NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

EVALUATING THE GENERALITY AND LIMITS OF BLIND RETURN-ORIENTED PROGRAMMING ATTACKS

by

Lawrence Keener

December 2015

Thesis Advisor:	Mark Gondree
Second Reader:	Chris Eagle

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12-18-2015	Master's Thesis

4. TITLE AND SUBTITLE

EVALUATING THE GENERALITY AND LIMITS OF BLIND RETURN-ORIENTED PROGRAMMING ATTACKS

5. FUNDING NUMBERS

6. AUTHOR(S)

Lawrence Keener

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

We consider a recently proposed information disclosure vulnerability called blind return-oriented programming (BROP). Under certain conditions, this attack allows a return-oriented programming attack against previously unknown binaries. We precisely enumerate the assumptions for a successful BROP attack to take place. We analyze prerequisite knowledge to perform a BROP attack, including the need to exploit a stack-based buffer overflow. In particular, we examine the types of buffer-handling functions and canaries that may render these functions useless for exploitation purposes. We survey network service binaries, to examine how often different BROP requirements are satisfied in real software, including the presence of certain gadgets and the behavior on crashes.

We find if an optimized attack fails, a "first principles" BROP attack is unlikely to succeed. Our survey shows that certain required gadgets are rare, limiting a first principles attack. We show the presence of required gadgets fluctuates with binary version number and build conditions. The majority of the services we survey do not appear vulnerable to BROP due to missing gadgets or re-randomization on crash. We suggest some ameliorations that may further limit the applicability of this attack.

14. SUBJECT TERMS	15. NUMBER OF PAGES
BROP, return-oriented programming, ROP, return-to-libc, implementation disclosure attacks	75
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

EVALUATING THE GENERALITY AND LIMITS OF BLIND RETURN-ORIENTED PROGRAMMING ATTACKS

Lawrence Keener
Civilian, Vista Research
B.S., California State University of Monterey Bay, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2015

Approved by: Mark Gondree
Thesis Advisor

Chris Eagle
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

We consider a recently proposed information disclosure vulnerability called blind return-oriented programming (BROP). Under certain conditions, this attack allows a return-oriented programming attack against previously unknown binaries. We precisely enumerate the assumptions for a successful BROP attack to take place. We analyze prerequisite knowledge to perform a BROP attack, including the need to exploit a stack-based buffer overflow. In particular, we examine the types of buffer-handling functions and canaries that may render these functions useless for exploitation purposes. We survey network service binaries, to examine how often different BROP requirements are satisfied in real software, including the presence of certain gadgets and the behavior on crashes.

We find if an optimized attack fails, a "first principles" BROP attack is unlikely to succeed. Our survey shows that certain required gadgets are rare, limiting a first principles attack. We show the presence of required gadgets fluctuates with binary version number and build conditions. The majority of the services we survey do not appear vulnerable to BROP due to missing gadgets or re-randomization on crash. We suggest some ameliorations that may further limit the applicability of this attack.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ASLR	address space layout randomization
BROP	blind return-oriented programming
CVE	common vulnerabilities and exposures
GCC	GNU compiler collection
GOT	global offset table
HadROP	Hardware-assisted Detection of Return-oriented Programming
JIT-ROP	just-in-time return-oriented programming
PIE	position-independent executable
PLT	procedure linkage table
ROP	return-oriented programming
SBIR	small business innovation research
SCHSIM	stochastic compiler hacks as software immunization mechanisms
SOAP	Simple Object Access Protocol
SSP	stack smashing protection

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

We thank Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh for writing the article that provided the basis of this research project and publishing their proof of concept attack code. In particular, we would like to thank Dan Boneh for correspondence and assistance with understanding BROP during this thesis. We are especially grateful to Ali Mashtizadeh for taking the time to assist during the project, providing guidance in understanding and using Braille.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Cyber attacks have evolved over time in order to counter ever-developing cyber defense methods. Defenses such as address space layout randomization (ASLR) and non-executable stacks have forced attackers to find new ways to deploy attacks. A new wave of recently deployed attacks employ memory disclosure side-channels to leak address layout information.

Bittau *et al.* describe one such approach, introducing the blind return-oriented programming (BROP) attack [1]. A traditional return-oriented (ROP) attack requires advance access to the binary to discover usable ROP gadgets. In comparison, BROP requires no prior access to the binary, and demonstrates the ability to perform a ROP attack under a strictly weaker set of assumptions. We analyze the requirements for this new attack and critically review if and how real-world binaries satisfy these prerequisites. We do this in consideration of views that BROP is dangerous because it "requires minimal knowledge and can easily be distributed to amateurs" [2]. To our knowledge, there has been no prior analysis of the potential impact of BROP beyond its original description.

The primary contributions of this project are:

- We enumerate the precise prerequisites for a BROP attack and survey real-world services to investigate if they satisfy these prerequisites.
- We find that several required gadgets (i.e., `syscall`, `pop rax`, and `pop rdx`) appear to be a limiting factor for a first principles attack BROP attack. In particular, the `rax` gadget is present in less than 30% of our surveyed binaries, and the `syscall` gadget in less than 20% of binaries. We believe this could impact BROP's effectiveness nearly as much as scarcity of `rdx` gadgets, acknowledged by Bittau *et al*.
- As a result, we suggest that, if the "optimized" BROP attack fails, a first principles BROP attack appears unlikely to succeed and is a poor fall-back option.
- We show the gadget composition for a binary changes with the version and build environment, making it difficult to summarize the frequency of a gadget's appearance

in a service without access to the binary. For example, in some cases, the binary for Redis lacks the gadgets necessary for a first principles attack and, in other cases, has them.

- We compare our surveyed binaries to the target used for BROP's proof-of-concept attack. We find that these binaries have weaker PLT signatures, making employing an optimized attack more difficult.
- We re-investigate the claim of Bittau *et al.* that "most of the PLT entries will not cause a crash regardless of arguments because they are system calls that return EFAULT on invalid parameters." While we find that 40% of PLT entries for the nginx binary are system calls, this number can be as low as 5% in the other services.
- We find the majority of the services we survey do not appear vulnerable to BROP, due to missing gadgets or re-randomization on crash.
- We survey and suggest some controls to reduce the threat of BROP attacks.

1.1 Relevance

The Navy has a demonstrated interest in both cyber attack and defense technologies. For example, one Small Business Innovation Research (SBIR) project sponsored by the Navy is focused on developing software that can proactively detect compromise and crash safely, instead of continuing in an unsafe mode, during or after an attack [3]. This project specifically mentions ROP attacks as a motivating case for the new defense technology. Another Navy project attempts to harden systems against ROP-style attacks, called Stochastic Compiler Hacks as Software Immunization Mechanisms (SCHSIM). Fugate and Petrie discuss SCHISM, describing a technical solution for creating simulated software diversity in order to prevent software monocultures [4]. The idea for SCHISM is to diversify binaries that are using the same service, to ensure that an attack on one system could not work on another system the exact same way, since the binary on each system would be different for the same service.

Investigating and understanding the limits of BROP could inform these Navy efforts. For example, the automated diversity created by SCHISM motivates a BROP attack, which operates without a priori knowledge of a binary and works even in the presence of unexpected diversity. Our project suggests some weaknesses to the effectiveness of BROP which may

be leveraged to construct more effective defenses.

1.2 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we present background on ROP and BROP necessary to understand our analysis. In Chapter 3, we present an enumerated list of the assumptions necessary for conducting a successful BROP attack, expanding on those discussions in the paper originally describing BROP. In Chapter 4, we analyze these assumptions and survey real-world services to see how often these assumptions are satisfied. In Chapter 5, we present approaches— based on the enumerated assumptions, analysis findings, and existing literature— to harden systems against BROP attacks.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

In this chapter, we briefly summarize both return-oriented programming (ROP), and the Blind ROP technique introduced by Bittau *et al.* [1].

2.1 Return-Oriented Programming

Return-oriented programming (ROP) is an attack method allowing an attacker to execute arbitrary code. The method is effective even in the presence of various stack smashing protections (SSP), such as non-executable memory protection. Rather than write payload code to memory and execute this injected exploit code, ROP uses control of the program counter to create new program control flow, chaining together segments of existing code already accessible from the target program's address space [5]. The first version of this style of attack focused on code in the C library, and was called *return-to-libc*. Later versions of this attack demonstrated that the entire binary could be leveraged in this manner.

The segments of code used by the attacker to implement their exploit are called *gadgets*. ROP *gadgets* are "short blocks placed on the stack that chain several of instruction sequences together" [6]. Discovering useful gadgets is a necessary pre-requisite step in launching a successful ROP attack. Shacham describes an efficient algorithm for static analysis of x86 binaries to discover gadgets [6]. The need for mining binaries to discover useful gadgets has led to the development of several "gadget finding tools." Example tools include ROP Gadget Finder [7], Ropper [8], RP++ [9] and MSFrop [10] (part of the Metasploit framework [10]). ROPSHELL [11] is a web-based gadget finding tool where users upload a target binary and it reports back an analysis of gadgets.

2.2 Blind ROP

Bittau *et al.* introduce a type of ROP attack, called *blind ROP* (BROP). This attack weakens the requirement that access to the target binary be prerequisite to a ROP-style attack. The attack leverages a novel technique of exploiting a timing side-channel to leak infor-

mation about the effect of an exploit action. BROP falls into a category of similar, recent attacks leveraging information leaks to circumvent ASLR and memory protection, called *implementation disclosure attacks*. Mohan observes that "since finding and closing all information leaks is well known to be prohibitively difficult and often intractable for many large software products, these attacks constitute a very dangerous development in the cyber-threat landscape; there is currently no well-established, practical defense" [12].

