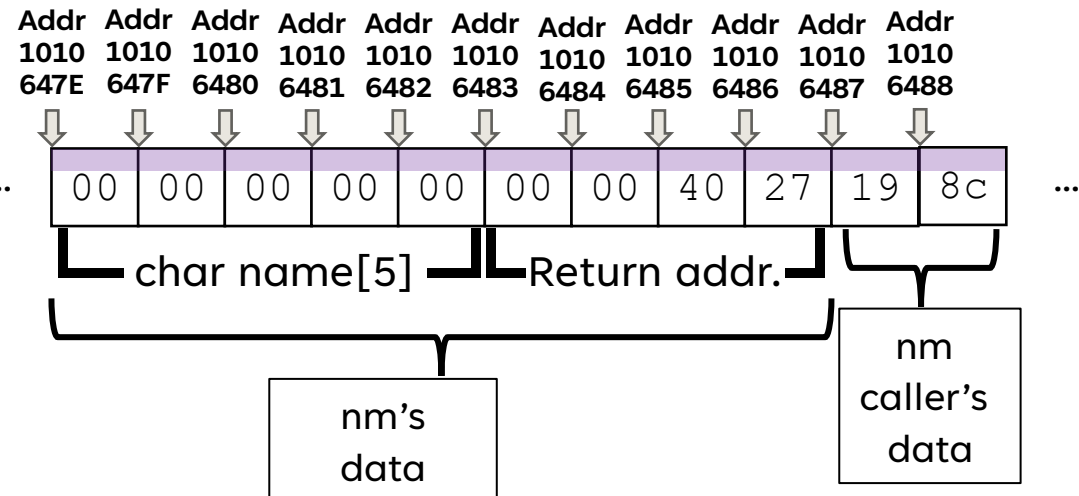
A HISTORY OF SUBVERSION

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

Program instructions



Program (meta)data

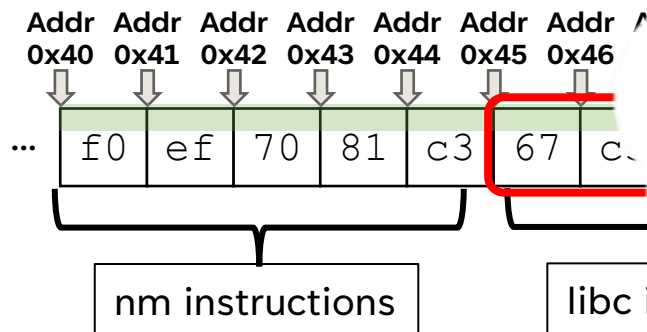


RETURN-INTO-LIBC

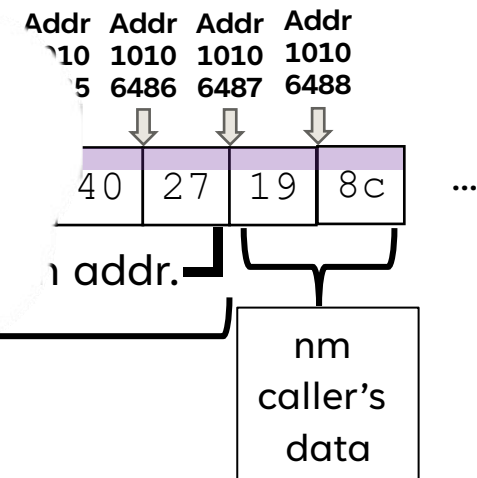
A HISTORY OF SUBVERSION

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

Program instructions



libc caller's data



IS IT TIME
TO PANIC?

nm's data

(yes, time to panic)



