Lecture #6: ROP

UCalgary ENSF619

Elements of Software Security

Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)

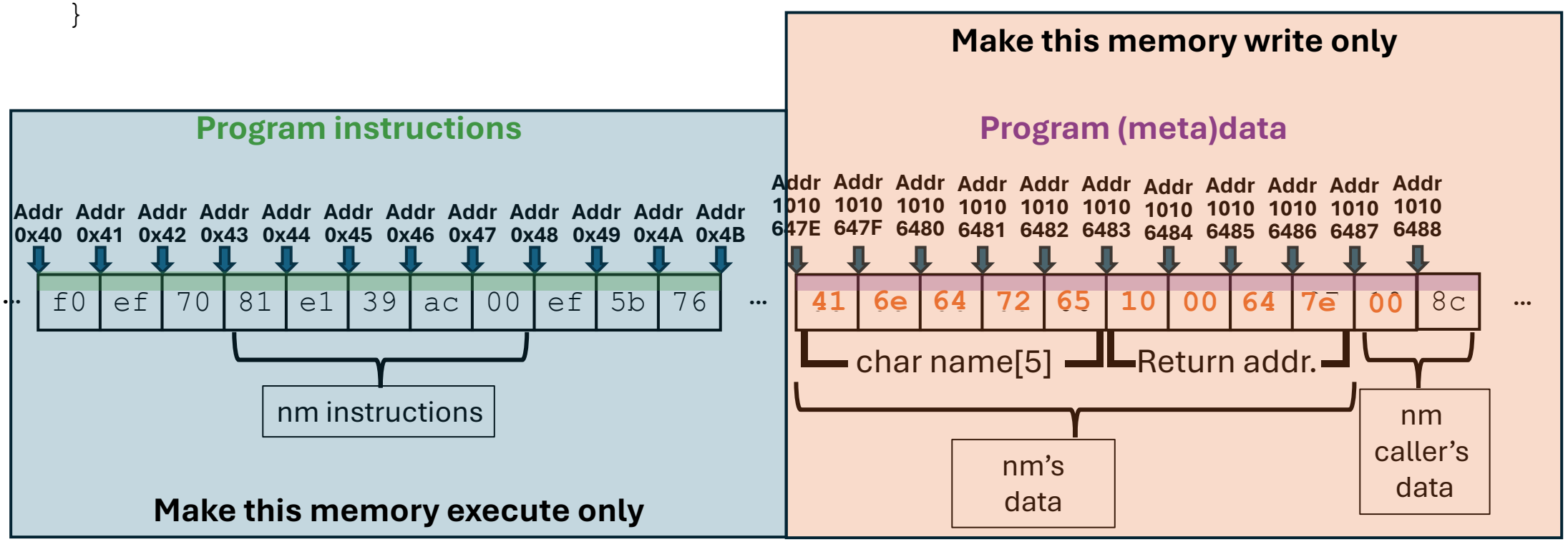
The context

- Stack overflow is a recurring problem
- Overwriting the stack is not too hard
- But… memory protection makes it more difficult!