The BROP attack requires apriori knowledge of a stack vulnerability and how to trigger it. Using this vulnerability, the attacker can remotely discover gadgets through a trial-and-error discovery process. This process depends on a *stop gadget*, which is any gadget that causes the service to pause or block in a manner that that can be differentiated from a crash. The feedback from the stop gadget creates a timing side-channel, allowing the remote attacker to procedurally probe the target address space and discover other gadgets, eventually performing a `write` syscall. When complete, the code portion of the target binary is sent to the remote attacker using this syscall, thereby facilitating local analysis of the binary and allowing a second-stage ROP attack.

As described, BROP is generic enough to work even when both memory protection and ASLR are employed. As part of gadget discovery, however, the target service crashes repeatedly to elicit timing behavior; thus, it must be the case that the service restarts on crash without re-randomizing its address space. For canary circumvention, Bittau *et al.* describe a method called *generalized stack reading*. This technique involves using another trial-and-error type discovery mechanism, placing data onto the stack byte-by-byte. The remote attacker observes if the service crashes or not, to determine whether the input had overwritten the stack with a valid canary value. When a no-crash state is observed, the data written to the stack has overwritten the canary with a valid value. Once the canary is discovered, the BROP attack proceeds with gadget discovery, each time overwriting the canary value exactly and the return address.

Bittau *et al.* provide a proof-of-concept attack implementation for 64-bit Linux binaries, called Braille (see Appendix A). This proof-of-concept demonstration targets `nginx` version 1.4.0, which has a known stack vulnerability (CVE-2013-2028) that "triggers an integer signedness error and a stack-based buffer overflow" [13].

2.2.1 Related Attacks

JIT-ROP [14] is another information disclosure attack that is able to overcome fine-grained ASLR. Both JIT-ROP and BROP attempt to find gadgets in the binary in order to develop a ROP attack, exploiting an information leak of some type. Unlike BROP, JIT-ROP is not an interactive attack that proceeds adaptively based on crash/no-crash behavior: the attacker sends a script that discovers gadgets and creates an attack to deliver. Additionally, JIT-ROP requires two different vulnerabilities (a stack and heap vulnerability) be known prior to attack, compared to BROP's requirement that a stack vulnerability be known.

2.2.2 Prior BROP Analysis

The original paper describing BROP was published in 2014 at the IEEE Symposium on Security & Privacy, one of the top-tier conferences in the security field. It has become one of the most widely-cited papers from that year's event, and generally accepted as an impactful result. Many cite it as proof of concept for exploiting a type of timing side channel, for exploiting fault analysis, for circumventing SSP and ASLR, etc. The most common criticism is that the attack is noisy, (e.g., it can only "be conducted against systems where repeated crashes should go unnoticed" [15] or "potentially detected by mechanisms that monitor for an abnormal number of program crashes" [16]). To our knowledge, no work has evaluated or surveyed other possible BROP limitations, such as those considered in this work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Assumptions

In this chapter, we enumerate the attack assumptions required by Bittau *et al.* for launching a blind ROP attack [1]. These attack assumptions are effectively pre-requisites for a successful BROP attack. Most assumptions are identified explicitly in the original paper or implicitly from context. We enumerate these precisely, to discuss them further in Chapter 4.

3.1 Goal

The goal of a BROP attack is to access the binary of a remote target service when the attacker has no other means of obtaining access to the binary. This is the simplest, general goal as it allows one to bootstrap a follow-on ROP exploit with full knowledge of the ROP gadgets made available by the binary.

Bittau *et al.* summarize the utility of BROP techniques compared to ROP techniques, characterizing BROP as useful in many scenarios where ROP is unable to be employed directly, with the exception of PIE binaries where crashes cause the address space to be re-randomized [1, Fig. 4]. Traditional ROP attacks require knowledge about canaries, linking dependancies and access to the target service binary; BROP attacks work in the absence of this knowledge.

3.2 Assumptions

We summarize the stated requirements upon which a successful BROP attack is contingent, and attempt to state any implicit requirements not explicitly stated earlier by Bittau *et al.* [1].

Assumption 3.1 (Execution Environment)**.** *The target service must be executing in some 64-bit x86 Linux environment.*

Bittau *et al.* briefly discuss the possibility of extending BROP to other execution environments but, as stated, the techniques for the BROP attack have not been generalized to other environments.

Assumption 3.2 (Vulnerability Foreknowledge)**.** *The target service must be exploitable via some known stack vulnerability. The attacker must have knowledge of how to exploit this vulnerability to control of the value of the instruction pointer (a pre-requisite for any ROP attack).*

Bittau *et al.* briefly discuss the possibility of bootstraping BROP attacks from other scenarios such as heap exploits but, as stated, the techniques for the BROP attack have not been generalized to these.

Assumption 3.3 (Crash-Restart Execution)**.** *The target service must restart after a crash, without re-randomizing its address space.*

Assumption 3.4 (Observable Status)**.** *The attacker must be able to observe the status of the service (crash or no-crash) during the attack and receive the final output of the exploited service.*

When considering remote attackers, the simplest interpretation of this requirement is for the target service to be a network service. In the case that the service is sandboxed without network access but remotely exploitable via some intermediary or proxy service, it must be the case that the service's status is inferable and that some mechanism allows the attacker to receive the final output of a successful attack. At a minimum, it is necessary for the target service to have permissions to employ the `write` syscall (expanded upon via Assumptions 3.5–3.9).

Assumption 3.5 (Full RDI Control)**.** *The target binary must contain a gadget allowing the attacker to control the `rdi` register's contents.*

Assumption 3.6 (Full RSI Control)**.** *The target binary must contain a gadget allowing the attacker to control the `rsi` register's contents.*

Assumption 3.7 (Full RAX Control)**.** *The target binary must contain a gadget allowing the attacker to control the `rax` register's contents.*

Assumption 3.8 (Weak RDX Control)**.** *It must be the case that the `rdx` register holds (or can be populated by the attacker to hold) a positive, non-zero value.*

Assumption 3.9 (Reachable syscall)**.** *The target binary must contain a reachable `syscall`.*

```
pop rdi; ret   // set file descriptor argument
pop rsi; ret   // set buffer
pop rdx; ret   // set length
pop rax; ret   // set syscall number for write call
syscall
```

Figure 3.1: Example gadget chain satisfying Assumptions 3.5–3.9, leading to an invocation of the `write` syscall.

The simplest gadget satisfying each of Assumptions 3.5–3.7 is a `pop` of the target register followed by a `ret`. Assumption 3.8 can also be satisfied this way, yielding complete control over `rdx`'s contents; alternatively, any function which populates `rdx` with a non-negative value may be employed. For example, `strcmp` places the length of the string being compared into this register. A series of gadgets simultaneously satisfying Assumptions 3.5–3.9 are listed in Figure 3.1.

Assumption 3.10 (Stack-Readable Canaries)**.** *Canary protections for the target service must be circumventable via generalized stack reading.*

Bittau *et al.* describe a method known as *generalized stack reading* for leaking canaries, employed by BROP as a method of circumventing canary protections. They comment "there are a few subtleties that apply to generalized stack reading" and Assumption 3.10 codifies the set of requirements implied by those subtleties.

3.3 BROP Optimizations

Bittau *et al.* propose a series of optimizations requiring fewer, but more specific, gadgets to enhance the speed of the BROP attack by reducing the number of explicit gadgets that must be discovered. They suggest finding two library calls, guessable by trial-and-error via calling into the PLT, and a single *BROP gadget* (see Figure 3.2). Misaligning this BROP gadget at different offsets yields two useful gadgets: a `pop rdi` gadget and a `pop rsi` gadget. This is a single gadget satisfying two assumptions simultaneously (Assumptions 3.5 and 3.6). The `write()` call replaces the need to find `pop rax` gadget and a `syscall` since these gadgets are collected in order to make a write call. This is a single function call satisfying two assumptions simultaneously (Assumptions 3.7 and 3.9). The `strcmp()` call

BROP Gadget:

```
pop rbx; pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
```

Call to PLT:

```
call x // <write@plt>
```

Call to PLT:

```
call y // <strcmp@plt>
```

Figure 3.2: Set of gadgets simplifying Assumptions 3.5–3.9, based on [1, Fig. 7].

yields control of the `rdx` register (Assumption 3.8), but this is possibly unnecessary if that register can be relied upon to hold a sane value during the attack.

CHAPTER 4:
Analysis

In this chapter, we consider the assumptions introduced in Chapter 3 individually and in combination, analyzing the likelihood these assumptions can be satisfied in a real-world context. The five general areas we will consider are: finding the initial stack vulnerability; using stack reading to circumvent canaries; likelihood the target binary is of the correct composition to employ the attack; ability of an optimized attack to be deployed (i.e., locating the PLT; and commonness of services using a forking model that permits the attack).

4.1 Finding the Vulnerability

There is a possible tension between BROP's objectives (i.e., to attack a service without any foreknowledge of the binary) and its requirement for foreknowledge of a vulnerability (Assumption 3.2). We investigate the process in which service vulnerabilities may be discovered, to consider if vulnerability discovery typically presumes access to the binary.

