EXERCISE #16

SIDE CHANNEL REVIEW

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

What is an advantage over an inline reference monitor over a reference monitor built into the OS? What about vice versa?

ADMINISTRIVIA AND ANNOUNCEMENTS

HISTORY OF MEMORY ATTACKS

EECS 677: Software Security Evaluation

Drew Davidson

CLASS PROGRESS

DESCRIBING SOME CLASSES OF ATTACKS AND DEFENSES

LAST TIME: REFERENCE MONITORS

REVIEW: LAST LECTURE

Live tracking of adherence to a safety policy

- **Kernelized reference monitor:** Add policy enforcement in the underlying trusted computing base, policies strictly over system events
- **Wrapper reference monitor:** Add a new enforcement layer that mediates system actions
- **Inline reference monitor:** add probes (and potentially enforcement) into the body of the program itself

OVERVIEW

PREVENTING BAD STUFF FROM HAPPENING IN A PROGRAM

HOW DO “BAD” PROGRAMS RUN?

A HISTORY OF SUBVERSION

Reactive Concerns

- Social engineering
- “Flaws” in system installation policies

Proactive Concerns

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

HOW DO "BAD" PROGRAMS RUN?

A HISTORY OF SUBVERSION

Reactive Concerns

- Social engineering
- "Flaws" in system installation policies

We're concerned about all these threats

Proactive Concerns

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

Let's focus on this one for now

LECTURE OUTLINE

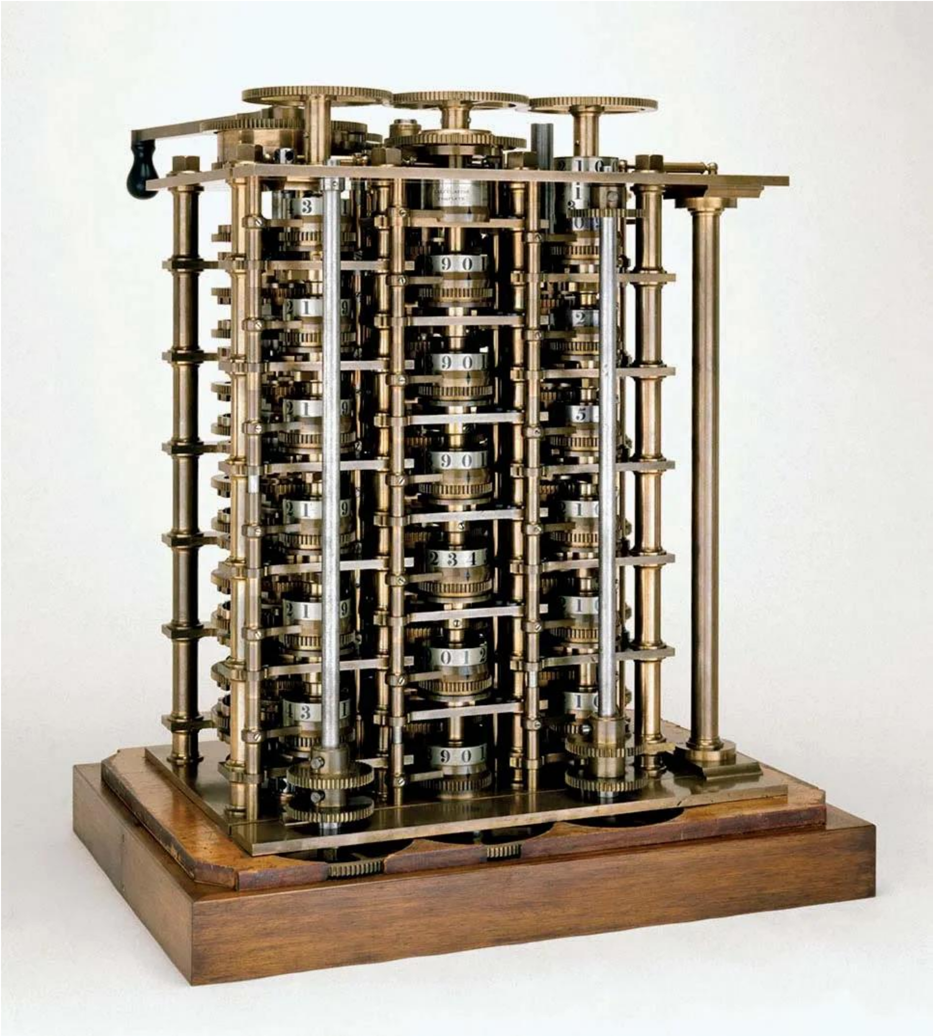
- A history of computers
- A history of subversion
- Defenses

