ABSTRACT

Title of thesis: PREVENTING BUFFER OVERFLOWS WITH BINARY REWRITING

Pádraig O'Sullivan, Master of Science, 2010

Thesis directed by: Professor Rajeev Barua
Department of Electrical and Computer Engineering
Professor Angelos Keromytis
Department of Computer Science, Columbia University

Buffer overflows are the single largest cause of security attacks in recent times. Attacks based on this vulnerability have been the subject of extensive research and a significant number of defenses have been proposed for dealing with attacks of this nature. However, despite this extensive research, buffer overflows continue to be exploited due to the fact that many defenses proposed in prior research provide only partial coverage and attackers have adopted to exploit problems that are not well defended. The fact that many legacy binaries are still deployed in production environments also contributes to the success of buffer overflow attacks since most, if not all, buffer overflow defenses are source level defenses which require an application to be re-compiled. For many legacy applications, this may not be possible since the source code may no longer be available. In this thesis, we present an implementation of a defense mechanism for defending against various attack forms due to buffer overflows using binary rewriting. We study various attacks that happen in the real world and present techniques that can be employed within a binary rewriter to

protect a binary from these attacks.

Binary rewriting is a nascent field and little research has been done regarding the applications of binary rewriting. In particular, there is great potential for applications of binary rewriting in software security. With a binary rewriter, a vulnerable application can be immediately secured without the need for access to it's source code which allows legacy binaries to be secured. Also, numerous attacks on application software assume that application binaries are laid out in certain ways or have certain characteristics. Our defense scheme implemented using binary rewriting technology can prevent many of these attacks. We demonstrate the effectiveness of our scheme in preventing many different attack forms based on buffer overflows on both synthetic benchmarks and real-world attacks.

PREVENTING BUFFER OVERFLOWS WITH BINARY REWRITING

by

Pádraig O'Sullivan

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2010

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Dr. Peter Petrov
Dr. Gang Qu

© Copyright by
Pádraig O'Sullivan
2010

Table of Contents

List of Tables iv

List of Figures v

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Outline . . . 3

2 Related Work 4
- 2.1 Catalog of Attack Techniques . . . 4
  - 2.1.1 Buffer Overflow Attacks . . . 4
  - 2.1.2 Return-to-libc Attacks . . . 6
  - 2.1.3 Return-Orientated Programming . . . 7
- 2.2 Catalog of Defense Techniques . . . 8
  - 2.2.1 Compile Time Defenses . . . 8
  - 2.2.2 Instruction Set Randomization . . . 9
  - 2.2.3 Address Space Layout Randomization . . . 10
  - 2.2.4 Control Flow Integrity . . . 10
  - 2.2.5 Program Sheperding . . . 11
- 2.3 Related Work in Binary Rewriting . . . 12
- 2.4 Summary . . . 13

3 Binary Rewriting and Our Protection Scheme 15
- 3.1 Binary Rewriting . . . 15
  - 3.1.1 Architecture of Binary Rewriter . . . 18
- 3.2 Stack Canary Insertion . . . 19
- 3.3 Base Pointer Elimination . . . 20
- 3.4 Return Address Protection . . . 21
- 3.5 Function Pointer Protection . . . 22
- 3.6 longjmp/setjmp Protection . . . 23
- 3.7 Summary . . . 24

4 Experimental Results 25
- 4.1 Synthetic Results . . . 25
  - 4.1.1 Benchmark Description . . . 25
  - 4.1.2 Methodology . . . 27
  - 4.1.3 Results and Analysis . . . 28
- 4.2 Overheads . . . 28
  - 4.2.1 Binary Rewriting Overhead . . . 28
  - 4.2.2 Security Related Overheads . . . 33
- 4.3 Real World Attacks . . . 34

5 Conclusions and Future Work 36

Bibliography 37

List of Tables

4.1 Results on the Wilander and Kamkar Benchmarks. 29
4.2 Application Characteristics 30

List of Figures

3.1 SecondWrite system 18

4.1 Normalized runtime of rewritten binary as compared to input binary (runtime=1.0) that is un-optimized 31

4.2 Normalized runtime of rewritten binary as compared to input binary (runtime=1.0) that is optimized 32

4.3 Runtime overhead of rewritten binaries after inserting security checks. 34

Chapter 1

Introduction

The dominant form of software security vulnerability is the buffer overflow vulnerability. Attacks based on this vulnerability have been the subject of extensive research and a significant number of defenses have been proposed for dealing with attacks of this nature. Despite this extensive research, buffer overflows continue to be exploited in real world attacks. This is because most buffer overflow defenses provide only partial coverage, and the attacks have adopted to exploit problems that are not well defended.

Numerous schemes have been proposed for dealing with buffer overflow attacks. In our eyes, the applicability of such schemes depends on a number of factors, including:

1. Ease of use
2. Availability
3. Robustness
4. Applicability to all binaries
5. Low overheads

In this thesis, we present a scheme which meets the applicability requirements for a good security scheme. Our scheme is implemented using a binary rewriter

developed within our research group named SecondWrite. A binary rewriter takes a binary executable program as input, and produces a modified, improved executable as output.

Two of the novel aspects of SecondWrite are: 1) the input binary is translated into an existing compiler's intermediate representation (IR); and 2) binaries without relocation or symbolic information can be rewritten.

The ability to translate a binary to a high-level IR allows SecondWrite to insert security mechanisms that would otherwise require access to source code. Using a rewriter to insert security in a binary is important for consumers/users of software, who are otherwise at the mercy of the vendor when it comes to using security mechanisms or fixing known problems (patch management). With a binary rewriter, an administrator can (in principle) modify a binary to fix or mitigate a vulnerability.