Memory protection bits

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

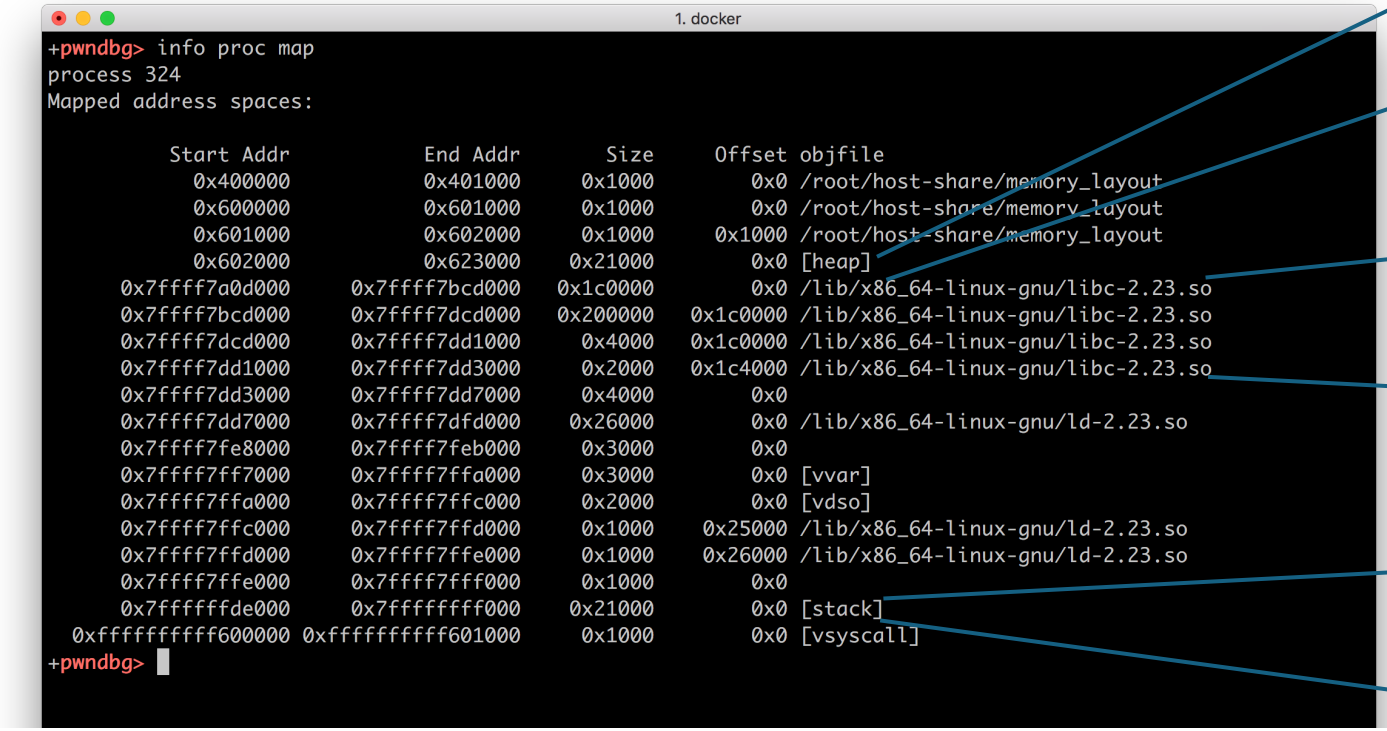
A	n	d	r	e	LF	LF	d	~	(end)
0x41	0x6e	0x64	0x72	0x65	0x10	0x10	0x64	0x7e	0x00



How can we move beyond this?

- Fundamental issue: now **we cannot add our own code** to the program
- It seems like this is **game over**, but…
- First (limited) approach to overcoming this: **return into libc**
- Core idea: rather than adding our own code, we are going to **leverage code which is in the memory space of the program**

Program memory layout



How can we use this code?

- In a return-into-libc attack, an attacker **takes control of the stack**…
- …and causes execution to **jump into a libc function**
- **Not very powerful/expressive**
- Can try to run a **syscall** such as **`system()`**
  - Runs a **shell** or similar
- **Not much else!**

Can we do better?

- For a while, it sounded like stack smashing could be somehow contained using memory protection bits etc.
- Until Hovav Sacham came along

Let's talk about the paper

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

Hovav Shacham*
hovav@cs.ucsd.edu

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all.* Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