THINKING ABOUT DEFENSES

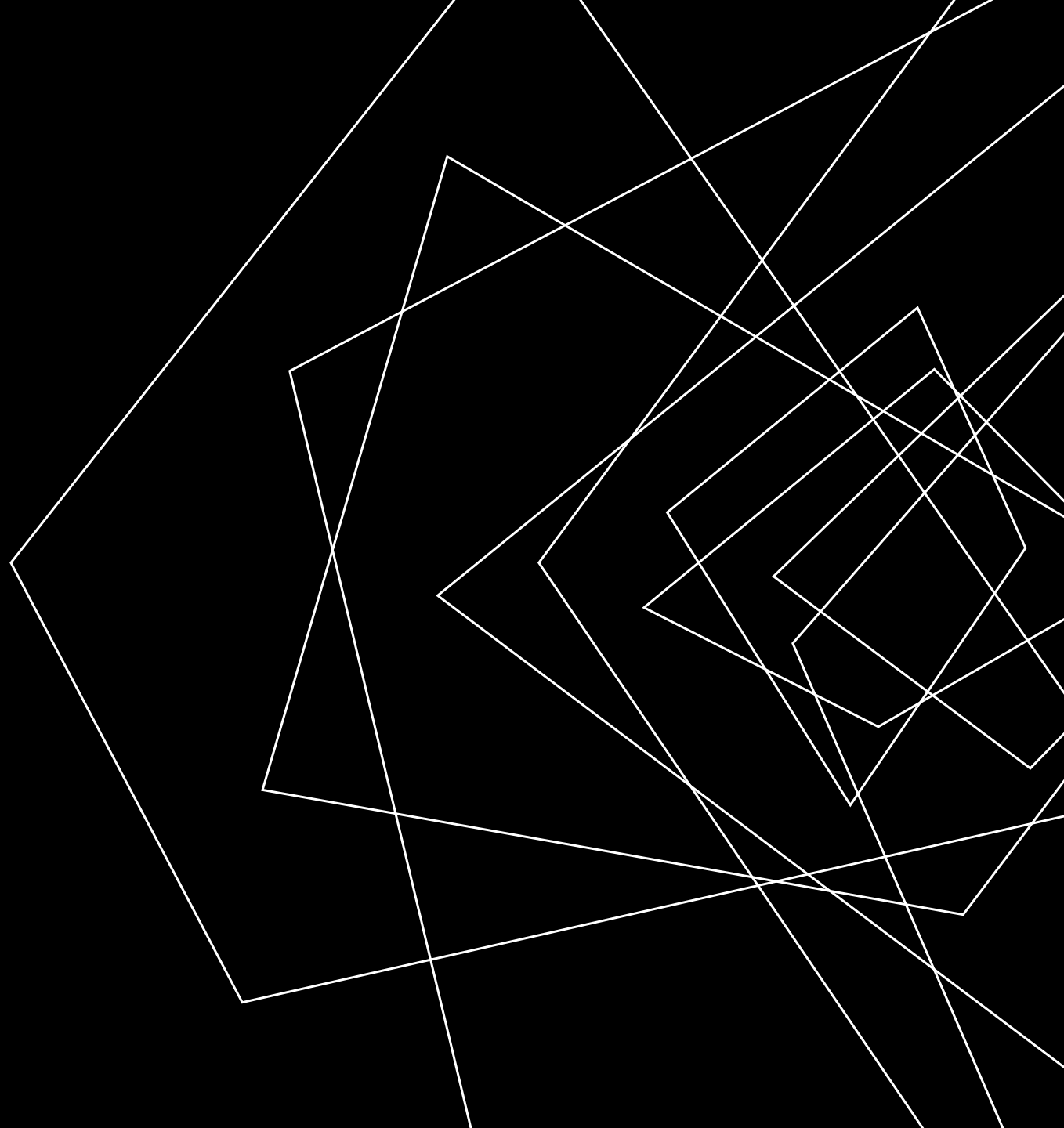
A HISTORY OF SUBVERSION

CAN WE DETECT PROGRAMS WITH OVERFLOW POTENTIAL?

en.wikipedia.org/wiki/C_file_input/output#gets

input/output		fwrite	Writes to a file
Unformatted input/output	fgetc getc	fgetwc getwc	Reads a byte/wchar_t from a file stream
	fgets	fgetws	Reads a byte/wchar_t line from a file stream
	fputc putc	fputwc putwc	Writes a byte/wchar_t to a file stream
	fputs	fputws	Writes a byte/wchar_t string to a file stream
	getchar	getwchar	Reads a byte/wchar_t from stdin
	gets	—	Reads a byte string from stdin until a newline or end of file is encountered (deprecated in C99, removed from C11)
	putchar	putwchar	Writes a byte/wchar_t to stdout
	puts	—	Writes a byte string to stdout
	ungetc	ungetwc	Puts a byte/wchar_t back into a file stream

END LECTURE



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

STACK CANARIES

A HISTORY OF SUBVERSION



STACK CANARIES

A HISTORY OF SUBVERSION

Program instructions (binary sequences)

Program data & metadata

User data

f0ef7081e1539ac00ef5b761b4fb01b351308dd003cb4b8930e27195a6ef34ba476e80e53f2af0		
--	--	--

ASLR

A HISTORY OF SUBVERSION



ASLR

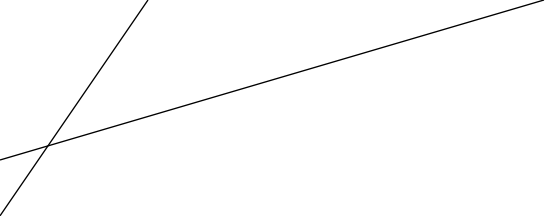
A HISTORY OF SUBVERSION

Program instructions (binary sequences)

Program data & metadata

User data

f0ef7081e1539ac00ef5b761b4fb01b351308dd003cb4b8930e27195a6ef34ba476e80e53f2af0		
--	--	--



CFI

A HISTORY OF SUBVERSION