1.1 Contributions

The contributions of this thesis are primarily in demonstrating a key application of binary rewriting. To this end, this thesis makes two major contributions:

1. A scheme is presented for protecting against commong buffer overflow attacks using a sophisticated and novel static binary rewriter

2. The scheme presented is practical, effective, and immediately deployable

1.2 Outline

In Chapter 2, we discuss some of the common attack forms and survey existing defenses for these attacks. We also briefly discuss some previous work in the field of binary rewriting. In Chapter 3, we describe how our binary rewriter works and how the various components of our scheme are implemented. In Chapter 4, we present experimental results which demonstrate the practicality of our scheme. Finally, in Chapter 5, we present our conclusions and discuss directions for future work.

Chapter 2

Related Work

Our work is related to many techniques that attempt to defend against malicious applications and vulnerabilities in applications which present attackers with an opportunity to attack. In this chapter, we elaborate on some of the pieces of work most closely related to ours. First, we present the various attack techniques utilized by attackers that are relevant for this thesis and then we go on to present various techniques proposed for mitigating these attack techniques. When discussing defenses, we draw attention to the drawbacks of each technique. We also briefly discuss related work in binary rewriting.

2.1 Catalog of Attack Techniques

2.1.1 Buffer Overflow Attacks

A buffer overflow refers to a situation that can occur when a function contains a local bounded array, or buffer, and writes into that buffer are not correctly guarded. Since C and C++ compilers typically use the stack for local variables as well as parameters, frame pointers, and saved return addresses, writes into a buffer that are not correctly guarded may overwrite and corrupt the return address stored on the stack. Data copied into the buffer whose length is larger than the buffer's size is referred to as a buffer overflow.

If an attacker controls the data used by the function, attackers can exploit buffer overflows and change the function's return address to an arbitrary value. In this case, when the function returns, the attacker can direct execution to code of their choice. This technique was first described in detail by AlephOne in 1996 [15]. However, attacks of this kind date back to before 1988 when the technique was used in the *fingerd* exploit of the Morris worm.

An example of a function with a buffer overflow vulnerability adopted from [8] follows:

```
int is_file_my_file(char *one, char *two)
{
  char tmp[MAX_FILE_LEN];
  strcpy(tmp, one);
  strcat(tmp, two);
  return (strcmp(tmp, "/home/posulliv/my_file"));
}
```

The above (somewhat unrealistic) function compares the concatentation of two input strings against */home/posulliv/my_file*. If the input strings to the above function can be chosen by an attacker, then the attacker can change the program's flow of control by overflowing the *tmp* buffer and changing the return address stored on the stack to an address of the attacker's choosing.

Commonly, an attacker would choose their input data so that the machine code for an attack payload would be present at the modified return address. When the vulnerable function returns, and execution of the attack payload begins, the attacker has gained control of the behavior of the target software. The attack payload is often called shellcode, since a common goal of an attacker is to launch a command line

interpreter (referred to as a shell in UNIX like environments) under their control.

2.1.2 Return-to-libc Attacks

As an alternative to supplying executable code (referred to as direct code injection), an attacker might be able to craft an attack that executes existing machine code (indirect code injection). This class of attacks has been referred to as jump-to-libc or return-to-libc (arc injection [6] has also been used to refer to this class of attacks) because the attack often involves directing execution towards machine code in the standard C library (libc) [6]. The standard C library is often the target for attacks of this type since it is loaded in nearly every UNIX program and it contains routines of the sort that are useful for an attacker. This technique was first suggested by Solar Designer in 1997 [7]. Attacks of this kind can evade defense mechanisms that protect the stack such as stack canaries and it is also effective against defenses that only allow memory to be writable or executable.

An example of a function vulnerable to such an attack adopted from [8] follows:

```
int median(int *data, int len, void *cmp)
{
  int tmp[MAX_INTS];
  /* copy the input integers */
  memcpy(tmp, data, len * sizeof(int));
  /* sort the local copy */
  qsort(tmp, len, sizeof(int), cmp);
  /* median is in the middle */
  return tmp[len / 2];
}
```

The above function is vulnerable to a buffer overflow as outlined in the previous

sub-section. However, an attacker can also corrupt the comparison function pointer cmp before it is passed to the *qsort* library function. If an attacker is able to accomplish this, he/she can gain control of execution at the point where the *qsort* function calls its copy of the corrupted *cmp* argument.

Traditionally, attacks of this kind have targeted the system function in the standard system library which allows the execution of an arbitrary command with arguments. However, recent attacks have been demonstrated which do not depend on calling functions in the standard C library.

2.1.3 Return-Orientated Programming

The technique of return-oreintated programming was introduced by Shacham [19]. Using this technique, an attacker can induce arbitrary behavior in a program whose control flow he/she has diverted without injecting any code. A return-orientated program chains together short instruction sequences already present in a program's address space, each of which ends in a return instruction. Several instructions can be combined into a gadget which is the basic block within a return-orientated program that performs operations. Gadgets are self-contained and perform one well-defined step of a computation. An attacker uses these gadgets to craft stack frames that can then perform arbitrary computations.

Shacham et al. [3] showed that the standard C library in both Linux running on the x86 platform and Solaris running on the SPARC platform contain enough useful intructions to construct meaningful gadgets. They manually analyzed the

standard C library on both platforms and constructed a library of gadgets that is Turing complete.

2.2 Catalog of Defense Techniques

2.2.1 Compile Time Defenses

StackGuard [4] places a 'canary' on the stack between local variables and the return address. This canary value is designed to warn of stack corruption since validating the integrity of the canary value is an effective means of ensuring that the function return address has not been corrupted. Microsoft's compiler also supports the insertion of stack canaries with the /GS option.

ProPolice [9] is similar to StackGuard in that it places a canary value on the stack. However, ProPolice also places arrays and other function-local buffers above all other function-local variables on the stack. Copies of all function arguments are also made into new, function-local variables that also sit below any buffers in the function. As a result, these variables and arguments are not subject to corruption through an overflow of these buffers.

StackGuard, PointGuard, and ProPolice involve compile-time analysis and transformation. Thus, unless the source code for an application is available, these techniques can not be used thereby hindering the ability to easily deploy these techniques. Our techniques do not suffer from this drawback since they can be easily deployed on any binary produced from any source language and compiler.