DATA AND CODE

A HISTORY OF COMPUTING

Consider the History of Computation

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



DATA AND CODE
A HISTORY OF COMPUTING

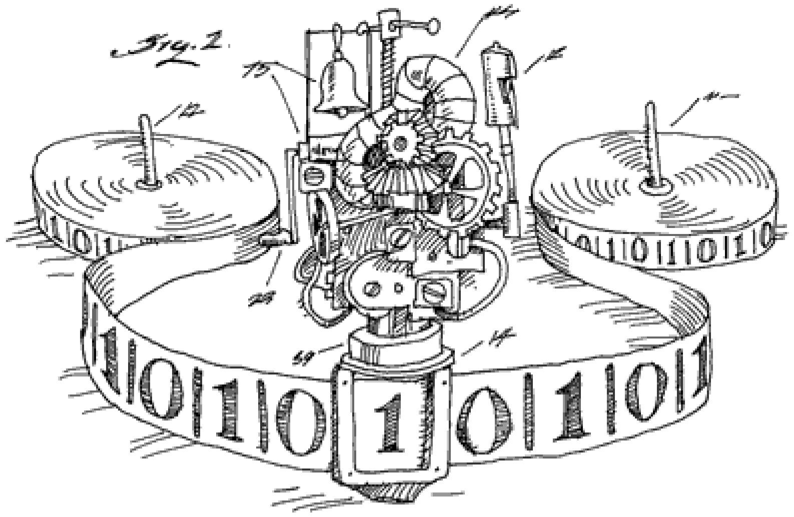
Consider the History of Computation

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 Fibonnaci Sequence

- Fibonnaci 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 HISTORY OF COMPUTING

A major paradigm shift: The Universal Computation Machine

The hardware contains generally-useful instructions

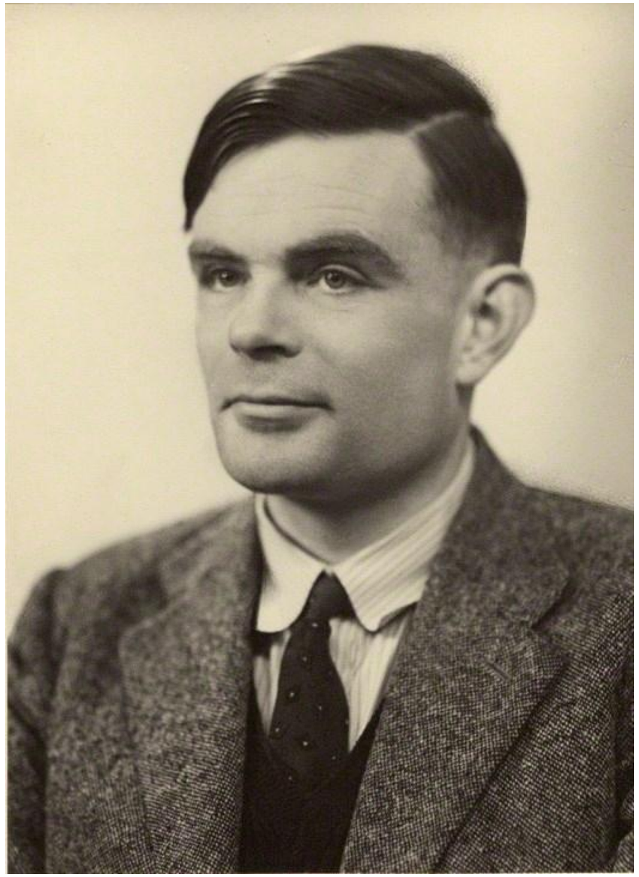
A particular algorithm is encoded in terms of those instructions