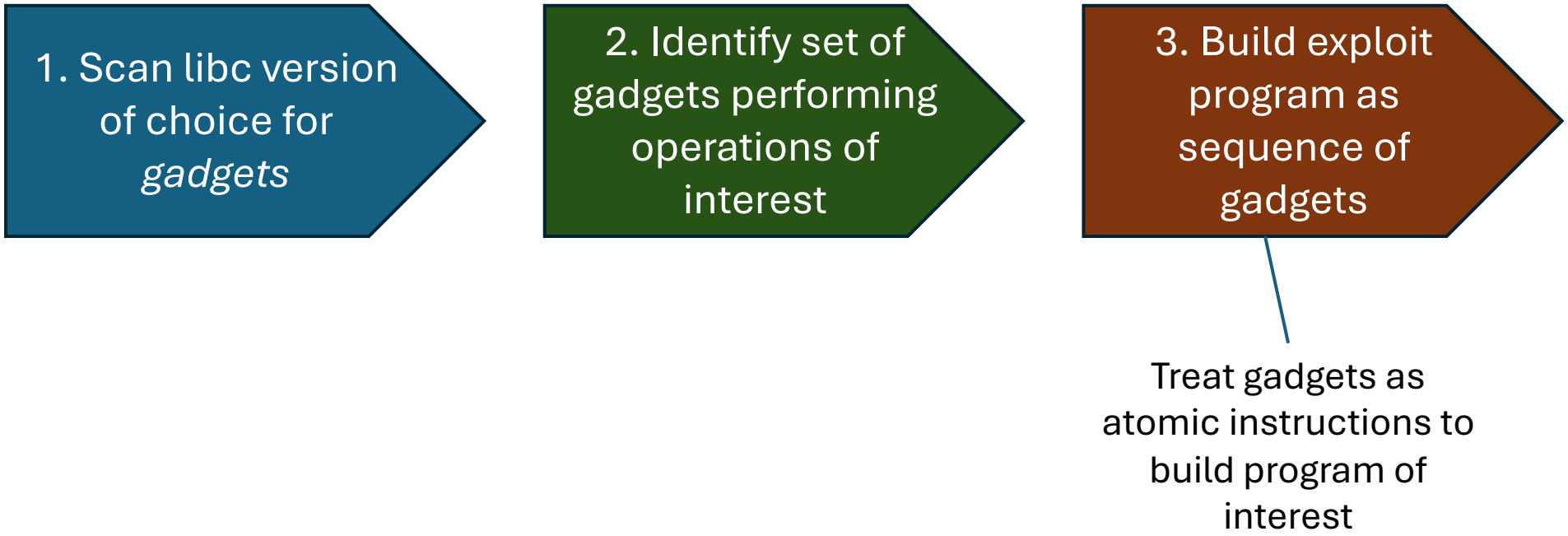
1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed "W⊕X" defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Attacks using our technique call no functions whatsoever. In fact, the use instruction sequences from libc that weren't placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler's code generation choices.

Unlike previous attacks, ours combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to build such gadgets using the short sequences we find in a specific distribution of GNU libc, and we conjecture that, because of the properties of the x86 instruction set, in any sufficiently large body of x86 executable code there will feature sequences that allow the construction of similar gadgets. (This claim is our *thesis.*) Our paper makes three major contributions:

10,000 feet view of the approach



Step 1: some context

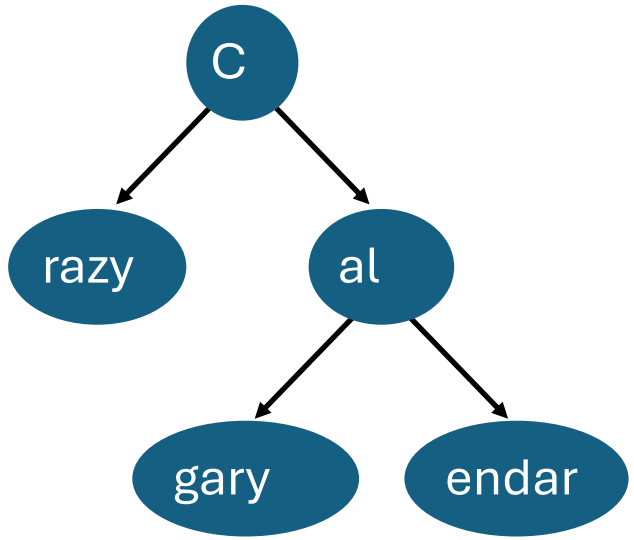
- A static analysis algorithm scans the libc binary, looking for useful instruction sequences and organizing them in **tries**
- What is a trie?

Lookup tries

- **Trie:** tree data structure used to store values associated with a **set of strings**
- Each **node** represents a **prefix** of one or more strings in the set
- Allows **compact encoding**, particularly when many strings share prefixes

Trie- example

- Example: need to store the following keys and values:
  - Calgary: 27
  - Calendar: 43
  - Crazy: 12



How are they used in the context of ROP?