2.2.2 Instruction Set Randomization

Instruction-set randomization [1] is a promising technique for protecting against buffer overflows (and many kinds of code injection attacks). This approach randomizes the underlying system's instructions so that foreign code injected by an attacker would fail to execute correctly since the attacker does not know the instruction set of the target system. However, as mentioned by the authors in [1], the main drawback of this technique as applied to binary code that is meant to execute on a hardware processor is the need for special support by the processor. Thus, even though instruction-set randomization offers a strong defense against buffer overflow attacks the fact that unless it is supported by specialized hardware, it incurs significant overheads means that it is inlikely to see adoption in practice for the foreseeable future. However, when applied to interpreted languages (such as SQL), it can prove to be very effective. However, interpreted languages are not vulnerable to the buffer overflow attacks being discussed in this thesis.

In [12], the authors utilized Strata and Diablo to implement instruction set randomization. Diablo is used to prepare a binary for string encryption and introduce the information necessary to detect foreign code. Strata is then used to provide the necessary virtual execution environment for safe execution. The main contribution of this work is that the instruction-set randomization implementation is efficient while requiring no special hardware support. However, the runtime overheads reported are still high and likely to limit the practical adoption of such a system.

A binary rewriter such as the one used for our research suffers from none of these issues. No special hardware is required in order to utilize a binary rewriter and overheads are relatively low. In fact, if an original binary was compiled without optimizations, it is likely to see a runtime improvement when rewritten.

2.2.3 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) can be seen as a relatively coarse grained form of software diversity [21]. ASLR shuffles, or randomizes, the layout of software in the memory address space. It is effective at preventing remote attackers that have no existing means of running code on a target system from crafting attacks that depend on addresses. ASLR is not intended to defend against attackers that are able to control the software execution and its utility on 32-bit architectures is limited by the number of bits available for address randomization [20].

A binary rewriter could easily be used to provide a similar defense mechanism as ASLR. In fact, an interesting future avenue of research is to investigate software diversity through binary rewriting.

2.2.4 Control Flow Integrity

Control Flow Integrity (CFI) [13] is a basic safety property that can prevent attacks from arbitrarily controlling program behavior. CFI dictates that software execution must follow a path of a control-flow graph that is determined ahead of time

by analysis (in this case, static binary analysis is performed). CFI is enforced using static verification and binary rewriting (with Microsoft's Vulcan tool) that instruments software with runtime checks. These checks aim to ensure that control flow remains within a given control-flow-graph. CFI is a very effective defense against buffer overflow attacks (and any attack which attempts to change a program's control flow) since any attempt by an attacker to divert the control flow of a program will be caught by CFI. However, the main barrier to CFI's adoption seems to be the overhead associated with the scheme. In [13], the authors state that the average overhead of CFI (as implemented by them) is 16% on the SPEC2000 benchmarks. Also, the binary rewriter used by CFI depends on a binary being compiled with debug information which may not always be available. If a binary is not compiled with debug information then the implemenation of CFI in [13] cannot be utilized.

Our schemes implemented through our binary rewriter can provide the same level of protection as CFI. An additional advantage of our scheme is that our binary rewriter does not require access to any special information in an input binary unlike the binary rewriter used in CFI which requires access to debug information.

2.2.5 Program Sheperding

Program sheperding employs an efficient machine-code interpreter (DynamoRIO [2]) for implementing a security enforcement mechanism. A broad class of security policies can be implemented using a machine interpreter such as DynamoRIO. For example, DynamoRIO could be used to enforce control-flow integrity. Program

shepherding enforces a similar policy that imposes certain runtime restrictions on control flow such that an attacker can not alter a program's flow of control.

Program sheperding can experience significant memory and runtime overheads, particularly on the Windows platform. The scheme requires an application and interpreter to be run simultaneously. The high overheads in some cases are likely to limit adoption of program sheperding.

2.3 Related Work in Binary Rewriting

Binary rewriting and link time optimizatizers have been considered by a number of researchers. Binary rewriting research is being carried out in two directions: static rewriting and dynamic rewriting. Dynamic binary rewriters rewrite the binary during its execution. Examples are PIN [5], BIRD [17], DynInst [11], DynamoRIO [2] and Valgrind [14]. None of the dynamic binary rewriters we found employ a compiler like intermediate representation. This is not surprising since dynamic rewriters construct their internal representation at run-time, and hence they would not have the time to construct a compiler IR. Dynamic rewriters are hobbled since they do not have enough time to perform complex compiler transformations either; they have been primarily used for code instrumentation and simple security checks in the past. We do not discuss dynamic rewriters further since the methods used for rewriting in this thesis are primarily directed at static binary rewriters such as SecondWrite.

Existing static binary rewriters related to our approach include Etch [16], ATOM [10], PLTO [18], and Diablo [22]. None of the existing rewriters employ a

compiler level intermediate representation; rather they define their own low-level custom intermediate representation. Diablo defines an augmented whole program control-flow-graph based intermediate representation with program registers as globals and memory as a black box. It does not attempt to obtain high-level information like function prototypes and is geared mainly towards optimizations like code compaction. Taking memory as a black box limits its applicability to architectures like x86 which have a limited set of registers. ATOM defines a symbolic RTL-based intermediate representation with infinite registers but does not make any attempt of analyzing or modifying the stack layout. Its mainly targeted towards RISC architectures. PLTO employs a whole program CFG based IR and implements stack analysis to determine the use-kill depths of each function. However, this information is not used for converting it into high-level IR; rather it is only used for low-level custom optimizations like load/store forwarding. Etch does not explicitly build an intermediate representation build an intermediate representation and allows users to add new tools to analyze binaries. The primary goal of Etch appears to be instrumentation and has only been shown to be applicable for simple optimizations like profile-guided code layout.

