CSE 610 Special Topics:
System Security - Attack and Defense for Binaries

Instructor: Dr. Ziming Zhao

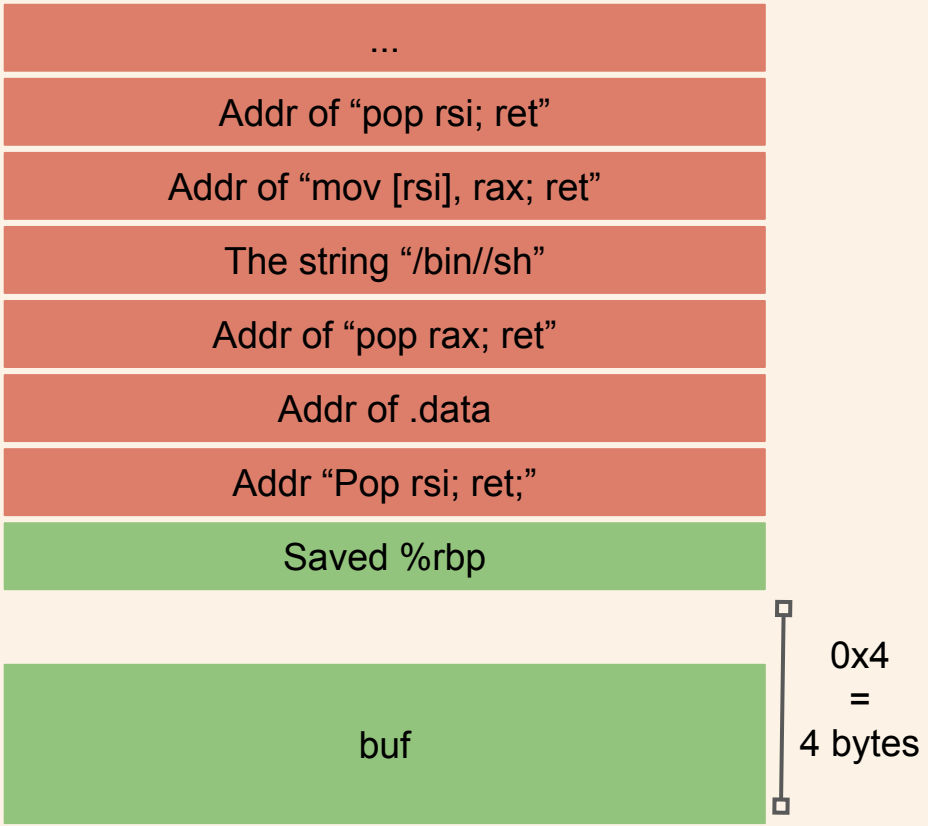
Location: Frnczk 408, North campus
Time: Monday, 5:20 PM - 8:10 PM

Today’s Agenda

1. Return-oriented Programing
    a. Built a ROP chain on LibC to open and read file
    b. Blind ROP
    c. Defenses

The Generated ROP Shellcode

```
...
p += pack('<Q', 0x000000000040f3ee) # pop rsi ; ret
p += pack('<Q', 0x00000000004c00e0) # @ .data
p += pack('<Q', 0x0000000000449237) # pop rax ; ret
p += '/bin//sh'
p += pack('<Q', 0x000000000047b755) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x000000000040f3ee) # pop rsi ; ret
p += pack('<Q', 0x00000000004c00e8) # @ .data + 8
p += pack('<Q', 0x0000000000443890) # xor rax, rax ; ret
p += pack('<Q', 0x000000000047b755) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x000000000040186a) # pop rdi ; ret
p += pack('<Q', 0x00000000004c00e0) # @ .data
p += pack('<Q', 0x000000000040f3ee) # pop rsi ; ret
p += pack('<Q', 0x00000000004c00e8) # @ .data + 8
p += pack('<Q', 0x000000000040176f) # pop rdx ; ret
p += pack('<Q', 0x00000000004c00e8) # @ .data + 8
p += pack('<Q', 0x0000000000443890) # xor rax, rax ; ret
p += pack('<Q', 0x0000000000470780) # add rax, 1 ; ret
p += pack('<Q', 0x0000000000470780) # add rax, 1 ; ret
...
```



Useful Gadgets

Skip data on stack:

pop rdx ; pop r12 ; ret
pop rdx ; pop rcx ; pop rbx ; ret

Useful Gadgets

Store value to registers and skip data on stack:

pop rdx ; pop r12 ; ret
pop rdx ; pop rcx ; pop rbx ; ret
pop rcx ; pop rbp ; pop r12 ; pop r13 ; ret

NOP:
ret;
nop; ret;

Useful Gadgets

syscall instruction is quite rare in normal programs; may have to call library functions instead.