Hutchins *et al.* describe the *cyber kill chain* as "a systematic process to target and engage an adversary to create desired effects" [17], defining it as a seven phase process for intrusion. The seven phases they describe are reconnaissance, weaponization, delivery, exploitation, installation, command and control, and actions on objectives. In this exploit framework, the process of initial discovery of a stack vulnerability prerequisite to a BROP attack would be considered part of the *reconnaissance* phase. In this phase, one decides what the target is and gathers the necessary information about the target to perform an attack. We find there are at least three reasonable scenarios for reconnaissance that do not presume access to the target binary:

Discovery with assistance. In this scenario, the attacker uses some third party who provides the required information. For example, consider an insider with direct knowledge of a stack vulnerability in a closed-source service who may be unable to exfiltrate the target binary itself. It is difficult to imagine a reasonable scenario where the insider may be able to provide good knowledge in one realm (i.e., presence of the

vulnerability and how to exploit it), but incomplete knowledge in another (i.e., the binary itself). Such a scenario may be possible, however, if divisions across the development team prevents the insider from direct access to the complete binary, or if controls are in place that prevent information transfer of the binary by unauthorized staff. As a more reasonable example, consider an intermediary selling an exploit but who has, otherwise, incomplete knowledge about the target system. This "as is" knowledge may be incomplete for the purposes of the desired attack, but may be leveraged to bootstrap the BROP exploit.

Discovery using an identical binary. In this scenario, the attacker may be able to interact with the target binary, but be otherwise unable to study it. Such access opens the possibility of using other tools to assist in vulnerability discovery, such as remote fuzzing. System monitoring likely prevents effectively fuzzing live targets during reconnaissance, as its particularly "noisy" and increases the risk of the attacker being noticed, stopped or caught early in the attack (which is, itself, noisy). Instead, this scenario makes most sense when the attacker has local access to an identical, duplicate system they may use for reconnaissance. For example, consider some closed-source embedded system from which the attacker is unable to extract the binary, but with which the attacker may interact. Otherwise, it is difficult to imagine a scenario in which an attacker has access to an identical target (for discovery of the stack vulnerability) but is otherwise compelled to use a blind attack.

Discovery using a non-identical binary. In this scenario, the attacker can obtain a notional binary for the service, but not necessarily one in an identical configuration as deployed on the target system. Examples include building a open-source service without knowledge of the compiler flags, or details of build options or exact dependencies associated with the target binary. In this scenario, the attacker knows the target and may discover the presence and method of exploitation for the stack vulnerability, generally. This information may be as simple to discover as using a vulnerability database. Some databases, like the National Vulnerability Database[1], provide resources to see if the target has any known stack buffer overflow vulnerabilities; other databases, such as Exploit Database [2], offer information on how to trigger

[1] https://nvd.nist.gov/

[2] https://www.exploit-db.com/

and exploit the vulnerability.

We acknowledge that all the above scenarios additionally require that, once the vulnerability is found, the remainder of the BROP assumptions are also satisfied. The above, however, provides some context suggesting the possible applicability of BROP, or at least suggests some caveats on the idea of a "blind" attack.

4.2 Canary Circumvention

Assumption 3.10 requires stack protections be circumventable via stack reading. A natural question is: what protections *can* be circumvented by stack reading? In this section, we attempt to characterize existing stack protections in terms of their susceptibility to generalized stack reading.

Canaries are a form of stack protection in which data is placed between the return address and any buffers in the stack frame. The idea is that attacks writing to some buffer in an attempt to overwrite the return address will also overwrite the canary. This protection mechanism is inserted by the compiler, along with canary-checking logic which attempts to detect if a canary has been overwritten before the function returns. The technique was first described by Cowan *et al.* for the StackGuard GCC patches [18]. Depending on the details of how elements are laid out on the stack, several different methods may be available to bypass the protection afforded by canaries. The following are some methods described in the open literature:

Overwriting the Canary. This method overwrites the canary exactly, so canary spoilage cannot be detected by the stack protection logic. Generalized stack reading is an implementation of this approach.

Exploiting Canary-less Functions. When canaries are employed, they may not protect all functions. For example, the default for GCC v4.8 is to employ the `-fstack-protector` flag [19], where canaries are inserted to protect "functions that call `alloca` and functions with buffers larger than 8 bytes," per the `man` page. So an attacker may exploit a 5-byte buffer, in the absence of other buffers local to the frame, as there should be no canary inserted. Some canary insertion logic protects buffers at all locations (e.g., the `-fstack-protector-all` flag).

Exploiting Pointers Before the Canary. This method relies on overwriting other pointers in the current stack frame. Bulba and Kil3r describe a method of overwriting the pointer to a buffer before exploitation, to skip over the canary and overwrite just the return address [20]. Similar techniques can be used to overwrite entries in the GOT to invoke payloads without modifying the return address [21].

Exploiting Exception Handlers. By overwriting exception handlers and triggering an exception, it is possible to invoke a payload before canary-handling logic is even invoked. Similar techniques allow overwriting `atexit` handlers to invoke a payload immediately after canary-handling logic is invoked [21].

Exploiting Some Other Vulnerability. For example, exploiting format string vulnerability allows to write any value any place. This, however, does not qualify as a buffer overflow and is out of the bounds of the assumed set-up for a BROP attack.

4.2.1 Canary Types

To consider the effectiveness of stack reading for overwriting a canary, we review the variety of canaries available for stack protection. The following is a list of canary types that we consider:

Terminator Canary. This canary is a special constant (`0x000aff0d`) that includes values terminating many string manipulating functions: null (`0x00`), linefeed (`0x0a`), end-of-file (`0xff`) and newline (`0x0d`) characters [22]. The intention of this canary is to cause those functions employed to copy data into a buffer to terminate, when they attempt to overwrite the canary. This type of canary is available in all versions of StackGuard [23].

Null Canary. This canary uses a constant sequence of null bytes (`0x00000000`), and can be considered a special type of terminator canary [24]. To our knowledge, no current stack protection implementations use this type of canary.

Random Canary. This canary is a 64-bit random value chosen when a program executes.[3] This type of canary is available in StackGuard prior to v2.0.1 and all versions of ProPolice.

[3] In some versions of ProPolice, when randomization is turned off or not available this defaults to a type of terminator canary with value `0xff0a0000` [25], [26].

Table 4.1: Buffer-handling function families

Family	Terminator	Operators
Null Terminating	`0x00`	stpcpy, stpncpy, strcpy, strncpy, strcat, strncat, [v]sprintf, [vf]scanf
Line Terminating	`0x0a`	scanf, gets, fgets, getc, getchar, fgetc
Custom Terminating	user-specified	memccpy
Wide-Null Terminating	`L'\0'`	wcpcpy, wcpncpy, wcscpy, wcsncpy
No Terminator	none	bcopy, memcpy, memmove, mempcpy, wmemcpy, wmempcpy

Random XOR Canary. This canary is formed by xor-ing the return address with a random value, performing a "one-time-pad encryption" of the return address [23]. Canary-checking logic verifies the canary matches the return address xor the random value before returning; thus, if the return address changes but the canary does not (i.e., it is skipped or overwritten with an identical value) the attack will be detected. This type of canary is available in StackGuard v1.21, but is no longer supported. PointGuard provides a similar protection mechanism for Windows by encrypting return pointers [27].

4.2.2 Effectiveness of Stack Reading

To evaluate the effectiveness of stack reading to overwrite canaries, we consider those string operators that would be exploited to write data to a buffer. We group functions into families based on their terminating behavior: Null, Wide-Null, Line and Custom Terminating families (see Table 4.1). For example, strcpy is unable to write bytes following a null byte into the target buffer, since the function terminates on the first `0x00` encountered. Thus, overwriting any canary containing an intermediate null byte cannot be accomplished by exploiting a single invocation of the strcpy function; however, a loop using multiple invocations of strcpy may have the correct properties to write the following bytes. In general, such features are not implied by Assumption 3.2, which requires the attacker is aware of the existence and method to exploit the overflow in general, without prior knowledge of which compiler-inserted canary protections may be used.

When a terminator canary is used for stack protection, stack reading cannot be employed in the case that the known stack vulnerability (Assumption 3.2) requires exploiting a func-

Table 4.2: Stack reading effectiveness against different canary types

Canary Type	Value	Overwritable
Terminator Canary	`0x000aff0d`	◓
Random Canary	64-bit random value	◓
Random XOR Canary	64-bit "encrypted" return pointer	◔

tion from the Null or Line Terminating families. Similar problems exist for the random canary, as there is a 1/256 chance that a random byte will be a terminating character for the Null, Line and Custom Terminating families. For 64-bit canaries, the total probability that some terminating character exists in a random canary becomes 1/32. Thus, we consider terminator canaries to partially prevent stack reading, and random canaries to provide low-probability partial prevention against stack reading. Against a random XOR canary, stack reading has no significant probability of success as long as the canary or random value remain secret: in general, it is not possible to overwrite the canary via stack reading and also modify the return pointer. These observations are summarized in Table 4.2.

We find that generalized stack reading cannot be used to reliably circumvent all types of canaries. For example, buffer exploits requiring misuse of function families with terminating values cause stack reading to have a non-negligible probability of failure for both random and terminator canaries, depending on the target function being exploited.

4.3 Binary Composition Survey

Assumptions 3.5–3.9 are requirements on the binary composition of the service which are prerequisites for performing a BROP attack against that target. In this section, we survey a variety of network services to determine if these assumptions are commonly satisfied. In particular, among the selected target services we survey, we resolve: (i) if the essential BROP binary composition requirements appear to be satisfied (i.e., if a "first principles" attack is possible); (ii) if the binary composition requirements appear to be satified to perform an optimized attack (i.e., the attack described by Bittau *et al.* requiring fewer but more complex gadgets).

We perform our survey by selecting a variety of target services written in C or C++ available on Linux (see Table 4.3). Our survey data is gathered following two methods:

Table 4.3: Services selected for the binary composition survey.

Service	Type
apache	web server
axis2c	web server
cyrus	mail server
dovecot	mail server
gsoap	web server
jabberd2	XMPP server
lighttpd	web server
mongoDB	database
nginx	web server
openswan	IPsec daemon
postfix	mail server
postgreSQL	database
redis	database
sendmail	mail server
tengine	web server
wu-ftp	FTP server

Pre-built target binaries. Survey the gadgets of each service binary as available by default on Ubuntu (i.e., the most recent version as distributed under Ubuntu 14.04.01 via the `apt-get` package manager).