2.4 Summary

Current defenses have a number of weaknesses:

- They are not easily deployable
  - Source code for an application is required

  - An application needs to be compiled with certain information
  - Hardware support is required

- Most are not readily available for use and evaluation

- Some suffer from un-acceptable overheads

- No scheme is immediately applicable to all binaries

Using the novel binary rewriting techniques developed within our research group, we have developed a scheme with the following characteristics:

- It is immediately deployable

- It is applicable to any binary

- Access to an application's source code is not required

- An application does not need to be compiled in any way such that it contains special information e.g. debug or relocation information

- It has low overheads

- Many common stack-based buffer overflow attacks are prevented

Chapter 3

Binary Rewriting and Our Protection Scheme

The cornerstone of our scheme is a binary rewriter which has been developed within our research group. In this chapter, we first discuss binary rewriting, give an overview of the binary rewriter we have developed in our research group, and then go on to discuss the various components of our scheme that we have implemented as part of our binary rewriter.

3.1 Binary Rewriting

Binary rewriters are pieces of software that accept a binary executable program as input, and produce an improved executable as output. The output executable typically has the same functionality as the input, but is improved in one or more metrics, such as run-time, energy use, memory use, security or reliability.

In recognition of its potential, binary rewriting has seen much active research over the last decade. The reason for great interest in this area is that binary rewriting offers additional advantages over compiler-produced optimized binaries:

- **Ability to do inter-procedural optimization.** Although compilers in theory can do whole-program optimizations, the reality is that they do little if any. Many commercial compilers - even highly optimizing ones - limit themselves to separate compilation, where each file (and sometimes each function)

is compiled in isolation. Inter-procedural link-time optimizations are often absent, and even when present, are usually far less powerful than compile-time optimizations since they work on low-level object code without the benefit of the extensive optimizations on IR available in the compiler. In contrast, binary rewriters have access to the complete application all at once, including libraries. This allows them to perform aggressive whole-program optimizations to exceed the performance of even optimized code.

- **Ability to do optimizations missed by the compiler.** Some binaries, especially legacy binaries or those compiled with inferior compilers, often miss certain optimizations. Binary rewriters can perform these optimizations missed by the compiler while preserving the optimizations the compiler did actually perform.

- **Increased economic feasibility.** It is cheaper to implement a code transformation once for an instruction set in a binary rewriter, rather than repeatedly for each compiler for the instruction set. For example, the ARM instruction set has over 30 compilers available for it, and the x86 has a similarly large number of compilers from different vendors and for different source languages. The high expense of repeated compiler implementation often cannot be supported by a small fraction of the demand.

- **Portable to any source language and any compiler.** A binary rewriter works for code produced from any source language by any compiler.

- **Works for hand-coded assembly routines.** Code transformations cannot be applied by a compiler to hand-coded assembly routines, since they are never compiled. In contrast, a binary rewriter can transform such routines.

However, binary rewriters today have fallen far short of this desired vision. Binary rewriters remain relatively crude tools today, capable of no more than simple program transformations such as peephole optimization and code instrumentation. Complex transformations such as extensive whole-program optimizations, automatic parallelization and sophisticated security enforcement, which we study in this thesis, remain outside the capabilities of current rewriters.

The binary rewriter developed by our group and utilized for this research is named SecondWrite. Our binary rewriter employs the widely used open-source LLVM compiler infrastructure and in particular, LLVM's high-level intermediate representation to represent code. Our custom binary reader and de-compiler modules read a binary and produce requivalent LLVM IR code using some of the techniques we will briefly describe in Section 3.1.1.

For this thesis, we study using binary rewriting to retroactively add security to a vulnerable binary. When this extra security is added, a binary is no longer vulnerable to common buffer overflow attacks.

Two notable properties of using binary rewriting to enforce security are low-overhead and real-time prevention of malicious behaviors as will be seen when we present our experimental results.

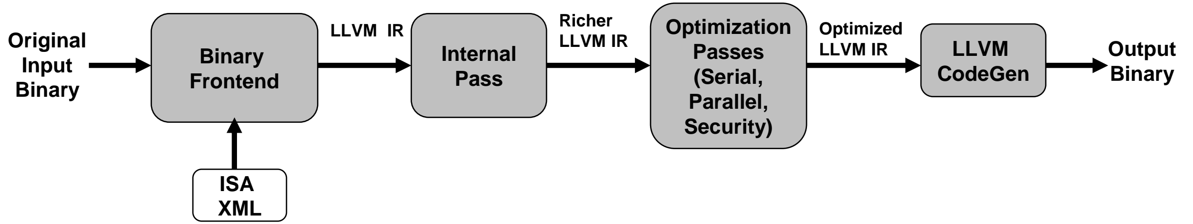

Figure 3.1: SecondWrite system

3.1.1 Architecture of Binary Rewriter

Figure 3.1 presents an overview of the SecondWrite system. The SecondWrite system consists of a frontend module for reading binary executables and generating an initial LLVM IR, an internal pass module for extracting more information about the underlying program, optimizing passes to implement various optimizations, and the LLVM code generator (codegen) for producing the rewritten binary.

The frontend module consists of a disassembler and a custom binary reader which processes the individual instructions and generates an initial LLVM IR. This initial representation is void of the desired IR features like function prototypes, abstract stack and virtual registers. The internal pass module analyzes this initial IR to obtain an improved IR which has all the information and features mentioned previously. Various optimization passes can be written on the above IR to obtain an optimized IR. Finally, the optimized IR is passed to the existing LLVM code generator to obtain the rewritten binary.

Various inherent characteristics of executables such as the unavailability of function prototypes, the use of a phyiscal stack and the use of the set of phyiscal

registers make it difficult to obtain a high-level IR from an input executable. A number of techniques have been developed within our group to extract this high-level information from executables whenever possible. We will not discuss those techniques in this thesis as the techniques are explained in detail elsewhere [].

3.2 Stack Canary Insertion

The first component of our scheme is the simplest. LLVM provides the ability to insert stack canaries during code generation. Utilizing this capability from LLVM allows us to easily provide the same level of protection to an un-protected binary as StackGuard would provide when given an application's source code.