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
A HISTORY OF COMPUTING

A major paradigm shift: The Universal Computation Machine

The hardware contains generally-useful instructions

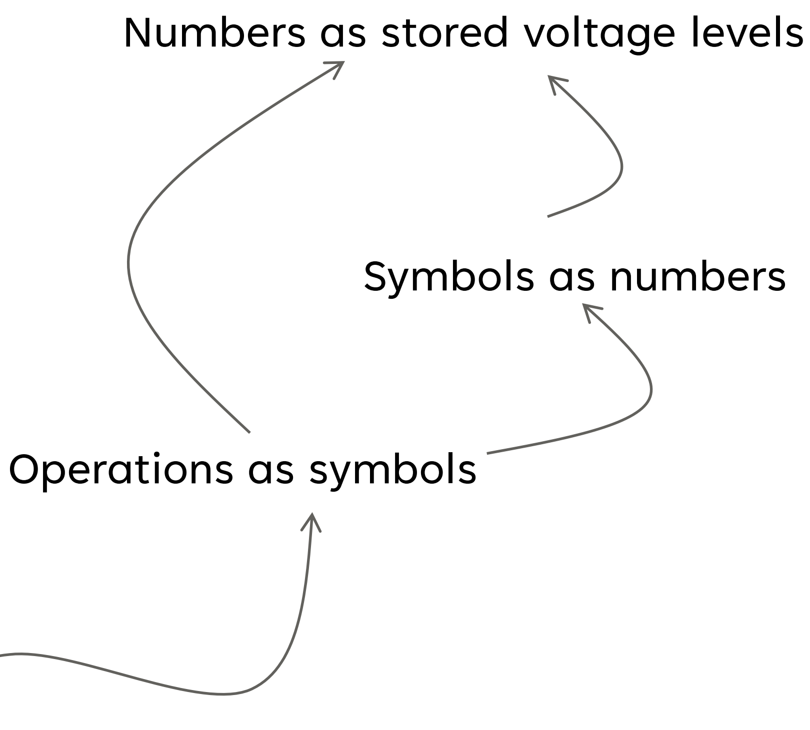
A particular algorithm is encoded in terms of those instructions