Compiled target binaries. Survey the gadgets of each service binary as it is built from source on different platforms: (i) on Ubuntu 14.04.01 using GCC 4.8.4, (ii) on Fedora 22 using GCC 5.1.1 and (iii) on CentOS 7.1.1503 using GCC 4.8.3.

When building from source, we survey the past ten releases (when available) and build using all default options following their build instructions. These two data sources highlight how different compiler versions and different target platforms may impact the binary composition with respect these gadgets.

The tool we use for surveying the gadgets in each binary is `rp++`, an open-source gadget-finding tool that works with 64-bit binaries. We use the tool's option to limit the found gadget size to count gadgets precisely and avoid counting the same gadget multiple times. Without this option, if the binary contained `pop rbx; pop rax; ret;` and one searched for the gadget `pop rax; ret;`, the default behavior is for the tool to discover two gadgets:

```
$ ./rp-lin-x64 -f a.out -r 2 |grep "pop r14 \; pop r15 \  ret \;"
0x00450f84:  pop r14 ; pop r15 ; ret ; (1 found)
0x0045157d:  pop r14 ; pop r15 ; ret ; (1 found)
0x00451a41:  pop r14 ; pop r15 ; ret ; (1 found)
$ ./rp-lin-x64 -f a.out -r 2 |grep "pop rdi \  ret \;"
0x00450f87:  pop rdi ; ret ; (1 found)
0x00451580:  pop rdi ; ret ; (1 found)
0x00451a44:  pop rdi ; ret ; (1 found)
```

Figure 4.1: Example rp++ output, showing an rdi gadget at an offset of each BROP gadget.

Table 4.4: Survey of /usr/bin and /usr/sbin, percentage of binaries containing each gadget.

OS	Binaries	syscall (%)	rax (%)	rsi (%)	rdi (%)	rdx (%)
Ubuntu	1144	15.65	26.22	61.71	97.2	8.83
Centos	1810	17.51	19.56	63.87	100.0	10.0
Fedora	1861	19.13	21.17	64.05	100.0	12.31

one gadget containing pop rax and a larger gadget containing both pop rax and pop rbx. It is also important to note rp++ has the ability to discover gadgets at any alignment (see Figure 4.1) and restricts itself to discovering gadgets in portions of the binary that would be loaded into memory marked executable. These features prevent us from discovering false-positive gadgets, from double-counting gadgets and from missing gadgets.

4.3.1 Survey: First Principles BROP Attack

The "first principles" BROP attack is based only on Assumptions 3.5–3.9, requiring five gadgets. We survey both pre-built target binaries (see Table 4.5) and compiled target binaries for Ubuntu (see Table B.1), CentOS (see Table B.2) and Fedora (see Table B.3) for the presence of these gadgets. As a point of comparison, we also survey the binaries under /usr/bin and /usr/sbin for each distribution (see Table 4.4).

While the total number of each gadget vary across versions of the same service, and across the same version compiled using different tools (i.e., on different platforms), the significant trends remain fairly stable. Among the services we survey, we find that rdx gadgets were relatively rare, followed in scarcity by syscall and rax. There was no strong correlation between the size of the binary and the number of rdx, rax and syscall gadgets: larger

Table 4.5: Gadgets for a "first principles" BROP attack, pre-built services on Ubuntu 14.04.01.

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
apache	2.4.7	637528	0	122	1	189	0
openswan	2.6.38	1031824	14	59	14	188	0
postgreSQL	9.3.9	5633016	64	17	23	6	4
axis2c server	1.6.0	14648	0	0	0	1	0
cyrus	2.4.17	198000	0	7	0	20	0
wu-ftpd	2.6.1	548343	0	19	0	54	0
gsoap	2.8.16	455352	2	20	1	34	3
postfix	2.11.0	14280	0	0	0	1	0
mongoDB	2.4.9	2158112	289	246	4	709	30
nginx	1.4.6	873176	1	175	2	383	0
lighttpd	1.4.33	198200	0	43	0	49	0
dovecot	2.2.9	79920	0	11	0	19	1
redis	2.8.4	635248	2	119	1	220	0
sendmail	8.14.4	819072	11	59	1	243	0
jabberd2 c2s	2.2.17	167000	0	24	0	63	0
tengine	2.1.1	6091896	1	85	0	267	0

binaries do not make it more likely for these gadgets to appear, or appear more frequently. In particular, we find at most 5 `rdx` gadgets, regardless of binary size (MongoDB being a notable exception to this). In contrast, there is an apparent correlation between the size of the binary and the other gadgets; we find ~3 `rsi` gadgets and ~6 `rdi` gadgets for every 1 KB.

Given the scarcity of some gadgets, the prerequisites for a "first principles" attack are not satisfied for 80% of the pre-built binaries we survey. For each service, we try to build the 10 most recent versions of the service from source on Ubuntu, CentOS and Fedora and survey if the attack prerequisites are satisfied in *any* of those versions (see Table 4.6). Ignoring MongoDB (which satisfies the prerequisites in all experiments), we find 98% of binaries we survey on Ubuntu do not satisfy the prerequisites (93% for CentOS, 81% for Fedora).

There are some interesting exceptions to the trends we observe. In particular, MongoDB exhibits an unusually high number of all pre-requisite gadgets. Redis shows strong variation across our experiments: on Ubuntu, every version lacked `rdx` gadgets; when built on

Table 4.6: Versions of compiled target binaries satisfying the prerequisites for a "first principles" attack, totals by platform (see Appendix B.1–B.3 for details).

Service	Ubuntu	CentOS	Fedora
redis	0	3	10
postgreSQL	0	0	0
mongoDB	10	10	10
lighttpd	0	0	0
nginx	0	0	0
tengine	1	1	1
dovecot	0	0	0

CentOS, three versions produced this gadget; when built on Fedora, all versions of Redis contained this gadget (in fact, all gadgets). Tengine demonstrates a similar build variability: across all versions, some prerequisite gadget is missing when compiled on each platform, except version 1.4.6 when built on CentOS and version 1.5.2 when built on Fedora.

In general, the relative scarcity of some gadgets make a first principles attack unfeasible for many of the services we survey. The optimized attack we survey next, however, reduces the number of primitive gadgets required, substituting new requirements related to the presence of specific PLT entries.

4.3.2 Survey: Optimized Attack

The BROP optimization described by Bittau *et al.* obviates the need for certain explicit gadgets, substituting the use of select functions and a specialty gadget (see Figure 3.2). We survey both pre-built target binaries (see Table 4.7) and compiled target binaries for Ubuntu (see Table B.4), CentOS (see Table B.5) and Fedora (see Table B.6) for the prerequisites for this optimized attack. Again, as a point of comparison, we also survey the binaries under `/usr/bin` and `/usr/sbin` for each distribution (see Table 4.8).

Of the pre-built binaries we survey, we find 75% satisfy the prerequisites for the optimized BROP attack. Of the binaries compiled from source, we find 94% satisfy the prerequisites for the optimized BROP attack, in all versions and regardless of target platform surveyed. When comparing to earlier results, we see nearly all services lacking prerequisites for a "first principles" attack have the requirements for the optimized BROP attack. Bittau *et*

Table 4.7: Gadgets for an optimized BROP attack, pre-built services on Ubuntu 14.04.01.

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write found
apache	2.4.7	637528	0	True	True
openswan	2.6.38	1031824	193	True	True
postgreSQL	9.3.9	5633016	1	True	True
axis2c server	1.6.0	14648	1	True	False
cyrus	2.4.17	198000	20	True	True
wu-ftpd	2.6.1	548343	52	True	True
gsoap	2.8.16	455352	35	True	False
postfix	2.11.0	14280	1	False	False
mongoDB	2.4.9	2158112	730	True	True
nginx	1.4.6	873176	399	True	True
lighttpd	1.4.33	198200	49	True	True
dovecot	2.2.9	79920	19	True	True
redis	2.8.4	635248	229	True	True
sendmail	8.14.4	819072	259	True	True
jabberd2 c2s	2.2.17	167000	64	True	True
tengine	2.1.1	6091896	289	True	True

Table 4.8: Survey of `/usr/bin` and `/usr/sbin` for optimized attacks, percentage of binaries containing each gadget.

OS	Binaries	strcmp (%)	write (%)	BROP (%)
Ubuntu	1144	65.65	68.53	97.12
Centos	1810	67.13	69.06	100.0
Fedora	1861	63.73	71.04	100.0

al. state that if "the program or one of its libraries does not use [strcmp] in which case the attacker can perform a 'first principles' attack." Based on our survey, however, we believe it to be unlikely that a "first principles" attack has a greater chance of succeeding than the optimized attack. We also leave to future work to test the validity of the assumption that different versions of strcmp will actually change the value in rdx.

4.4 Finding the PLT

Finding the PLT section of a binary is a necessary step for the optimized BROP attack, to locate the function pointers to call `strcmp` and `write`. In this section, we evaluate the requirements identified by Bittau *et al.* for locating the PLT for the optimized attack. It is argued that the PLT has a unique signature which can be detected via probing: each entry is 16 bytes apart, where "most of the PLT entries will not cause a crash regardless of arguments because they are system calls that return EFAULT on invalid parameters"; thus, the PLT can be located with "great confidence if a couple of addresses 16 bytes apart do not cause a crash, and can verify that the same addresses plus six do not cause a crash" [1].