Essentially, a random canary value is generated and placed on the stack during a function's prolog. In the function epilog, the value stored on the stack is compared with the random canary value for this process. If there is any difference, execution is halted as the canary value has been corrupted.

While this component of our scheme is simple, it demonstrates a key advantage of SecondWrite. By translating the input binary to LLVM's high-level intermediate representation, we were able to take advantage of features LLVM already provides. Thus, to achieve the same level of protection as StackGuard, we had to do very little once the binary was translated to LLVM's IR.

3.3 Base Pointer Elimination

The next component of our scheme is again due to existing LLVM optimizations. LLVM is an optimizing compiler and the binaries produced by LLVM are highly optimized. One common optimization applied by modern compilers on the x86 platform is to free up the EBP register for register allocation by removing the base (or frame) pointer.

Eliminating the base pointer also removes the ability for an attacker to craft an attack by modifying the old base pointer stored in a stack frame. In a binary which has not been compiled at a high optimization level, base pointers may still be used. In this case, the value of the base pointer will be pushed on the stack upon function entry in order for the value to be restored on function return. If an attacker is able to modify the value of the base pointer, he/she could have a fake stack frame they created be used thereby allowing the attacker to alter the flow of control of the program.

When the base pointer is eliminated by LLVM, any attack of this form is immediately prevented. There will be no base pointer for an attacker to modify. While corruption of the stack may still occur if an attacker overflows a buffer in order to attempt to overwrite the base pointer, no attack will be successful.

Again, the elimination of the base pointer highlights the advantages of Second-Write. By utilizing an existing compiler framework, we are able to produce highly optimized binaries which eliminate the ability for an attacker to perform an attack on a base pointer.

3.4 Return Address Protection

Given that stack canaries as inserted by LLVM do not provide the same level of protection as the ProPolice mechanism that comes with GCC, we decided to implement a more complete solution for protecting against corruption of the return address.

The basic idea of our return address protection scheme is as follows:

1. During the function prolog, store the return address of the current function in a global variable
2. In function epilog, compare the current return address on the stack with the value saved in a global variable
3. If there is any difference between these values, execution is halted

This simple scheme will detect if the return address has been modified either directly or indirectly. One complication with this scheme is the fact that a global variable is used for storing the return address. If a separate global variable was created for each function, memory overhead would become quite high. One solution is to use an array of global variables of a bounded size for saving return values. However, if function nesting is deep as in recursive functions, issues can still occur.

We applied an optimization for relieving this problem. We observed that this protection mechanism is only necessary if a function contains a write to a buffer. Thus, we analyze a function to look for any write to a buffer. If a function does not contain any write to a buffer then there is no need for the return address protection

mechanism to be inserted. During our experimental evaluation of our scheme, we have not yet come across a recursive function, which could cause issues for our scheme, that required return address protection to be inserted.

3.5 Function Pointer Protection

One common attack method used by attackers is to overwrite a function pointer so that when it is de-referenced, code of the attacker's choosing will be executed. In a binary executable, function pointers will appear as indirect calls. Thus, another component of our scheme concentrates on protecting all indirect calls and branches.

We have a scheme implemented which works correctly as will be seen in the experimental results chapter. However, research within our group is currently ongoing that will convert all indirect calls and branches in a binary to direct calls and branches. We will discuss how this affects our scheme after first describing what we have implemented.

Our initial scheme adds checking code before all indirect calls and branches. A global variable is declared in the data segment and its address is used as a boundary value. The checks inserted before any indirect call or branch ensure that the target of the indirect call or branch points to memory below the address of the global boundary variable. If the target points above the address of this global bondary variable then execution is halted.

With the new research occuring within our group, there will be no need to

insert checks into a binary. Basically, an indirect call in an input binary will be replaced by a large switch statement. This switch statement will contain a number of cases which are direct calls. SecondWrite will determine by analysis what all the valid targets of this indirect call are and each valid target will be a case in the switch statement. Thus, if at runtime, the target is not valid, execution will be abort.

3.6 longjmp/setjmp Protection

The paired functions *setjmp* and *longjmp* provide a means to alter a program's control flow besides the usual subroutine call and return sequence. First, *setjmp* saves the environment of the calling function into a data structure, and then *longjmp* can use this structure to jump back to the point it was created, at the *setjmp* call. A typical use for *setjmp/longjmp* is exception handling.

The data structure used by *setjmp* for saving the execution state is referred to as a *jmp_buf*. Within this structure, enough information is stored to restore a calling environment. In particular, one member of this structure saves the value of the program counter which is used when restoring the calling environment.

An attack method used by attackers is to overwrite the value of the member of a *jmp_buf* structure after a call to *setjmp* and before a call to *longjmp*. If an attacker has the ability to change the value of the program counter member of the *jmp_buf* structure then when *longjmp* is called, control will be transferred to an address of the attacker's choosing.

Our method for dealing with attacks of this kind is as follows:

- create a hash table within the global segment of the rewritten binary.
- after a call to *setjmp* use the current value of the program counter member of the *jmp_buf* structure as the key to the hash table.
- before a call to *longjmp* get the current value of the *jmp_buf* structure that will be used. Attempt to perform a lookup in the hash table for the value of the program counter.
- if the lookup in the hash table fails, then the value of the program counter has been modified and so we abort; otherwise execution continues

For now, the hash table created is of a fixed size. We have not experimented with binaries that have many calls to *setjmp* and *longjmp*.

The above scheme effectively mitigates attacks of this kind as demonstrated in the experimental results section.

3.7 Summary

In this chapter, we gave an overview of the binary rewriter developed within our group. We also described each of the individual components which make up our protection scheme. Together, these components can provide a high level of protection as demonstrated in the next chapter.