The Theory: The Universal Turing Machine

Consider the theory analogy: a Turing Machine that computes

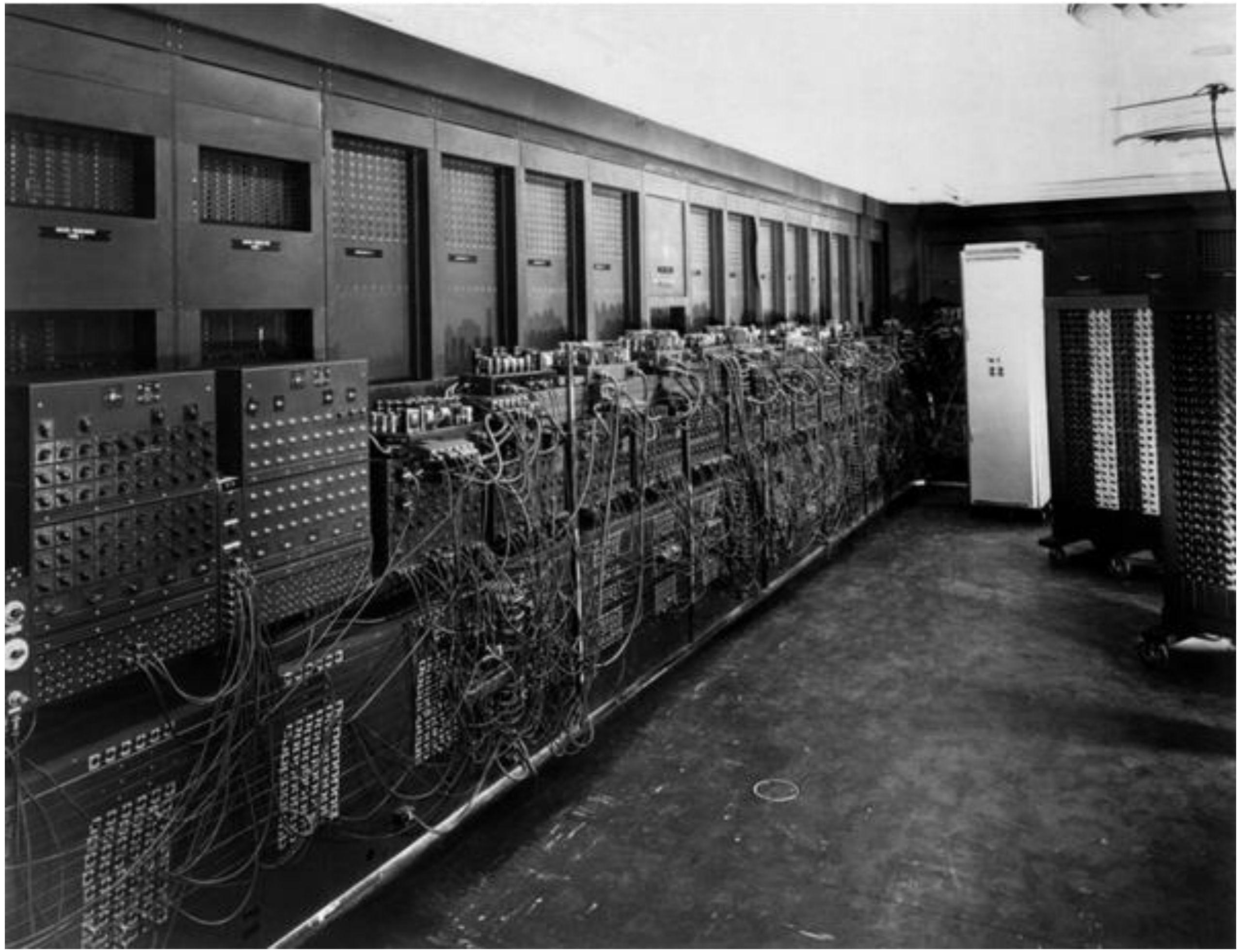
<u>any</u> function

- “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 HISTORY OF COMPUTING



Numbers as stored voltage levels

Symbols as numbers

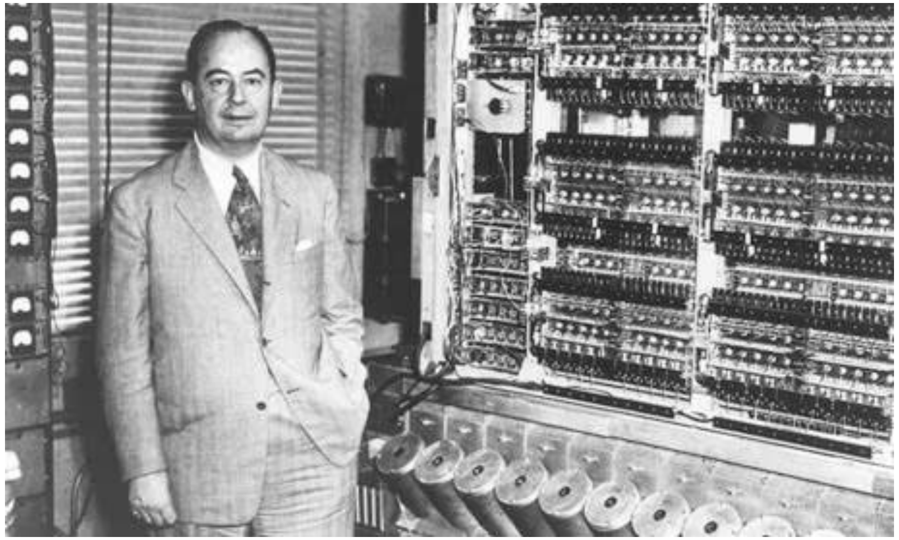
Operations as symbols

Code _is_ data

DATA AND CODE
A HISTORY OF COMPUTING

The Von Neumann architecture

Another big idea: Code and data share memory



Code <u>is</u> data

DATA AND CODE

A HISTORY OF COMPUTING