- The algorithm instantiates a **trie** with a **return instruction** (“RET”) as **root node**
- The rest of the trie is constructed by:
  - Finding ret instructions
  - **Parsing the binary backward** from the ret instruction, interpreting it in every possible way until a ”boring” instruction is found
  - ”Boring” instruction: instruction that would prevent execution from reaching ret
- At the end, the trie represents every possible linear sequence of instructions ending with a RET

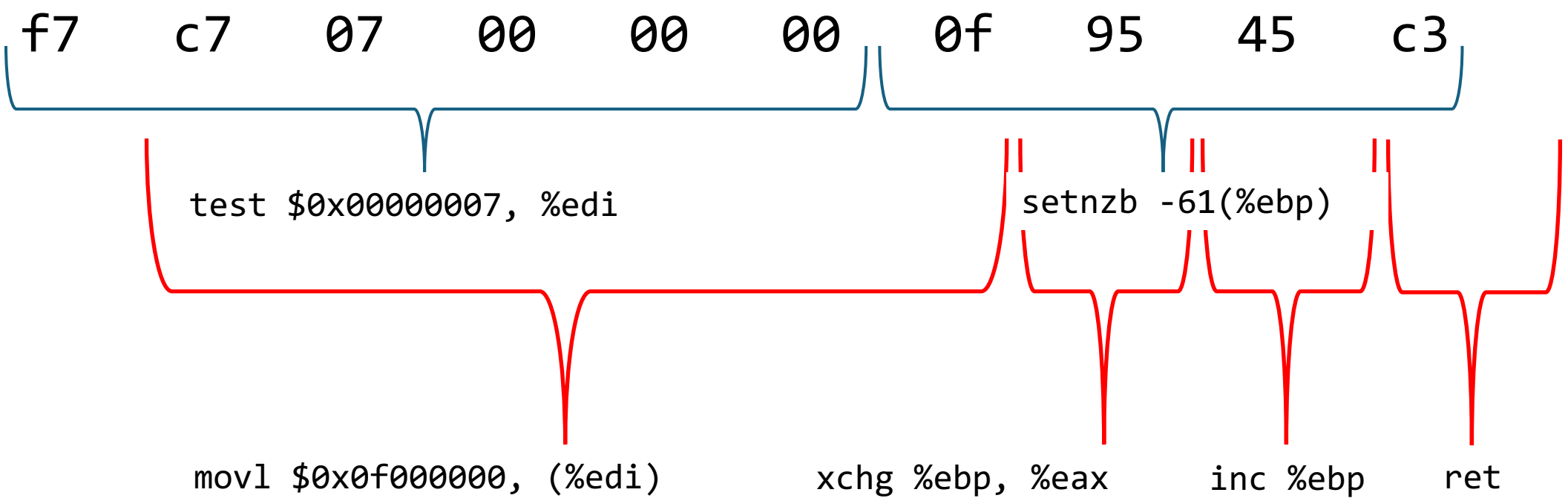
Step 2: gadgets – some more context

- x86 has various “return from call” instructions; the one used in the paper is “near return to calling procedure”, encoded as **byte C3**
- (There are other ones, but this is pretty much **the only one used** with a flat, virtually-addressed memory model)
- RET takes **no parameters**
- It will
  - **Pop the word** on top of the stack
  - Load it into the **IP register** (index pointer, AKA address of next instruction)

…even more context

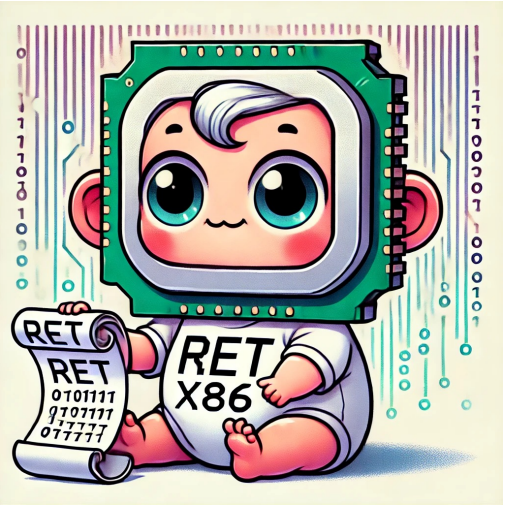
- x86 is an old, ~~poorly designed~~ complex instruction set
  - Many instructions have **variable length** and take **different parameters**
  - Given a **random sequence of bytes**, it is somewhat likely that that sequence will **contain at least some valid instructions**
- Consequence: a binary code segment will contain **many instruction sequences ending with RET**
  - Some of them are the **actual instructions** in the program
  - Some of them are **accidental** – misaligned sequences of bytes that just happen to look exactly like legitimate instructions

Example (from the paper)



What are those gadgets anyway?

