View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

RUNTIME BINARY ANALYSIS FOR SECURITY

SARAVANAN SINNADURAI S/O GUNACHILAN

NATIONAL UNIVERSITY OF SINGAPORE

2007

RUNTIME BINARY ANALYSIS FOR SECURITY

SARAVANAN SINNADURAI S/O GUNACHILAN
(B.Comp.(Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2007

Acknowledgements

I am greatly indebted to A/P Wong Weng Fai for his invaluable assistance and guidance in the course of this project. I would also like to acknowledge Mr Zhao Qin, to whom I am thankful for all help rendered. I am also thankful to Mr Timothy Tsai's useful insights and to Mr John Wilander for providing the testbed program.

Table of Contents

Acknowledgements .... **i**

Table of Contents .... **ii**

Summary .... **vi**

List of Tables .... **viii**

List of Figures .... **ix**

Chapter 1 Introduction **1**

1.1 Background 4

1.2 Objective 5

1.3 Roadmap 6

Chapter 2 Executable File Formats **7**

2.1 Executable And Linkable File (ELF) Format 7

2.1.1 Accessing Symbols In Shared Library 10

2.2 Microsoft Portable Executable (PE) Format 11

2.2.1 Accessing Symbols In Shared Library 13

Chapter 3 Previous Work **15**

3.1 Techniques 15

3.1.1 Tools For Linux 16

3.1.2 Tools For Windows 24

3.1.3 Tools That Operate In Linux And Windows 25

Chapter 4 DynamoRIO Operations 27

4.1 Basic Block Cache 28

4.2 Trace Cache 29

4.3 DynamoRIO Interface 29

Chapter 5 Security Modules 31

5.1 Return Address Defense 31

5.1.1 Return Address Modification Exploit 32

5.1.2 Preventing Return Address Attack In TRUSS 34

5.2 Base Pointer Defense 35

5.2.1 Base Pointer Modification Exploit 36

5.2.2 Preventing Base Pointer Attack In TRUSS 37

5.3 Global Offset Table Defense 37

5.3.1 Global Offset Table Modification Exploit 39

5.3.2 Preventing Global Offset Table Attack In TRUSS 41

5.4 Format String Defense 42

5.4.1 Reading The Stack 43

5.4.2 Writing Into Arbitrary Memory Locations 45

5.4.3 Format String Exploit 46

5.4.4 Preventing Format String Attack In TRUSS 49

5.5 Vulnerable C Library Functions Defense 51

5.5.1 Preventing Vulnerable C Library Functions Attack In TRUSS 52

5.5.2 Vulnerable C Library Functions BSS/DATA/HEAP Overflow Exploit 54

5.5.3 Preventing BSS/DATA/HEAP Overflow Attack In TRUSS 57

5.6 Longjmp Buffer Defense 58

5.6.1 Longjmp Buffer Modification Exploit..60

5.6.2 Preventing Longjmp Buffer Modification Attack In TRUSS....................61

5.7 Function Pointer Defense...61

5.7.1 Function Pointer Modification Exploit...62

5.7.2 Preventing Function Pointer Modification Attack In TRUSS...................63

Chapter 6 Security Evaluations..64

6.1 Security Evaluation With John Wilander's Testbed Of Twenty Buffer Overflow Attacks 64

6.2 Security Evaluation With BASS...65

6.3 Security Evaluation With Libsafe Exploits..66

6.4 Security Evaluation With cOntext's GOT Attack 67

6.5 TRUSS vs Four Different Tricks To Bypass StackShield And StackGuard Protection 68

Chapter 7 Experiments...71

7.1 Experimental Setup...71

7.2 Performance Test With DynamoRIO's Profiling 71

7.3 Performance Test With SPEC CINT2000 Benchmark Programs.............71

7.4 Performance Test With Bapco Sysmark Benchmark Programs 74

7.5 Performance Of LibSafe And StackShield With SPEC CINT2000 Benchmark Programs 75

Chapter 8 Conclusion 78

8.1 Limitations 79

8.2 Future Research 80

References 81

Summary

Exploitation of buffer overflow vulnerabilities constitutes a significant portion of security attacks in computer systems. Common buffer overflow attacks include return address attacks, format string attacks, vulnerable C function attacks, stack-smashing attacks, heap overflows and GOT modifications. The aim of these attacks is typically to hijack critical information in the process address space so as to redirect the program's control flow to any malicious code injected into the process memory. Previous solutions to these problems are based either on hardware or compiler. The former requires special hardware while the latter requires the source code of the application.