We survey the 16 services previously identified for our survey (see Section 4.3), scanning PLT entries and noting the maximum number of consecutive syscalls (see Table 4.9). As a point of comparison, we also investigate `nginx` 1.4.0, as the BROP proof-of-concept exploit has been demonstrated against this binary. For `nginx` 1.4.0, we find the maximum number of consecutive syscalls is five; this is larger than in any other service we survey. Further, syscalls account for 44% of the PLT entries for this version of `nginx`; the other services we survey have a much lower percentage of syscall entries for their PLTs. Based on this survey, we believe it is not possible to conclude that most PLT entries are syscalls, or that a large contiguous segment of the PLT tends to be populated by syscalls.

The BROP proof-of-concept attack code, however, require only a single (apparent) PLT entry to decide the PLT has been located. This decision is based on the non-crashing behavior at the 16-byte-aligned address, and at the address 6 bytes later (the location of the resolver code). If this probe fails, the code jumps to offset 0x10 * 30, skipping 30 entries. Thus, if an actual PLT entry fails this check, 29 PLT entries may be skipped during this scan. Such a failed probe may result due to entries for `memcpy` or other library calls which crash on invalid input. For a service like `axis2c`, where there are only 38 total PLT entries, this may prove significant, especially since only 5% percent of PLT entries are syscalls (i.e., the PLT may never be discovered following this scanning approach). We defer for future work any analysis of the more basic claim that most syscalls do not crash on invalid input.

Table 4.9: Survey of syscalls in PLT, prebuilt services on Ubuntu 14.04.01.

Service	Version	Total PLT Entries	Total syscalls	% syscalls	Max Consecutive syscalls
apache2	2.4.7	2880	19	0.7	2
openswan	2.6.38	3147	43	1.4	3
postgreSQL	9.3.9	184	15	8.2	3
axis2c server	1.6.0	38	2	5.3	1
cyrus	2.4.17	1324	52	3.9	4
wu-ftp	2.6.1	114	26	22.8	3
gsoap	2.8.16	47	2	4.3	1
postfix	2.11.0	73	5	6.8	1
mongoDB	2.4.9	782	38	4.9	3
nginx	1.4.6	360	65	18.1	4
lighttpd	1.4.33	228	46	20.2	3
dovecot	2.2.9	201	27	13.4	2
redis	2.8.4	213	49	23.0	4
sendmail	8.14.4	113	9	8.0	2
jabberd2-c2s	2.2.17	355	25	7.0	2
tengine	2.1.1	321	65	20.2	2

4.5 Forking Model

Assumption 3.3 requires the target service does not re-randomize its address space after a crash. Specifically, Bittau *et al.* discuss that "the BROP attack cannot target PIE servers that re-randomize (e.g., execve) after a crash" [1]. We survey the 16 services previously identified (see Section 4.3), manually inspecting their source code to determine how each handles incoming connections. We characterize each service based on their connection-handling behavior.

We distinguish between processes than handle connections in a single-thread model, those that fork on each connection and those that utilize a thread pool. Services that fork on each connection satisfy the forking model required for the BROP attack. For POSIX threads and single-thread processes, a thread crash will cause the service to crash, generally. The behavior on crash, however, is consequential to the effective forking model: some services either do not restart and simply crash, while others are restarted by some background monitor which detects the failure. For any service which is restarted on crash, it may either

Table 4.10: Survey of forking models for connection handling.

Service	Version	Model	Source
apache	2.4.7	hybrid	server/mpm/worker/worker.c:1394
openswan	2.6.38	fork	programs/pluto/pluto_crypt.c:899
postgreSQL	9.3.9	fork-exec	src/backend/postmaster/postmaster.c:4185
axis2c server	1.6.0	thread pool	transport/tcp/receiver/tcp_svr_thread.c:151
cyrus	2.4.17	fork-exec	imap/smtpclient.c:71
wu-ftpd	2.6.1	inetd/fork	src/ftpd.c:7042
gsoap	2.8.16	various	-
postfix	2.11.0	fork-exec	src/global/mail_run.c:90
mongoDB	2.4.9	thread pool	src/mongo/db/db.cpp:281
nginx	1.4.6	fork	src/os/unix/ngx_process.c:186
lighttpd	1.4.33	fork-exec	src/lighttpd-angel.c:95
dovecot	2.2.9	fork	src/lib-lda/smtp-client.c:100
redis	2.8.4	fork	src/aof.c:961
sendmail	8.14.4	fork	sendmail/daemon.c:636
jabberd2 c2s	2.2.17	single-thread	router/main.c:343
tengine	2.1.1	fork	src/os/unix/ngx_process.c:177

restart via fork or via fork-exec. Those which restart via fork also satisfy the BROP forking model and are, effectively, following a forking model. These are annotated as, effectively, adopting a fork model (see Table 4.10).

Of the 16 services surveyed, about half satisfy the requirement to restart without randomization, either by following a pure fork model or a hybrid model. For example, Apache uses a hybrid multi-process multi-threaded model, creating a pool of connection-handling processes, each of which maintains a thread pool; more importantly, processes are restarted on crash via fork. WuFTP may be started either by `inetd`, which will fork-exec the service, or as a standalone daemon that forks on each connection. We note that gSOAP is not really a service itself, but rather a toolkit for building SOAP/XML web services, and it supports various connection handling models.

CHAPTER 5:
Preventions

In this chapter, we discuss ways of preventing BROP attacks from executing successfully. We summarize arguments previously presented by Bittau *et al.* [1] and others, and express arguments based on the analysis from prior chapters.

5.1 Behavioral Analysis

There are several methods proposed to detect when a BROP attack is being launched, using either host-based or network-based controls.

Pfaff *et al.* present a method (HadROP) for stopping BROP, as well as any other ROP-style attacks [2]. They claim that ROP attacks "trigger micro-architechtural events in modern processors differently than conventional programs." HadROP uses machine learning to detect these differences and protect the target service.

From a networking standpoint, the way BROP works should allow a defender to describe a list of behaviors, that upon seeing, would stop a targeted service. Bittau *et al.* [1] describe an "a fully automated exploit that yielded a shell in under 4,000 requests (20 minutes)." Since BROP's generalized stack reading works by crashing the service this could mean thousands of crashes on a service in a short time frame. Re-launching a service, and rerandomizing, after the number of crashes within some period of time exceeds a specified threshold may effectively reduce the threat of BROP attacks. Because of its high number of crashes in a short amount of time, it is likely that a BROP attack would have a signature that is distinct and identifiable. We leave future work exploring this aspect and proposing specific.

5.2 Encrypting Pointers

Encryption of pointers, as done by ASLR-Guard and PointGuard could prove to be effective BROP countermeasures [27], [28]. Similar to how XOR canaries work (see Section 4.2.1), ASLR-Guard and PointGuard provide protection by encrypting pointers, XORing them

with a random key. When pointers are encrypted, BROP would not be able to search for portions of consecutive entries that mark the PLT, since alterations to the ESP would point to unpredictable addresses. This protection may relegate BROP to being a brute force attack, not being able to distinguish between usable gadgets and binary found in the text segment.

5.3 Rerandomization

Seibert *et al.* note that BROP-style attacks, "can typically only be conducted on systems where repeated crashing goes undetected, the application is restarted after crashing, and memory is not re-randomized after restarting" [15]. This highlights a potentially simple preventions that can be implemented for BROP defense. When it is possible, re-randomization after a crash is the most absolute way to prevent BROP from being successful. Bittau *et al.* highlight that services that re-randomize after crashing are the bounds of a BROP attack. When the re-randomization occurs, the search for gadgets must start anew, preventing a BROP attack from progressing to its later stages.

Bittau *et al.* suggest rerandomization of canaries and address space after any crash in order to protect from their attack [1]. Employing a fork-exec model for handling new connections would cause the child to have a randomized address space. Execing, however, may not always be practical after a fork, although those services following this model have a type of built-inBROP protection.

Similar in spirit to target re-randomization via exec, Friedman *et al.* describe a potential prevention employing a technique called chronomorphic binaries. They develop a proof of concept amelioration in which binaries"that change their in-memory instructions and layout repeatedly" during their lifetime [29]. Friedman *et al.* specifically mention this defense as applicable to BROP: by continually changing instructions and layout, the gadgets required for a BROP attack could not be gathered fast enough to effectively execute the attack.

Another option would be to run a service under xinetd instead of stand alone. This offers more protection as rerandomization would then occur on restart.

5.4 Forcing a Weaker Attack

The findings of our binary survey suggest that a "first principles" attack has a far greater chance of failure, due to missing gadgets (see Section 4.3.1). This suggests a potential amelioration, by instrumenting binaries with mechanisms that force the attacker to default to this weaker attack. In particular, inlining the functions strcmp and write may render the attacker unable to perform an optimized BROP attack.

Similarly, changing the geometry of the PLT may prevent an optimized attack from succeeding in its search for this data structure. It is noteworthy that, when symbols are manually resolved using dlopen, the PLT may not follow the pattern described by Bittau *et al*. Future work may look into changing the way the PLT is formatted or guards to PLT entries, such as function pointers that are expected to never be called and would cause a crash or program re-randomization.

Relatedly, possible preventions may seek to weaken the first principles attack, by hardening binaries through reducing the quantity of gadgets required for a BROP attack, like `pop rdx`. It may be possible for build tools to select machine instructions in a way that avoids the use of some first principles gadgets. Onarlioglu *et al.* describe a type of this protection that could prove useful for stopping BROP attacks [30]. Their method transforms instructions that could be offset to produce gadgets with into similar functioning, but more benign, instructions.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusions

We have shown some of the abilities and limitations of the BROP attack. We have also enumerated the assumptions that must be satisfied for a BROP attack to occur successfully. Our work is based on the BROP attack described by Bittau *et al.*, though we imagine variant attacks, yet to be described, may impact this analysis and its conclusions.