- Small, cutesy **sequences of instructions**
- Basically, each gadget implements a **basic op**
  - E.g., sums, conditionals, load/store, etc.
- Put together the gadgets, and you have a **Turing-complete language**!
  - … what is a Turing-complete language?



How do those gadgets look?

- LOAD CONSTANT instruction:

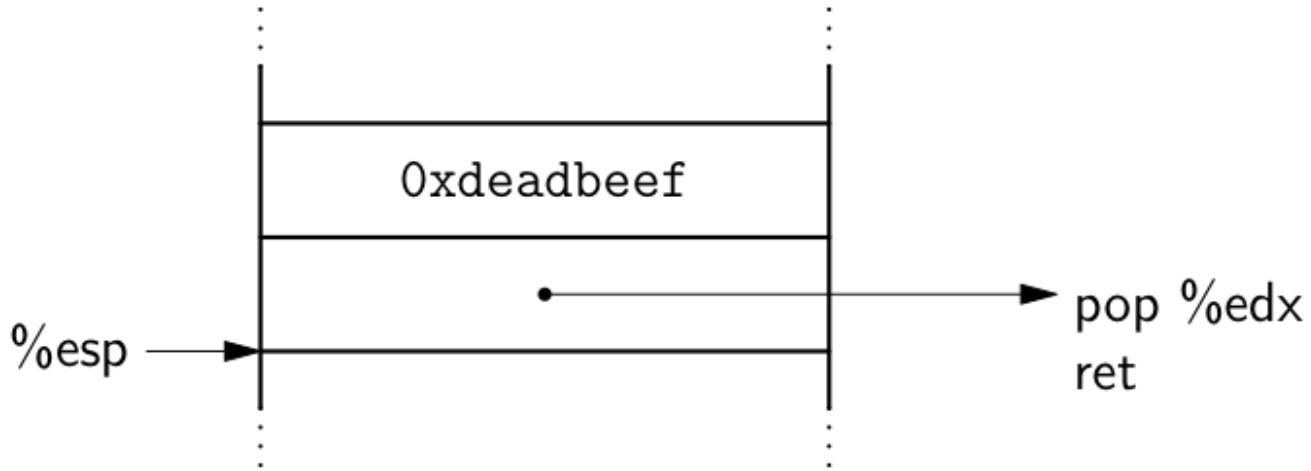


Figure 2: Load the constant `0xdeadbeef` into %edx.

How do those gadgets look? /2

- STORE value into memory:

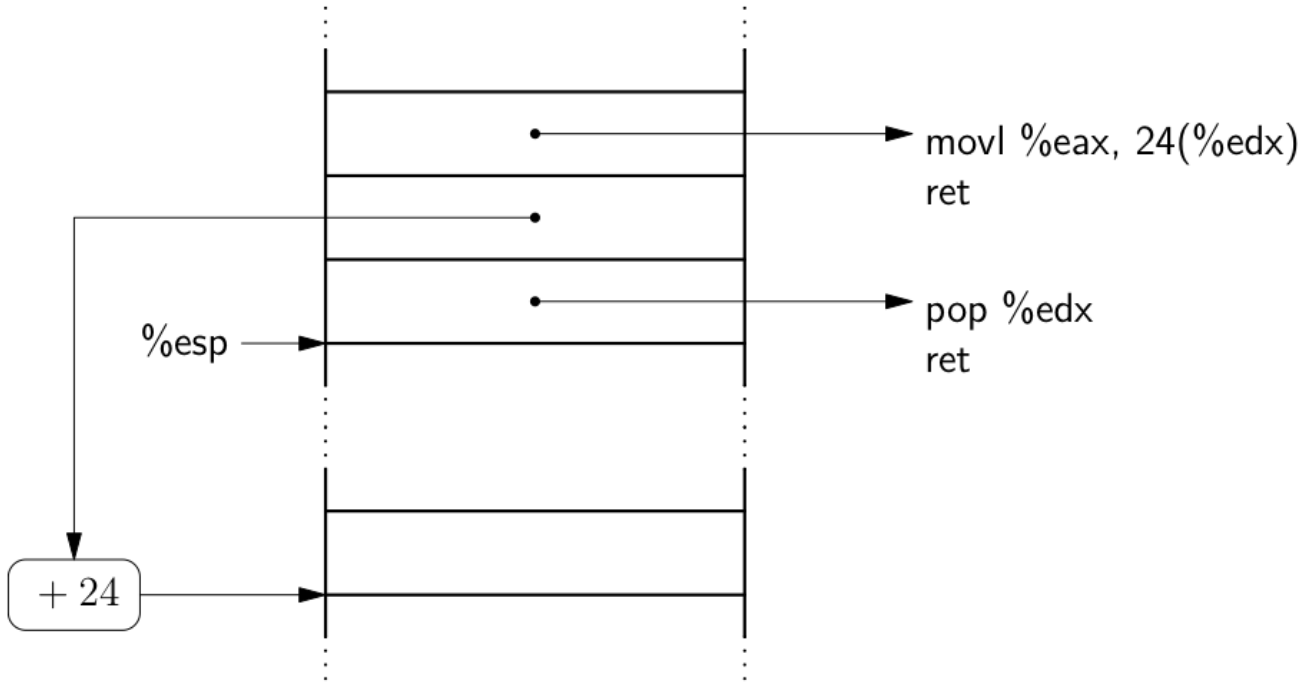


Figure 4: Store %eax to a word in memory.

Putting it all together…

- **Overflow stack**
- **Store sequences of addresses of gadgets in memory**
  - Sequence should match the program that needs to be executed
  - Program is the composition of multiple gadgets
- When the function being exploited returns, **control will transfer to the first gadget**…
- …then the second…
- … and so on…

Some considerations

Does this really work?

- **Yes!**
- It may sounds unlikely to be able to find all the correct gadgets but…
- … libc contains **millions of bytes** in its binary code
- … you only need **one of each gadget**!

Can we try to have less gadgets in binaries?

- Difficult… again, you only need one

Is Less *Really* More? Why Reducing Code Reuse Gadget Counts via Software Debloating Doesn't Necessarily Indicate Improved Security

Michael D. Brown, *Georgia Institute of Technology* Santosh Pande, *Georgia Institute of Technology*

Abstract

Nearly all modern software suffers from bloat that negatively impacts its performance and security. To combat this problem, several automated techniques have been proposed to debloat software. A key metric used in many of these works to demonstrate improved security is code reuse gadget count reduction. The use of this metric is based on the prevailing idea that reducing the number of gadgets available in a software package reduces its attack surface and makes mounting a gadget-based code reuse exploit such as return-oriented programming (ROP) more difficult for an attacker.

In this paper, we challenge this idea and show through a variety of realistic debloating scenarios the flaws inherent to the gadget count reduction metric. Specifically, we demonstrate that software debloating can achieve high gadget count reduction rates, yet fail to limit an attacker's ability to construct an exploit. Worse yet, in some scenarios high gadget count reduction rates conceal instances in which software debloating makes security *worse* by introducing new, useful gadgets.

To address these issues, we propose a set of four new metrics for measuring security improvements realized through software debloating that are quality-oriented rather than quantity-oriented. We show that these metrics can identify when debloating negatively impacts security and be efficiently calculated using our static binary analysis tool, the Gadget Set Analyzer. Finally, we demonstrate the utility of these metrics in two realistic case studies: iterative debloating and debloater evaluation.

libraries such as `libc` typically only require a small number of functions provided by the library, but load the entire library into the program's memory space at runtime.

Software bloat also occurs laterally within software packages suffering from feature creep [23]. Examples include software such as cUrl, which can be used to transfer data via 23 different protocols, and iTunes, which features a media player, ecommerce platform, and hardware device interface within a single package. Since end users are unlikely to use every feature within these packages, the code associated with unused features contributes to software bloat.

Recently, several software debloating techniques [2-5, 20, 24-27] have been proposed that promise to improve software security by removing code bloat at various stages of the software lifecycle. One frequently utilized metric for measuring security improvements realized via debloating is the reduction in total count of code reuse gadgets available to an attacker, which we refer to as *gadget count reduction*. Several recent debloating publications [3-5, 24-27] claim their methods improve security citing gadget count reduction data as one form of evidence.

The relationship between gadget count reduction and improved security is based on the premise that reducing the total number of code reuse gadgets available in a software package reduces its attack surface. In turn, this decreases the likelihood of an attacker successfully constructing a code reuse exploit using techniques such as return, jump, or call-oriented programming (also known as ROP, JOP, and COP [7-9, 21]).