Useful Gadgets

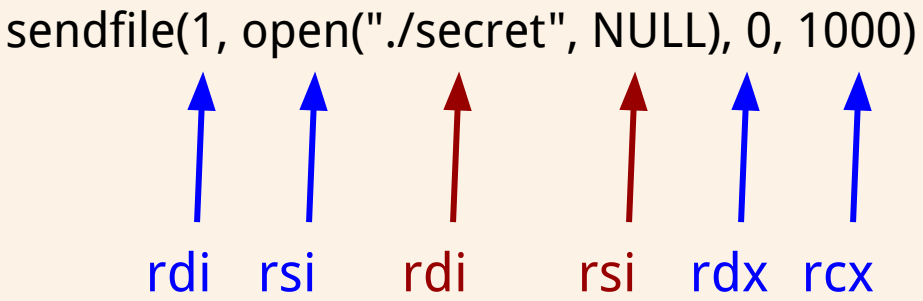
Stack pivot:

xchg rax, rsp; ret

pop rsp; ...; ret

ROP Shellcode to Read *secret* File

ret2libc64 dynamically linked



Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) %rdi, %rsi, %rdx, %rcx, %r8, %r9, ... (use stack for more arguments)

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

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:

IEEE S&P 2014

Defenses

G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries

Kaan Onarlioglu
Bilkent University, Ankara
onarliog@cs.bilkent.edu.tr

Leyla Bilge
Eurecom, Sophia Antipolis
bilge@eurecom.fr

Andrea Lanzi
Eurecom, Sophia Antipolis
lanzi@eurecom.fr

Davide Balzarotti
Eurecom, Sophia Antipolis
balzarotti@eurecom.fr

Engin Kirda
Eurecom, Sophia Antipolis
kirda@eurecom.fr

ABSTRACT

Despite the numerous prevention and protection mechanisms that have been introduced into modern operating systems, the exploitation of memory corruption vulnerabilities still represents a serious threat to the security of software systems and networks. A recent exploitation technique, called Return-Oriented Programming (ROP), has lately attracted a considerable attention from academia. Past research on the topic has mostly focused on refining the original attack technique, or on proposing partial solutions that target only particular variants of the attack.

In this paper, we present G-Free, a compiler-based approach that represents the first practical solution against any possible form of

to find a technique to overwrite a pointer in memory. Overflowing a buffer on the stack [5] or exploiting a format string vulnerability [26] are well-known examples of such techniques. Once the attacker is able to hijack the control flow of the application, the next step is to take control of the program execution to perform some malicious activity. This is typically done by injecting in the process memory a small payload that contains the machine code to perform the desired task.

A wide range of solutions have been proposed to defend against memory corruption attacks, and to increase the complexity of performing these two attack steps [10, 11, 12, 18, 35]. In particular, all modern operating systems support some form of memory pro-

kBouncer: Efficient and Transparent ROP Mitigation

Vasilis Pappas
Columbia University
vpappas@cs.columbia.edu

April 1, 2012

Abstract

The wide adoption of non-executable page protections in recent versions of popular operating systems has given rise to attacks that employ return-oriented programming (ROP) to achieve arbitrary code execution without the injection of any code. Existing defenses against ROP exploits either require source code or symbolic debugging information, impose a significant runtime overhead, which limits their applicability for the protection of third-party applications, or may require to make some assumptions about the executable code of the protected applications. We propose kBouncer, an efficient and fully transparent ROP mitigation technique that does not requires source code or debug symbols. kBouncer is based on runtime detection of abnormal control transfers using hardware features found on commodity processors.

1 Problem Description

The introduction of non-executable memory page protections led to the development of the return-to-libc exploitation technique [11]. Using this method, a memory corruption vulnerability can be exploited by transferring control to code that already exists in the address space of the vulnerable process. By jumping

HW Reading

SoK: Eternal War in Memory

László Szekeres[†], Mathias Payer[‡], Tao Wei[*‡], Dawn Song[‡]
[†]*Stony Brook University*
[‡]*University of California, Berkeley*
[*]*Peking University*

***Abstract*—Memory corruption bugs in software written in low-level languages like C or C++ are one of the oldest problems in computer security. The lack of safety in these languages allows attackers to alter the program's behavior or take full control over it by hijacking its control flow. This problem has existed for more than 30 years and a vast number of potential solutions have been proposed, yet memory corruption attacks continue to pose a serious threat. Real world exploits show that all currently deployed protections can be defeated.**

This paper sheds light on the primary reasons for this by describing attacks that succeed on today's systems. We systematize the current knowledge about various protection techniques by setting up a general model for memory corruption attacks. Using this model we show what policies can stop which attacks. The model identifies weaknesses of currently deployed techniques, as well as other proposed protections

try to write safe programs. The memory war effectively is an arms race between offense and defense. According to the MITRE ranking [1], memory corruption bugs are considered one of the top three most dangerous software errors. Google Chrome, one of the most secure web browsers written in C++, was exploited four times during the Pwn2Own/Pwnium hacking contests in 2012.

In the last 30 years a set of defenses has been developed against memory corruption attacks. Some of them are deployed in commodity systems and compilers, protecting applications from different forms of attacks. Stack cookies [2], exception handler validation [3], Data Execution Prevention [4] and Address Space Layout Randomization [5]