We enumerate the precise prerequisites for a BROP attack. This helped us to analyze BROP and find limits to the attack. We found that the technique of generalized stack reading as a method for circumventing canaries may be more or less successful, depending on the type of canaries. This limits the types of stack buffer overflows that could lead to a BROP success.

We survey service binaries, finding that a first principals BROP attack is not a reliable fall-back plan when an optimized attack fails. This is due to a lack required gadgets to perform a first principles attack in many of the binaries surveyed. We believe that the majority of the services we survey should not be vulnerable to BROP, either because they lack gadgets or because they would re-randomize on crash.

From our survey, we find that 100% of manually built services had the required gadgets for an optimized BROP attack, and 75% of prebuilt services also had these gadgets. More generally, there appears to be a difference in gadget composition, depending on the environment and build options. This makes assessing the likelihood of a successful first principles attack complicated without access to the binary. We leave investigating this further to future work.

We show that `syscall` and `pop rax` gadgets, while not as rare as `pop rdx` gadgets, are uncommon and may be a limiting factor for first principles attack. The `rax` gadget is necessary to find a `syscall` gadget, yet we find that under 30% of common system utilities contain an `rax gadget`, while under 20% have a `syscall` gadgets (see Table 4.4).

We find the existence of required gadgets in a service binary fluctuates with version number

and build environment (see Appendix B). For example, we observe that only some binaries for Redis built on CentOS 7.1.1503 have the prerequisite gadgets; when built on Fedora 22, all versions have the necessary gadgets; when built on Ubuntu 14.04.01, no versions satisfy these prerequisites.

We find that none of the services have signatures as strong as the proof of concept service, with regards to system calls in a row in the PLT and percentage of the PLT that is system calls. We believe, as it is, BROP may have difficulty with services that have small PLTs.

We observe that none of the surveyed services have as large a percentage of system calls in their PLTs as the target service of the proof-of-concept exploit, nginx 1.4.0. Additionally, we find that most of the entries in the services we surveyed, including nginx 1.4.0, are not system calls. This contradicts certain assumptions of the optimized attack and suggests limitations. We leave it as future work to explore this further. For example, it remains to be seem if this is a weakness in practice, and whether all system calls in the PLT do produce EFAULT on invalid parameters.

APPENDIX A:
Braille Installation

The following instructions are for installing Braille reproducing the BROP demo against nginx 1.4.0. These instructions were tested with 64-bit Ubuntu Server 14.04.2 LTS.

1. Download nginx
   `wget http://nginx.org/download/nginx-1.4.0.tar.gz`
2. Download braille
   `wget http://www.scs.stanford.edu/brop/nginx-1.4.0-exp.tgz`
3. `tar -xzvf nginx-1.4.0-exp.tgz`
4. Install dependencies
   (a) `sudo apt-get install make ruby`
   (b) `sudo apt-get install libpcre3 libpcre3-dev`
   (c) `sudo apt-get install zlibc zlib1g zlib1g-dev`
5. Unizip and build nginx
   (a) `tar -xzvf nginx-1.4.0.tar.gz`
   (b) `cd nginx-1.4.0`
   (c) `./configure`
   (d) add "-fstack-protector" to obj/Makefile CFLAGS
   (e) `make`
   (f) `sudo make install`
6. Start nginx
   `sudo /usr/local/nginx/sbin/nginx`
7. Run BROP
   `nginx-1.4.0-exp/brop.rb 127.0.0.1`

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Survey Data

In the following sections, we present the raw data for the binary composition survey described in Section 4.3.

B.1 Basic BROP Gadgets Survey on Ubuntu

The below data was gathered by surveying 10 versions of 7 different services building from source using default options on Ubuntu.

Table B.1: Gadgets for a "first principles" BROP attack, building services on Ubuntu 14.04.01.

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
redis	2.6.5	3276994	4	142	3	253	0
	2.6.7	3682371	2	148	2	268	0
	2.6.6	3686355	2	147	2	270	0
	2.6.9	3688470	4	148	3	269	0
	2.6.8	3686763	4	148	2	268	0
	2.6.11	3695250	4	148	1	268	0
	2.6.10	3692970	4	149	3	269	0
	2.6.13	3704840	6	149	3	270	0
	2.6.12	3697058	8	150	3	268	0
	2.6.14	3705120	6	149	2	271	0
postgreSQL	9.3.8	441914	0	15	1	43	0
	9.3.9	441914	0	15	1	43	0
	9.4.0	454719	0	18	3	43	0
	9.4.1	454766	0	18	3	43	0
	9.4.2	454913	0	18	1	43	0
	9.4.3	454913	0	18	1	43	0
	9.4.4	454913	0	18	1	43	0
	9.3.7	441914	0	15	1	43	0

Gadgets for a "first principles" BROP attack, building services on Ubuntu (continued).

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
	9.3.6	441767	0	16	1	45	0
	9.3.5	441762	2	15	1	44	0
mongoDB	3.1.4	11537432	776	1389	99	3649	92
	3.1.5	13893975	738	1406	81	3677	88
	3.1.6	14387925	825	37	28	13	4
	3.1.7	22749569	1037	78	53	47	35
	3.1.0	11601496	714	1422	88	3746	96
	3.1.1	11376376	621	1352	96	3639	97
	3.1.2	11520504	751	1379	92	3662	97
	3.1.3	11446936	749	1369	90	3636	88
	3.0.6	11686488	689	1403	91	3726	98
	3.1.8	18671777	945	39	30	15	13
lighttpd	1.4.29	873388	2	44	0	46	0
	1.4.28	861247	0	44	0	46	1
	1.4.30	873868	0	44	0	46	0
	1.4.31	868737	0	44	0	46	0
	1.4.32	869377	0	44	0	46	0
	1.4.33	870530	0	44	0	45	0
	1.4.34	867803	0	44	0	45	0
	1.4.35	853304	1	44	0	45	0
	1.4.36	909685	1	47	1	49	0
	1.4.37	929595	3	46	1	51	0
nginx	1.8.0	3403395	1	68	1	198	0
	1.7.6	3342630	0	67	0	193	0
	1.7.7	3362982	0	68	0	196	0
	1.9.4	3447159	3	71	0	199	0
	1.7.8	3373897	0	69	1	195	0
	1.7.9	3396115	0	69	0	196	1
	1.9.0	3431972	1	71	0	197	0

Gadgets for a "first principles" BROP attack, building services on Ubuntu (continued).

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
	1.9.1	3439638	1	71	0	199	0
	1.9.2	3445276	1	71	0	199	0
	1.9.3	3446756	2	71	0	199	0
tengine	1.5.0	5582267	0	0	0	0	0
	1.5.1	5583155	0	73	0	238	0
	1.5.2	5587403	0	73	0	238	0
	1.4.6	5483648	2	73	0	235	0
	2.1.1	6091896	1	85	0	267	0
	2.1.0	6062797	0	87	0	261	0
	2.0.2	5805721	1	78	0	248	0
	2.0.3	5884270	1	82	0	252	0
	2.0.0	5771674	3	77	1	246	4
	2.0.1	5805721	1	78	0	248	0
dovecot	2.2.12	300397	0	9	0	19	0
	2.2.13	300373	0	9	0	19	0
	2.2.10	300397	0	9	0	19	0
	2.2.11	300397	0	9	0	19	0
	2.2.16	304165	0	9	0	18	0
	2.2.17	301523	0	9	0	18	1
	2.2.14	304205	0	9	0	18	0
	2.2.15	304205	0	9	0	18	0
	2.2.18	301523	0	9	0	18	1
	2.2.9	299621	0	9	0	19	0

B.2 Basic BROP Gadgets Survey on CentOS

The below data was gathered by surveying 10 versions of 7 different services building from source using default options on CentOS.

Table B.2: Gadgets for a "first principles" BROP attack, building services on CentOS 7.1.1503.

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
redis	2.6.5	3160887	4	156	3	253	0
	2.6.7	3547064	5	164	4	269	1
	2.6.6	3546944	3	162	4	277	1
	2.6.9	3549019	3	163	4	269	0
	2.6.8	3547248	4	162	5	269	0
	2.6.11	3559875	2	162	2	269	0
	2.6.10	3557623	4	163	2	270	0
	2.6.13	3565385	5	163	1	271	0
	2.6.12	3561683	3	164	1	269	1
	2.6.14	3565657	5	163	1	271	0
postgreSQL	9.3.8	441747	0	15	2	44	0
	9.3.9	441747	0	15	2	44	0
	9.4.0	450418	0	18	2	43	0
	9.4.1	450465	0	18	1	43	0
	9.4.2	454708	0	18	1	44	1
	9.4.3	454708	0	18	1	44	1
	9.4.4	454708	0	18	1	44	1
	9.3.7	441747	0	15	2	44	0
	9.3.6	441600	0	15	1	43	1
	9.3.5	437499	2	15	1	44	0
mongoDB	3.1.4	11537432	776	1389	99	3649	92
	3.1.5	13893975	738	1406	81	3677	88
	3.1.6	14387925	825	37	28	13	4
	3.1.7	22749569	1037	78	53	47	35
	3.1.0	11601496	714	1422	88	3746	96

Gadgets for a "first principles" BROP attack, building services on CentOS (continued).