Chapter 4

Experimental Results

In this chapter, we present a number of experimental results. First, we examine the effectiveness of our security schemes as implemented in our binary rewriter on a set of security benchmarks previously proposed for evaluating the effectivness of buffer overflow defenses. Next, we examine the overheads of both the binary rewriter and our security scheme. Finally, we examine how effective our scheme is in protecting against a real-world attack.

4.1 Synthetic Results

In order to test how effective our scheme is, we utilized the benchmarks provided by Wilander and Kamkar [].

4.1.1 Benchmark Description

Wilander and Kamkar developed twenty buffer overflow attack forms in order to evaluate the effectiveness of tools available at the time that aimed to stop buffer overflows. An attack form is defined as a combination of a technique, location, and an attack target. These terms are in turn defined by Wilander and Kamkar as:

- **Technique** - either the buffer is overflowed all the way to the attack target or the buffer is overflowed to redirect a pointer to the target

- **Location** - the types of location for the buffer overflow are the stack or the heap/BSS/data segment
- **Attack target** - there are four targets - 1) the return address, 2) the old base pointer, 3) function pointers, and 4) longjmp buffers

Considering all practically possible combinations gives the twenty attack forms listed below:

1. Buffer overflow on the stack all the way to the target
   (a) Return address
   (b) Old base pointer
   (c) Function pointer as a local variable
   (d) Function pointer as parameter
   (e) Longjmp buffer as local variable
   (f) Longjmp buffer as function parameter
2. Buffer overflow on the heap/BSS/data segment all the way to the target
   (a) Function pointer
   (b) Longjmp buffer
3. Buffer overflow of a pointer on the stack and then pointing at target
   (a) Return address

(b) Old base pointer

(c) Function pointer as a local variable

(d) Function pointer as parameter

(e) Longjmp buffer as local variable

(f) Longjmp buffer as function parameter

4. Buffer overflow of a pointer on the heap/BSS/data segment and then pointing at target

    (a) Return address

    (b) Old base pointer

    (c) Function pointer as a local variable

    (d) Function pointer as parameter

    (e) Longjmp buffer as local variable

    (f) Longjmp buffer as function parameter

Of the twenty attack forms, we obtained the source code to only eighteen of these attack targets.

4.1.2 Methodology

We compiled the benchmarks using gcc 4.4. We compiled two versions of the benchmarks - one version had the *-fno-stack-protector* flag while the other had the *-fstack-protector* flag. The *-fstack-protector* flag creates a binary with the ProPolice protection mechanism embedded within it.

4.1.3 Results and Analysis

The results we recorded are shown in Table 4.1. In the table, "prevented" means that the process execution is unharmed. "halted" means that the attack is detected but the process is terminated. "missed" means the attack was successful. We refer to each attack form by the number assigned to it in Section 4.1.1.

As can be seen from the results in Table 4.1, the security inserted by our binary rewriter surpasses what is achieved by the ProPolice mechanism in the GCC compiler.

4.2 Overheads

4.2.1 Binary Rewriting Overhead

A subset of SPEC benchmarks and other benchmarks were selected to substantiate the performance of our binary rewriter. The benchmarks were selected purely at random, and are limited only by the criteria that they are correctly rewritten by our still-early prototype. Table 4.2 lists the set of benchmarks which are used for carrying out the experiments. All the benchmarks are compiled with GCC v4.4.1.

In the first experiment, all binaries executed correctly after rewriting thereby demonstrating the robustness of our binary rewriter. We were able to correctly apply the standard suite of LLVM optimization passes without any changes. These include CFG simplification, global optimization, global dead-code elimination, interprocedural constant propagation, instruction combining, condition propagation, tail-

Attack Form	**ProPolice**	**Binary Rewriter**
1 (a)	halted	halted
1 (b)	missed on ¡= O1/prevented ¿ O1	prevented
1 (c)	prevented	halted
1 (d)	prevented	halted
1 (e)	prevented	halted
1 (f)	missed	halted
2 (a)	missed	halted
2 (b)	missed	halted
3 (a)	prevented	halted
3 (b)	missed	prevented
3 (c)	prevented	halted
3 (d)	prevented	halted
3 (e)	prevented	halted
3 (f)	prevented	halted
4 (a)	missed	halted
4 (b)	missed	prevented
4 (c)	missed	halted
4 (e)	missed	halted

Table 4.1: Results on the Wilander and Kamkar Benchmarks.

Application	Source	Lines of C Source Code
perm	None	56
laplace	None	40
dijkstra	MiBench	134
tsp	Olden	473
lbm	SpecFP2006	1155
mcf	SpecInt2006	2685
libquantum	SpecInt2006	4357

Table 4.2: Application Characteristics

call elimination, induction variable simplification and selective loop unrolling.

Besides correctness, the next most important metrics are the run-time speedup or overhead of the rewritten binaries versus the input binaries. Two scenarios are of interest: when the input binaries were un-optimized, and when they were highly optimized. Both scenarios are discussed in turn below.

Figure 4.1 shows the normalized run-time of each rewritten binary compared to an input binary produced using GCC with no optimization (-O0 flag). We obtain an average improvement of 31% in execution time on our benchmarks with an improvement of over 40% in some cases. This result shows that our rewriter is useful for binaries that are not highly optimized, such as legacy binaries from older compilers, or binaries from compilers that are inferior compared to the best available