Table 3: Quality Gadget Counts and Average Quality

	Debloated Variant	Quality ROP Gadgets	Average ROP Gadget Quality
CARVE	libmodbus (C)	228 (61)	**0.81 (0.10)**
	libmodbus (M)	231 (58)	0.86 (0.05)
	libmodbus (A)	205 (84)	**0.76 (0.14)**
	Bftpd (C)	246 (16)	0.96 (-0.02)
	Bftpd (M)	227 (35)	0.93 (0.01)
	Bftpd (A)	197 (65)	0.91 (0.03)
	libcurl (C)	1800 (25)	0.86 (0.00)
	libcurl (M)	1619 (206)	0.85 (0.01)
	libcurl (A)	1331 (494)	0.84 (0.03)
	Mongoose (C)	436 (8)	1.07 (0.02)
	Mongoose (M)	422 (22)	1.07 (0.02)
	Mongoose (A)	400 (44)	1.05 (0.03)
CHISEL	bzip2	96 (55)	1.30 (-0.20)
	chown	99 (124)	1.35 (-0.01)
	date	100 (120)	**1.01 (0.14)**
	grep	277 (106)	**1.13 (0.17)**
	gzip	90 (75)	1.13 (0.00)
	mkdir	66 (46)	1.10 (0.05)
	rm	72 (155)	**1.23 (0.11)**
	tar	123 (298)	**1.15 (0.44)**
	uniq	66 (78)	1.23 (0.04)

Do we even need to have access to the bin?

- Not really! …provided that a certain conditions are met

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*

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.

What is this “hacking blind” business?

- Just a fancy technique to **copy the content of a binary** from **memory** to the **network**
- Requires that the binary **restarts when crashed** (e.g., web server service)
- Keep trying a ROP attack until identifies an address that causes **write** to a network socket before crashing
- Then, use that repeatedly to **exfiltrate in-memory binary data**
- Once enough binary exfiltrated, can use **standard tools** to carry a **ROP attacks**

Do ROP attacks only work on x86?

- No! They also work on **fixed-width instruction architectures** such as **ARM**
- At face value, ARM is a **terrible architecture for ROP**
  - ARM has 32-bit instructions that are **memory-aligned**
  - **Jumping** in the middle of an instruction will cause the program to **crash**!
  - **There is not even a proper RET instruction!**

Fear not! There's a solution for that

- Modern tools can find **plenty of ROP gadgets** even **within the constraints of aligned instructions**
  - Plus, ARM has **two modes**, regular and 16-bit (THUMB)
  - If you can find a gadget to switch CPU modes you get:
    - **Double the gadgets**
    - Double the fun (OK, I got this one from reddit)
- While it is true that there is no RET, there are other instructions
  - E.g., **POP {PC}**
  - Pops the top of the stack into the **program counter**
  - Tomato, tomato

What was the impact of this paper?

- This is a **foundational paper** in **system security**
- **ROP techniques** are used in **countless modern exploits**
- …oftentimes, together with other **circumvention techniques** such as **stack pivoting**
- **Probably as important as the Fun & Profit paper**

ROP tools

ropper
Standalone ROP gadget finder written in Python, can also display useful information about binary files. It has coloured output, interactive search and supports bad character lists. Check out the github page for more information.

ROPGadget
Another powerful ROP gadget finder, doesn't have the interactive search or colourful output that ropper features but it has stronger gadget detection when it comes to ARM architecture. It too has a github page.

pwntools
CTF framework written in Python. Simplifies interaction with local and remote binaries which makes testing your ROP chains on a target a lot easier. Check out the github page. After solving this site's first "ret2win" challenge, consider browsing an example solution written by the developer/maintainer of pwntools.

radare2
radare2 is a disassembler, debugger and binary analysis tool amongst many other things. It's absurdly powerful and you have more or less everything you need to complete the challenges on this site entirely within the radare2 framework. It's actively developed and you can find more detail on their github page which also hosts a cheatsheet.

pwndbg
pwndbg is a GDB plugin that greatly enhances its exploit development capability. It can make it much easier to understand your environment when debugging your solution to a challenge. The project has a homepage and is hosted on github. Also worth checking out are the list of features and cheatsheet.

https://ropemporium.com/guide.html

See you in the next lecture!