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
	3.1.1	11376376	621	1352	96	3639	97
	3.1.2	11520504	751	1379	92	3662	97
	3.1.3	11446936	749	1369	90	3636	88
	3.0.6	11686488	689	1403	91	3726	98
	3.1.8	18671777	945	39	30	15	13
lighttpd	1.4.29	849595	1	42	0	45	1
	1.4.28	837622	0	42	0	45	0
	1.4.30	850059	0	42	0	45	1
	1.4.31	844672	2	42	0	45	1
	1.4.32	845320	2	42	0	45	1
	1.4.33	-	-	-	-	-	-
	1.4.34	843930	0	43	0	43	0
	1.4.35	830661	1	43	0	43	0
	1.4.36	883690	0	47	0	47	0
	1.4.37	-	-	-	-	-	-
nginx	1.8.0	3283695	1	69	0	197	0
	1.7.6	3229674	0	67	2	192	0
	1.7.7	3248658	0	68	0	194	0
	1.9.4	3321760	1	71	0	198	0
	1.7.8	3255069	0	69	2	194	0
	1.7.9	3272719	4	69	0	195	0
	1.9.0	3311311	1	71	1	196	0
	1.9.1	3314841	1	72	0	198	0
	1.9.2	3319885	1	71	0	198	0
	1.9.3	3321365	1	73	0	198	0
tengine	1.5.0	-	-	-	-	-	-
	1.5.1	5459180	2	73	0	237	0
	1.5.2	5459244	2	73	0	237	0
	1.4.6	5361625	2	72	1	233	1

Gadgets for a "first principles" BROP attack, building services on CentOS (continued).

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
	2.1.1	5969567	4	85	2	266	0
	2.1.0	5941596	5	86	0	260	0
	2.0.2	5696179	4	78	1	247	0
	2.0.3	5770248	6	81	0	251	0
	2.0.0	5659012	4	77	0	245	0
	2.0.1	5696179	4	78	1	247	0
dovecot	2.2.12	293429	0	9	0	20	0
	2.2.13	293405	0	9	0	20	0
	2.2.10	293429	0	9	0	20	0
	2.2.11	293429	0	9	0	20	0
	2.2.16	296557	0	9	0	20	1
	2.2.17	294331	0	9	0	20	0
	2.2.14	296579	0	9	0	20	1
	2.2.15	296579	0	9	0	20	1
	2.2.18	294331	0	9	0	20	0
	2.2.9	292533	2	9	0	20	0

B.3 Basic BROP Gadgets Survey on Fedora

The below data was gathered by surveying 10 versions of 7 different services building from source using default options on Fedora.

Table B.3: Gadgets for a "first principles" BROP attack, building services on Fedora 22.

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
redis	2.6.5	3221896	2	154	2	243	2

Gadgets for a "first principles" BROP attack, building services on Fedora (continued).

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
	2.6.7	3601328	4	165	2	255	2
	2.6.6	3601280	4	164	4	256	2
	2.6.9	3607072	4	164	5	256	2
	2.6.8	3601344	7	164	4	256	2
	2.6.11	3613856	2	164	3	256	2
	2.6.10	3611616	2	164	2	257	2
	2.6.13	3624656	4	165	2	258	2
	2.6.12	3615448	3	165	3	256	2
	2.6.14	3625088	4	165	2	257	2
postgreSQL	9.3.8	441832	0	13	2	42	0
	9.3.9	441832	0	13	2	42	0
	9.4.0	450504	0	16	1	43	0
	9.4.1	450552	0	16	1	44	0
	9.4.2	454792	0	16	1	43	0
	9.4.3	454792	0	16	1	43	0
	9.4.4	454792	0	16	1	43	0
	9.3.7	441832	0	13	2	42	0
	9.3.6	441688	0	13	1	42	0
	9.3.5	441680	2	14	1	42	0
mongoDB	3.1.4	11537432	776	1389	99	3649	92
	3.1.5	13893975	738	1406	81	3677	88
	3.1.6	14387925	825	37	28	13	4
	3.1.7	22749569	1037	78	53	47	35
	3.1.0	11601496	714	1422	88	3746	96
	3.1.1	11376376	621	1352	96	3639	97
	3.1.2	11520504	751	1379	92	3662	97
	3.1.3	11446936	749	1369	90	3636	88
	3.0.6	11686488	689	1403	91	3726	98
	3.1.8	18671777	945	39	30	15	13

Gadgets for a "first principles" BROP attack, building services on Fedora (continued).

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
lighttpd	1.4.29	864336	3	43	0	45	0
	1.4.28	852176	2	43	1	45	0
	1.4.30	868320	3	43	0	45	0
	1.4.31	866448	4	43	0	45	0
	1.4.32	860256	4	43	0	45	1
	1.4.33	-	-	-	-	-	-
	1.4.34	-	-	-	-	-	-
	1.4.35	-	-	-	-	-	-
	1.4.36	897008	0	47	2	48	0
	1.4.37	919768	0	46	1	51	0
nginx	1.8.0	3390136	4	64	2	212	0
	1.7.6	3339176	2	62	2	178	0
	1.7.7	3357784	2	63	2	180	0
	1.9.4	3434760	7	68	1	179	0
	1.7.8	3368664	6	64	2	176	0
	1.7.9	3385416	6	64	1	177	0
	1.9.0	3420264	4	69	1	178	0
	1.9.1	3427928	4	67	2	180	0
	1.9.2	3432888	4	67	1	180	0
	1.9.3	3434056	4	67	1	179	0
tengine	1.5.0	5590048	8	70	2	218	0
	1.5.1	5591016	6	70	2	218	0
	1.5.2	5591304	6	70	2	218	1
	1.4.6	5488248	5	69	2	214	0
	2.1.1	6096040	3	79	1	245	0
	2.1.0	6069272	3	78	3	241	0
	2.0.2	5820056	3	73	3	229	0
	2.0.3	5898648	3	75	2	233	0
	2.0.0	5787344	5	72	3	227	0

Gadgets for a "first principles" BROP attack, building services on Fedora (continued).

		Binary	Gadgets				
Service	Version	Size	syscall	rsi	rax	rdi	rdx
	2.0.1	5820056	3	73	3	229	0
dovecot	2.2.12	288592	0	8	0	18	0
	2.2.13	288592	0	8	0	18	0
	2.2.10	288592	0	8	0	18	0
	2.2.11	288592	0	8	0	18	0
	2.2.16	290936	0	8	0	18	0
	2.2.17	288720	0	8	0	18	0
	2.2.14	290856	0	8	0	18	0
	2.2.15	290856	0	8	0	18	0
	2.2.18	288720	0	8	0	18	0
	2.2.9	287984	0	8	0	18	0

B.4 Optimized BROP Gadgets Survey on Ubuntu

The below data was gathered by surveying 10 versions of 7 different services building from source using default options on Ubuntu.

Table B.4: Gadgets for the optimized BROP attack, building services on Ubuntu 14.04.01.

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
redis	2.6.5	3276994	265	True	True
	2.6.7	3682371	275	True	True
	2.6.6	3686355	276	True	True
	2.6.9	3688470	275	True	True
	2.6.8	3686763	275	True	True
	2.6.11	3695250	275	True	True

Gadgets for the optimized BROP attack, building services on Ubuntu (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
	2.6.10	3692970	276	True	True
	2.6.13	3704840	277	True	True
	2.6.12	3697058	275	True	True
	2.6.14	3705120	277	True	True
postgreSQL	9.3.8	441914	43	True	True
	9.3.9	441914	43	True	True
	9.4.0	454719	43	True	True
	9.4.1	454766	43	True	True
	9.4.2	454913	43	True	True
	9.4.3	454913	43	True	True
	9.4.4	454913	43	True	True
	9.3.7	441914	43	True	True
	9.3.6	441767	43	True	True
	9.3.5	441762	44	True	True
mongoDB	3.1.4	11537432	3841	True	True
	3.1.5	13893975	3852	True	True
	3.1.6	14387925	8	True	True
	3.1.7	22749569	8	True	True
	3.1.0	11601496	3963	True	True
	3.1.1	11376376	3828	True	True
	3.1.2	11520504	3845	True	True
	3.1.3	11446936	3825	True	True
	3.0.6	11686488	3947	True	True
	3.1.8	18671777	8	True	True
lighttpd	1.4.29	873388	44	True	True
	1.4.28	861247	44	True	True
	1.4.30	873868	44	True	True
	1.4.31	868737	44	True	True
	1.4.32	869377	44	True	True

Gadgets for the optimized BROP attack, building services on Ubuntu (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
	1.4.33	870530	44	True	True
	1.4.34	867803	44	True	True
	1.4.35	853304	44	True	True
	1.4.36	909685	48	True	True
	1.4.37	929595	52	True	True
nginx	1.8.0	3403395	214	True	True
	1.7.6	3342630	208	True	True
	1.7.7	3362982	210	True	True
	1.9.4	3447159	215	True	True
	1.7.8	3373897	210	True	True
	1.7.9	3396115	211	True	True
	1.9.0	3431972	213	True	True
	1.9.1	3439638	214	True	True
	1.9.2	3445276	215	True	True
	1.9.3	3446756	215	True	True
tengine	1.5.0	5582267	255	True	True
	1.5.1	5583155	255	True	True
	1.5.2	5587403	256	True	True
	1.4.6	5483648	251	True	True
	2.1.1	6091896	289	True	True
	2.1.0	6062797	284	True	True
	2.0.2	5805721	268	True	True
	2.0.3	5884270	274	True	True
	2.0.0	5771674	265	True	True
	2.0.1	5805721	268	True	True
dovecot	2.2.12	300397	18	True	True
	2.2.13	300373	18	True	True
	2.2.10	300397	18	True	True
	2.2.11	300397	18	True	True

Gadgets for the optimized BROP attack, building services on Ubuntu (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
	2.2.16	304165	18	True	True
	2.2.17	301523	18	True	True
	2.2.14	304205	18	True	True
	2.2.15	304205	18	True	True
	2.2.18	301523	18	True	True
	2.2.9	299621	18	True	True

B.5 Optimized BROP Gadgets Survey on CentOS

The below data was gathered by surveying 10 versions of 7 different services building from source using default options on CentOS.

Table B.5: Gadgets for the optimized BROP attack, building services on CentOS 7.1.1503.