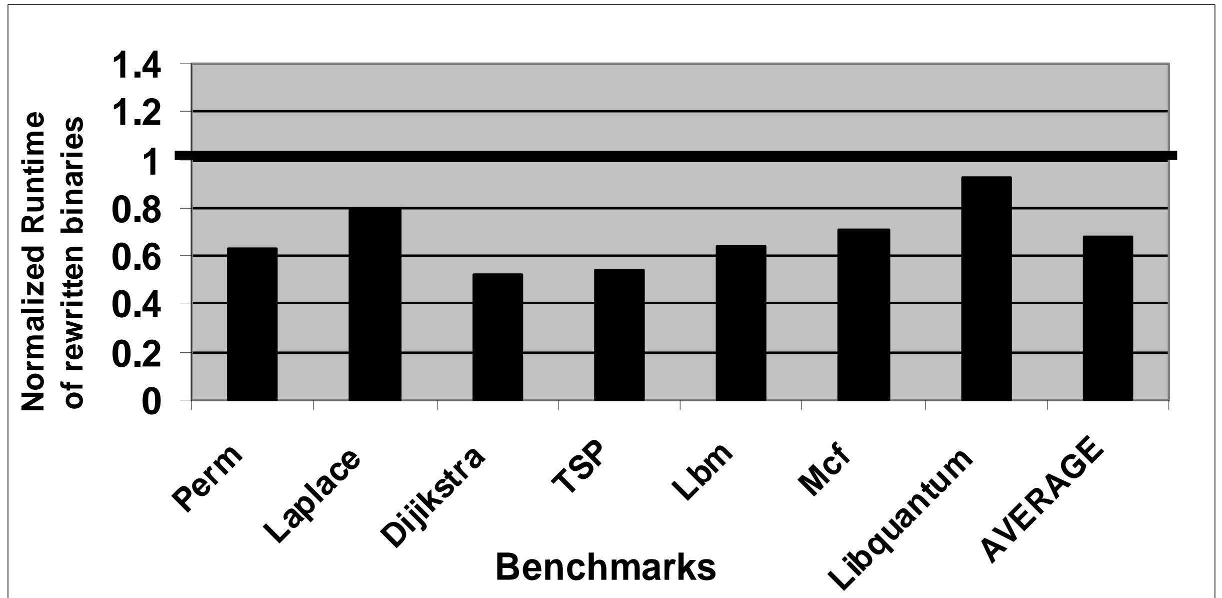

Figure 4.1: Normalized runtime of rewritten binary as compared to input binary (runtime=1.0) that is un-optimized

compilers. In most cases, after rewriting we raised the performance close to that of an optimized binary produced directly by GCC, showing the effectiveness of our approach.

Next, we study the performance of our rewriter on already optimized binaries. Figure 4.2 shows the normalized execution time of each rewritten binary compared to an input binary produced using GCC with the highest available level of optimization (-O3 flag). In this case, the results are mixed, with most benchmarks nearly breaking even or showing a small slowdown, one benchmark showing a larger slowdown of 13%, and one benchmark actually shows a speedup of 10%. The average is 4% slowdown.

We consider this near break-even performance on highly optimized binaries a

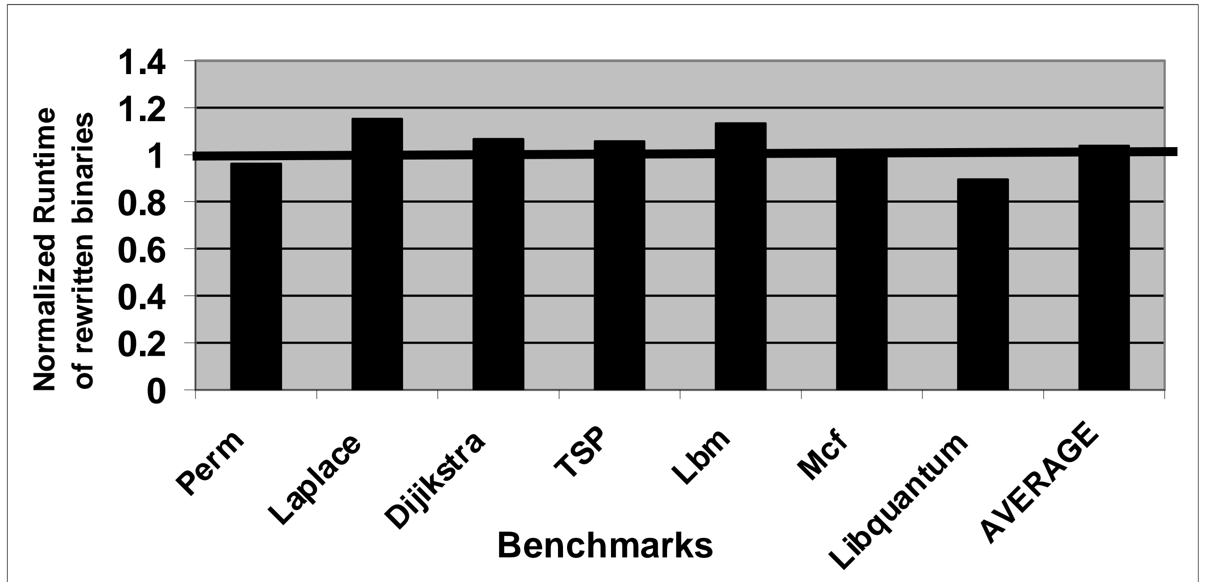

Figure 4.2: Normalized runtime of rewritten binary as compared to input binary (runtime=1.0) that is optimized

good result for the following three reasons:

1. our initial goal was not necessarily to get a speedup, but to generate correct IR without without introducing too much overhead. This would enable the IR to be a starting point for various custom compiler transformations we wanted to perform thereafter, such as automatic parallelization or security as covered in this thesis. Ultimately, these optimizations determine the utility of the rewriter.

2. these numbers represent our first cut implementation devoid of any attempt at producing a better IR more geared towards optimization. We believe these numbers can be substantially improved with more detailed IR and are exploring several related avenues.

3. we have currently not implemented any custom serial optimizations that are likely to improve performance further, such as the inter-procedural versions of common sub-expression elimination and loop-invariant code motion, changing the compiler-enforced calling convention for registers for better run-time, and more aggressive inlining. We believe these optimizations hold promise as the inter-procedural optimization abilities of current compilers are very limited compared to their intra-procedural performance.