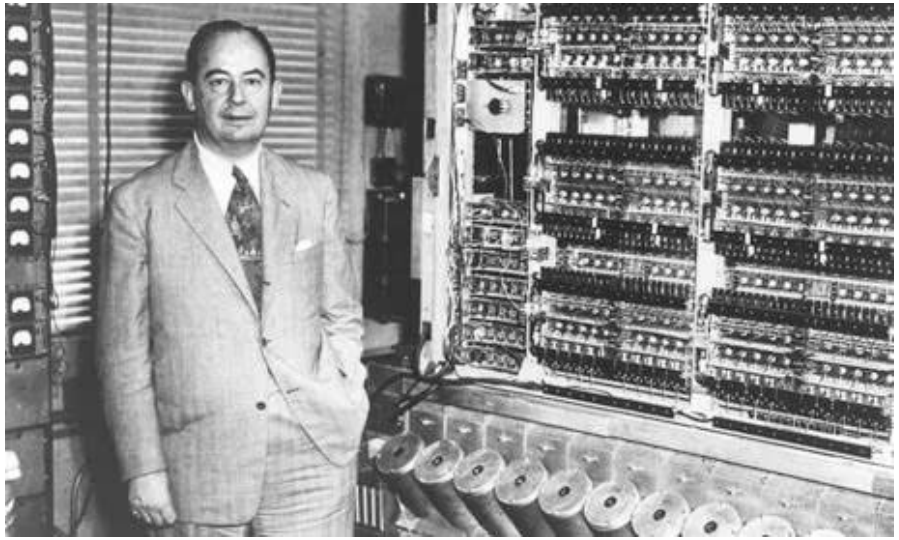
The Von Neumann architecture

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

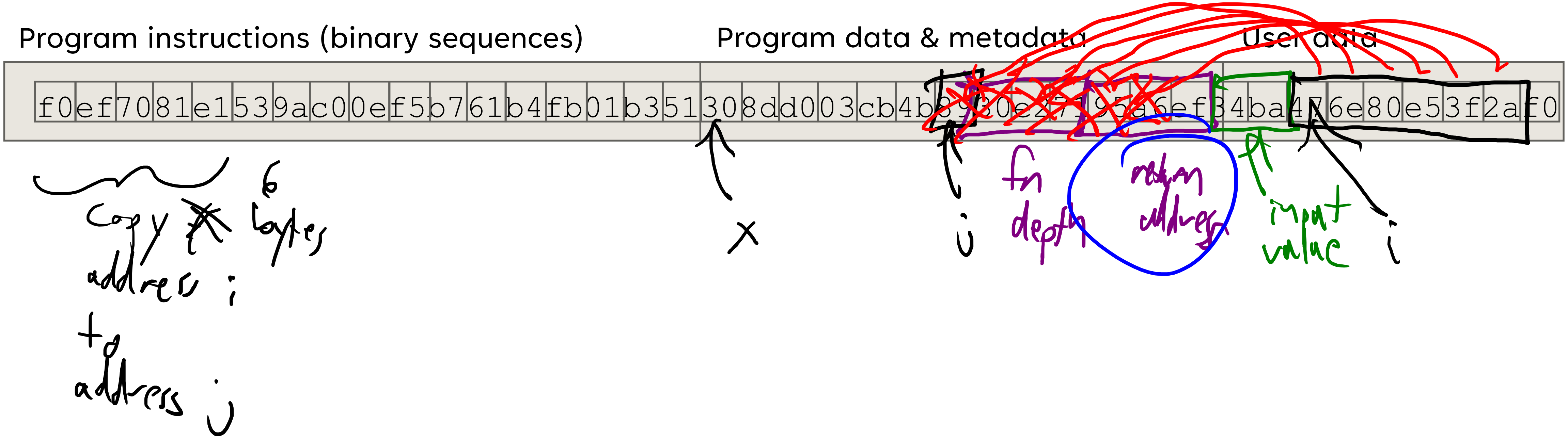
LECTURE OUTLINE

- A history of computers
- A history of subversion
- Defenses

BUFFER OVERFLOWS

A HISTORY OF SUBVERSION

A SIMPLIFIED VIEW OF PROGRAM MEMORY

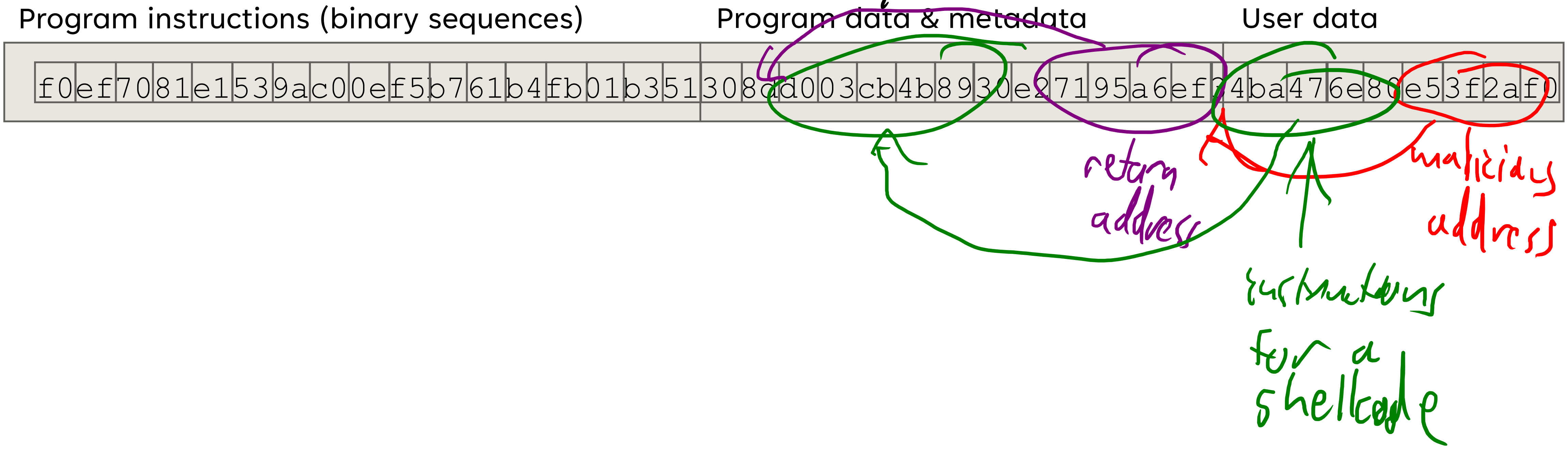


CODE INJECTION

A HISTORY OF SUBVERSION

Treating User Data as Code

- Adversary can overwrite return address via a buffer overflow
- Choose the bytes in the buffer