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
redis	2.6.5	3160887	266	True	True
	2.6.7	3547064	277	True	True
	2.6.6	3546944	278	True	True
	2.6.9	3549019	277	True	True
	2.6.8	3547248	277	True	True
	2.6.11	3559875	277	True	True
	2.6.10	3557623	278	True	True
	2.6.13	3565385	279	True	True
	2.6.12	3561683	277	True	True
	2.6.14	3565657	279	True	True
postgreSQL	9.3.8	441747	43	True	True

Gadgets for the optimized BROP attack, building services on CentOS (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
	9.3.9	441747	43	True	True
	9.4.0	450418	43	True	True
	9.4.1	450465	43	True	True
	9.4.2	454708	43	True	True
	9.4.3	454708	43	True	True
	9.4.4	454708	43	True	True
	9.3.7	441747	43	True	True
	9.3.6	441600	43	True	True
	9.3.5	437499	44	True	True
mongoDB	3.1.4	11537432	3841	True	True
	3.1.5	13893975	3852	True	True
	3.1.6	14387925	8	True	True
	3.1.7	22749569	8	True	True
	3.1.0	11601496	3963	True	True
	3.1.1	11376376	3828	True	True
	3.1.2	11520504	3845	True	True
	3.1.3	11446936	3825	True	True
	3.0.6	11686488	3947	True	True
	3.1.8	18671777	8	True	True
lighttpd	1.4.29	849595	44	True	True
	1.4.28	837622	44	True	True
	1.4.30	850059	44	True	True
	1.4.31	844672	44	True	True
	1.4.32	845320	44	True	True
	1.4.33	-	-	-	-
	1.4.34	843930	42	True	True
	1.4.35	830661	42	True	True
	1.4.36	883690	46	True	True
	1.4.37	-	-	-	-

Gadgets for the optimized BROP attack, building services on CentOS (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
nginx	1.8.0	3283695	214	True	True
	1.7.6	3229674	207	True	True
	1.7.7	3248658	209	True	True
	1.9.4	3321760	215	True	True
	1.7.8	3255069	209	True	True
	1.7.9	3272719	211	True	True
	1.9.0	3311311	213	True	True
	1.9.1	3314841	214	True	True
	1.9.2	3319885	215	True	True
	1.9.3	3321365	215	True	True
tengine	1.5.0	-	-	-	-
	1.5.1	5459180	254	True	True
	1.5.2	5459244	255	True	True
	1.4.6	5361625	250	True	True
	2.1.1	5969567	288	True	True
	2.1.0	5941596	283	True	True
	2.0.2	5696179	267	True	True
	2.0.3	5770248	273	True	True
	2.0.0	5659012	264	True	True
	2.0.1	5696179	267	True	True
dovecot	2.2.12	293429	20	True	True
	2.2.13	293405	20	True	True
	2.2.10	293429	20	True	True
	2.2.11	293429	20	True	True
	2.2.16	296557	20	True	True
	2.2.17	294331	20	True	True
	2.2.14	296579	20	True	True
	2.2.15	296579	20	True	True
	2.2.18	294331	20	True	True

Gadgets for the optimized BROP attack, building services on CentOS (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
	2.2.9	292533	20	True	True

B.6 Optimized BROP Gadgets Survey on Fedora

The below data was gathered by surveying 10 versions of 7 different services building from source using default options on Fedora.

Table B.6: Gadgets for the optimized BROP attack, building services on Fedora 22.

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
redis	2.6.5	3221896	243	True	True
	2.6.7	3601328	255	True	True
	2.6.6	3601280	256	True	True
	2.6.9	3607072	256	True	True
	2.6.8	3601344	256	True	True
	2.6.11	3613856	257	True	True
	2.6.10	3611616	257	True	True
	2.6.13	3624656	258	True	True
	2.6.12	3615448	257	True	True
	2.6.14	3625088	258	True	True
postgreSQL	9.3.8	441832	43	True	True
	9.3.9	441832	43	True	True
	9.4.0	450504	44	True	True
	9.4.1	450552	44	True	True
	9.4.2	454792	44	True	True
	9.4.3	454792	44	True	True

Gadgets for the optimized BROP attack, building services on Fedora (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
	9.4.4	454792	44	True	True
	9.3.7	441832	43	True	True
	9.3.6	441688	43	True	True
	9.3.5	441680	43	True	True
mongoDB	3.1.4	11537432	3841	True	True
	3.1.5	13893975	3852	True	True
	3.1.6	14387925	8	True	True
	3.1.7	22749569	8	True	True
	3.1.0	11601496	3963	True	True
	3.1.1	11376376	3828	True	True
	3.1.2	11520504	3845	True	True
	3.1.3	11446936	3825	True	True
	3.0.6	11686488	3947	True	True
	3.1.8	18671777	8	True	True
lighttpd	1.4.29	864336	43	True	True
	1.4.28	852176	43	True	True
	1.4.30	868320	43	True	True
	1.4.31	866448	43	True	True
	1.4.32	860256	43	True	True
	1.4.33	—	-	-	-
	1.4.34	-	-	-	-
	1.4.35	-	-	-	-
	1.4.36	897008	46	True	True
	1.4.37	919768	49	True	True
nginx	1.8.0	3390136	192	True	True
	1.7.6	3339176	189	True	True
	1.7.7	3357784	191	True	True
	1.9.4	3434760	191	True	True
	1.7.8	3368664	187	True	True

Gadgets for the optimized BROP attack, building services on Fedora (continued).

		Binary		Gadgets	
Service	Version	Size	BROP	strcmp Found	write Found
	1.7.9	3385416	189	True	True
	1.9.0	3420264	190	True	True
	1.9.1	3427928	192	True	True
	1.9.2	3432888	192	True	True
	1.9.3	3434056	191	True	True
tengine	1.5.0	5590048	228	True	True
	1.5.1	5591016	228	True	True
	1.5.2	5591304	229	True	True
	1.4.6	5488248	224	True	True
	2.1.1	6096040	260	True	True
	2.1.0	6069272	256	True	True
	2.0.2	5820056	241	True	True
	2.0.3	5898648	247	True	True
	2.0.0	5787344	238	True	True
	2.0.1	5820056	241	True	True
dovecot	2.2.12	288592	18	True	True
	2.2.13	288592	18	True	True
	2.2.10	288592	18	True	True
	2.2.11	288592	18	True	True
	2.2.16	290936	18	True	True
	2.2.17	288720	18	True	True
	2.2.14	290856	18	True	True
	2.2.15	290856	18	True	True
	2.2.18	288720	18	True	True
	2.2.9	287984	18	True	True

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014, pp. 227–242.

[2] D. Pfaff, S. Hack, and C. Hammer, "Learning how to prevent return-oriented programming efficiently," in *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS)*. Springer, 2015, pp. 68–85.

[3] Attack sensitive brittle software. http://www.navysbir.com/n15_2/N152-120.htm/. Accessed: November 29, 2015.

[4] S. Fugate and P. Petrie, "SCHSIM: Using randomized compiling to immunize software systems against hacking," *CHIPS: the Department of the Navy's Information Technology Magazine*, July 2014. [Online]. Available: http://www.doncio.navy.mil/CHIPS/ArticleDetails.aspx?id=5334

[5] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[6] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[7] Ropgadget. https://github.com/JonathanSalwan/ROPgadget/tree/master. Accessed: November 20, 2015.

[8] Ropper. https://github.com/sashs/Ropper. Accessed: November 20, 2015.

[9] Rp. https://github.com/0vercl0k/rp. Accessed: November 20, 2015.

[10] Msfrop exploit development : Payloads. https://www.offensive-security.com/metasploit-unleashed/msfrop/. Accessed: November 20, 2015.

[11] Ropshell> free online rop gadgets search! http://www.ropshell.com/. Accessed: November 20, 2015.

[12] V. R. Mohan, "Source-free binary mutation for offense and defense," Ph.D. dissertation, The University of Texas at Dallas, 2014.

[13] CVE-2013-2028. urlhttp://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028 . Accessed: 27 September 2015.

[14] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2013, pp. 574–588.

[15] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2014, pp. 54–65.

[16] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[17] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, p. 80, 2011.

[18] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the USENIX Security Symposium*, 1998, pp. 63–78.

[19] R. Hill, "GCC 4.8.3 defaults to -fstack-protector," mailing list, 2014. [Online]. Available: https://www.gentoo.org/support/news-items/2014-06-15-gcc48_ssp.html

[20] Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack*, vol. 10, no. 56, 2000.

[21] G. Richarte, "Four different tricks to bypass StackShield and StackGuard protection," 2002. [Online]. Available: http://www.coresecurity.com/files/attachments/StackGuard.pdf

[22] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with stackguard," in *Proceedings of the Fifth Linux Expo*, May 1999.

[23] P. Wagle and C. Cowan, "StackGuard: Simple stack smash protection for GCC," in *Proceedings of the GCC Developers Summit*, 2003, pp. 243–255.

[24] P. Silberman and R. Johnson, "A comparison of buffer overflow prevention implementations and weaknesses," White Paper, IDEFENSE, 2004.

[25] H. Fritsch, "Buffer overflows on linux-x86-64," in *BlackHat Europe*, 2009. [Online]. Available: https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Buffer-Overflows-Linux-whitepaper.pdf

[26] D. Glynos, "libc6: use –enable-stackguard-randomization when building glibc," mailing list, 2009. [Online]. Available: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=511811

[27] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard^{TM}: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*, vol. 12, 2003, pp. 91–104.

[28] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 280–291.

[29] S. E. Friedman, D. J. Musliner, and P. K. Keller, "Chronomorphic programs: Using runtime diversity to prevent code reuse attacks," in *Proceedings of the 9th International Conference on Digital Society (ICDS)*, 2015.

[30] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 49–58.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California