One additional advantage of the binary rewriter is that it accumlates optimizations accross two compilers - rewritten binaries have an optimization if it is either present in the compiler that produced the input binary, or in the rewriter. In our case, if either GCC or LLVM had an optimization, the output binary should have it. This is why, for example, one of our rewritten binaries (libquantum) had a 10% speedup versus the input binary. Although GCC with the -O3 optimization flag is known to produce good code, in some cases it missed promoting structure fields to registers whereas LLVM did, explaining the speedup in libquantum. With better IR and more aggressive optimizations, we expect to see more consistent speedups in output binaries in the future.

4.2.2 Security Related Overheads

We measured the overhead of the security schemes we implemented. These results are presented in Figure 4.3. As seen the average overhead introduced is low at only 18%.

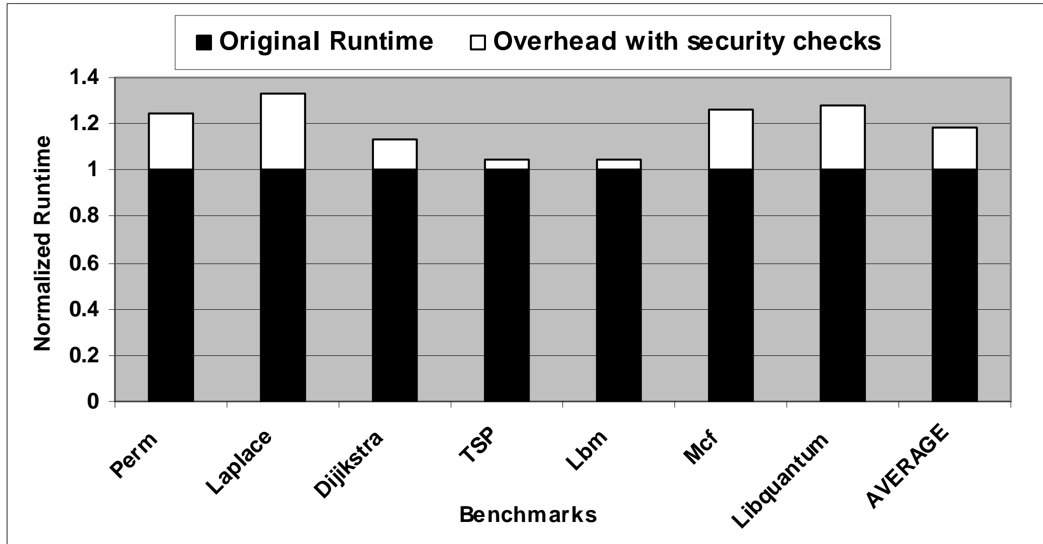

Figure 4.3: Runtime overhead of rewritten binaries after inserting security checks.

4.3 Real World Attacks

Ultimately, the success of the scheme presented in this thesis is only valuable if it is applicable to real-world attacks i.e. whether it can prevent attacks that have been observed in practice. In this section, we reproduce a real-world attack and demonstrate that our rewriter halts this attack.

A HTTP server, GHTTPD, has a stack buffer overflow vulnerability in its logging function. We produced an exploit for this server which overflows a stack-based buffer and corrupts the return address.

Using the return address protection component of our scheme, we were able to rewrite the GHTTPD server, add protection of the return address and prevent the attack which uses the buffer overflow vulnerability to corrupt the return address. When our protection scheme is enabled, the return address corruption is detected

and the application is aborted.

Chapter 5

Conclusions and Future Work

We presented a scheme developed using binary rewriting that can defend against a number of buffer overflow attacks. Our scheme is practical and easy to deploy with the only factor limiting deployment right now is maturity of the binary rewriter.

We demonstrated the effectiveness of our scheme using the benchmarks developed by Wilander and Kamkar by successfully defending against all attack methods in those benchmarks. A real-world attack on a lightweight HTTP server was also mitigated using our scheme.

Future work involves extending the binary rewriter to work on larger binaries and testing against more real-world attacks. Interesting avenues for future research are software diversification with a binary rewriter and self-healing techniques with binary rewriting.

Bibliography

[1] S.W. Boyd, G.S. Kc, M.E. Locasto, A.D. Keromytis, and V. Prevelakis. On The General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2006.

[2] D.L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[4] D. Maier J. Walpole P. Bakke S. Beattie A. Grier P. Wagle Q. Zhang C. Cowan, C. Pu and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Symposium*, pages 63–78. USENIX Association, 1998.

[5] R. Muth H. Patil A. Klauser G. Lowney S. Wallace V. J. Reddi C.-K. Luk, R. Cohn and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language design and implementation*, pages 190–200, 2005.

[6] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *discex*, page 1119. Published by the IEEE Computer Society, 2000.

[7] S. Designer. return-to-libc attack. *Bugtraq, Aug*, 1997.

[8] U. Erlingsson. Low-level software security: Attacks and defenses. *Foundations of Security Analysis and Design IV*, 4677:92–134, 2007.

[9] H. ETO and K. Yoda. propolice: Improved Stack-smashing Attack Detection. *Transactions*, 43(12):4034–4041, 2002.

[10] Alan Eustace and Amitabh Srivastava. Atom: a flexible interface for building high performance program analysis tools. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[11] JK Hollingsworth, BP Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 841–850, 1994.

[12] W. Hu, J. Hiser, D. Williams, A. Filipi, J.W. Davidson, D. Evans, J.C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Conference on Virtual Execution Environments, Ottawa, Canada*, 2006.

[13] U. Erlingsson M. Abadi, M. Budiu and J. Jigatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[14] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):100, 2007.

[15] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):1996–11, 1996.

[16] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, page 1. USENIX Association, 1997.

[17] L.-C. Lam S. Nanda, W. Li and T. Chiueh. Bird: Binary interpretation using runtime disassembly. In *CGO'06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370. IEEE Computer Society, 2006.

[18] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001).* Citeseer, 2001.

[19] 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*, page 561. ACM, 2007.

[20] H. Shacham, M. Page, B. Pfaff, E.J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[21] P.X. Team. Documentation for the PaX project. *Homepage of The PaX Team*, 2003.

[22] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, Athens, December 2005. IEEE.