Time to panic? Not so fast

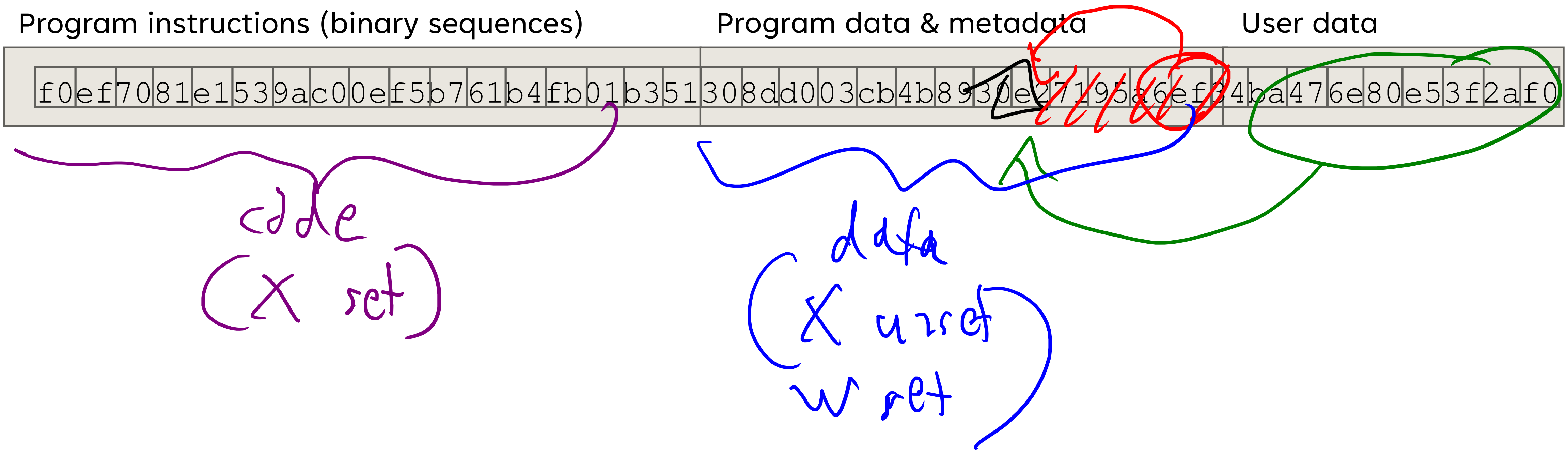
DEFEATING CODE INJECTION: W⊗X

A HISTORY OF SUBVERSION

SEPARATE ALL USER DATA FROM CODE

New rule for EVERY byte of process memory:

A byte can be writeable OR executable but never both



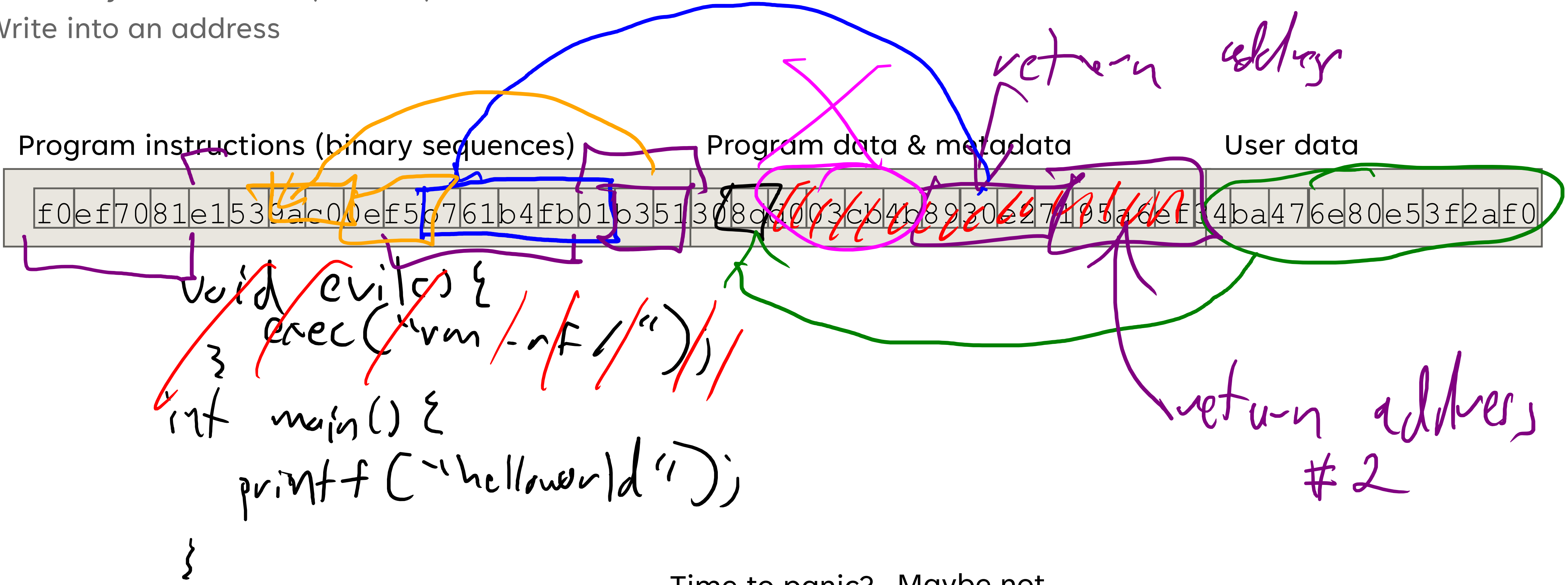
RETURN-ORIENTED PROGRAMMING

A HISTORY OF SUBVERSION

RECALL THAT CODE HAS AN UNDERLYING BINARY (NUMERIC) REPRESENTATION

Adversary still controls (at least) one control transfer

Write into an address



Time to panic?   Maybe not...

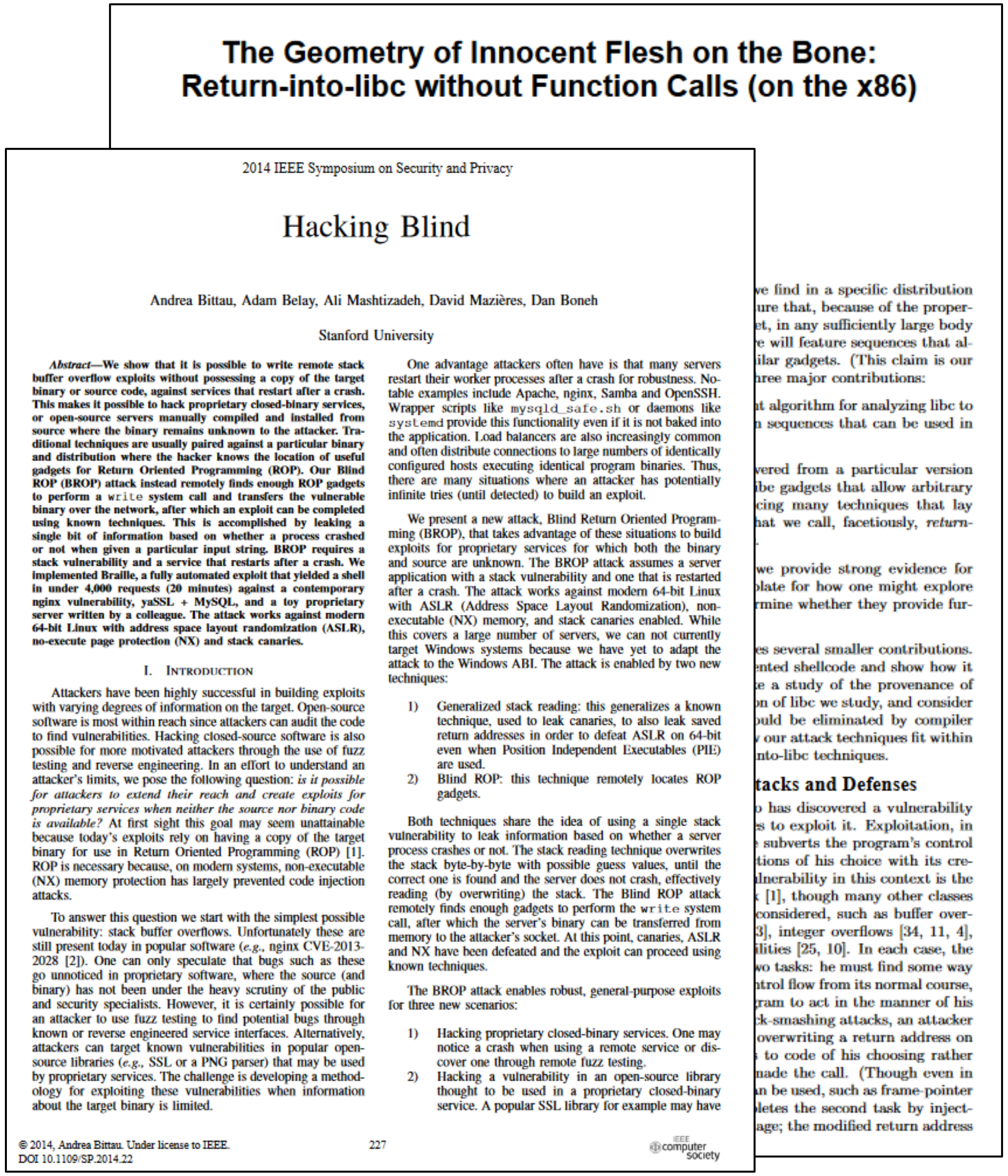
ROP CHALLENGES

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

© 2014, Andrea Bittau. Under license to IEEE.
DOI 10.1109/SP.2014.22

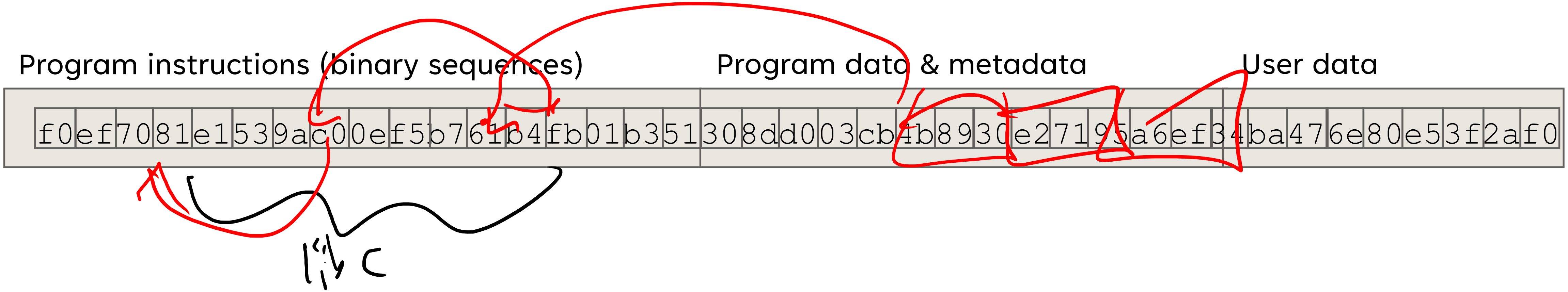
227

IEEE computer society

RETURN INTO LIBC

A HISTORY OF SUBVERSION

RECALL THAT CODE HAS AN UNDERLYING BINARY (NUMERIC) REPRESENTATION



Time to panic?


(yes, time to panic)

LECTURE OUTLINE

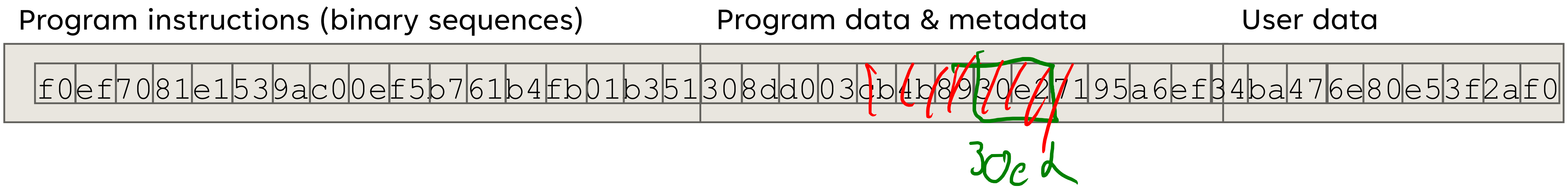
- A history of computers
- A history of subversion
- Defenses

STACK CANARIES

A HISTORY OF SUBVERSION

STACK CANARIES

A HISTORY OF SUBVERSION

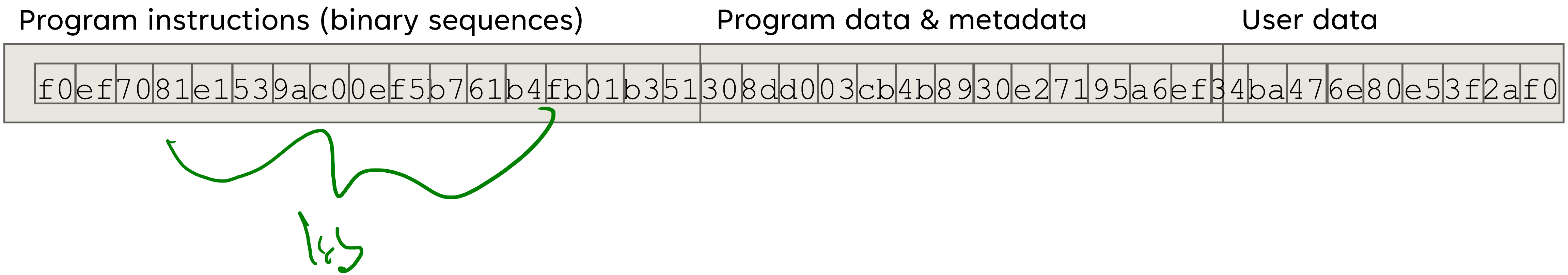


ASLR

A HISTORY OF SUBVERSION

ASLR

A HISTORY OF SUBVERSION



CFI

A HISTORY OF SUBVERSION

WRAP-UP