In this thesis, I have introduced a runtime security mechanism - Transparent RUntime Security Suite (TRUSS) - that can protect applications against common buffer overflow attacks. The objective of TRUSS is to protect applications against buffer overflow attacks during execution. TRUSS works with the binaries of the applications and it does not require the applications' source code. Furthermore, TRUSS does not require any modification to the system that executes the applications. Many previous tools in this area focused their security effort on some specific vulnerability. Unlike such tools, TRUSS consolidates a number of techniques that can monitor various parts of the memory to detect, prevent and protect against buffer overruns. Additionally, TRUSS includes a few novel techniques to thwart attacks on the Global Offset Table entries and heap memory.

TRUSS is built as a client program in DynamoRIO, a dynamic binary rewriting framework. DynamoRIO is implemented on both Windows and Linux. Hence, this

scheme is able to protect applications on both operating systems. TRUSS has been successfully tested on the SPEC CINT2000 benchmark programs (on both Windows and Linux), on John Wilander's "Dynamic testbed for twenty buffer overflow attacks", on James Poe's and Tao Li's "BASS - A Benchmarking suite for evaluating Architectural Security Systems" as well as on Microsoft Access, PowerPoint, Excel and Word 2002. This thesis includes the implementation details of TRUSS. It also provides a performance evaluation, which will show that TRUSS is able to operate with an average overhead factor of up to 0.5 in Linux and 1.5 in Windows.

List of Tables

Table 1: Security performance on John Wilander's testbed of twenty buffer overflow attacks in Linux 65

Table 2: Security performance on John Wilander's testbed of buffer overflow attacks in Windows 65

Table 3: Security performance on BASS in Linux 66

Table 4: Security performance on Libsafe exploit code in Linux 67

Table 5: Performance of SPEC CINT2000 benchmark programs on Linux 72

Table 6: Performance of SPEC CINT2000 benchmark programs on Windows 73

Table 7: Performance of Sysmark benchmark programs on Windows 74

Table 8: Number of call and return pairs (in millions) in SPEC CINT2000 benchmark programs for Windows 75

Table 9: Number of call and return instructions (in millions) in Sysmark benchmark programs for Windows 75

Table 10: Performance of LibSafe with SPEC CINT2000 benchmarks in Linux 76

Table 11: Performance of StackShield-0.7 with SPEC CINT2000 benchmarks in Linux 77

List of Figures

Figure 1: Software vulnerabilities reported to CERT 2

Figure 2: Linking view of the ELF file 9

Figure 3: Execution view of the ELF file 9

Figure 4: Symbol table entry information..9

Figure 5: PE file layout 12

Figure 6: COFF file layout...12

Figure 7: Operations of DynamoRIO 28

Figure 8: Layout of a x86 process stack frame 32

Figure 9: Vulnerable C function - 1 33

Figure 10: Procedure linkage table 38

Figure 11: Vulnerable C function - 2 40

Figure 12: Copy of basic block in code cache 42

Figure 13: Bogus format string 44

Figure 14: Detecting position of format string in x86 process stack 44

Figure 15: Reading content at arbitrary memory location 45

Figure 16: Proper use of '%n' format specifier 45

Figure 17: Overwriting content of desired memory location.......................................46

Figure 18: Vulnerable C function - 3 47

Figure 19: Saturating counter approach...48

Figure 20: Heap memory chunks 54

Figure 21: Memory management macro..55

Figure 22: Vulnerable C program - 4..56

Figure 23: Symbol structure...57

Figure 24: Setjmp/Longjmp buffer 59
Figure 25: Setjmp/Longjmp example 59
Figure 26: Function pointer assembly code 62
Figure 27: Vulnerable C program - 5 62
Figure 28: Performance of SPEC CINT2000 benchmark programs on Linux 72
Figure 29: Performance of SPEC CINT2000 benchmark programs on Windows 73
Figure 30: Performance of Sysmark benchmark programs on Windows 74

Chapter 1

Introduction

Computer security is a field in computer science that is concerned with the control of menace associated with computer use. It is essential that any application that is executed in computer systems is safe and does not compromise with the security of the systems. The widespread viruses, worms and Trojan horses have the ability to intrude into systems and either steal critical information from the systems illegally or take control of the systems at privileged levels to perform unauthorized operations. It is therefore extremely imperative to protect computer systems from these malwares.

The rapid development of the Internet has further spawned a sharp increase in the number of computer systems being violated by malicious attacks. From the time of the infamous Internet Worm written by Robert T. Morris in 1988 [35], several security breaches causing much damage to systems have been reported. On 4th May 2000, a virus known as "I Love You" spread through Asia, Europe and the US within five hours via e-mails. It was estimated that the damage caused by the virus resulted in a loss of nearly one billion dollars [30]. In the following year, "Code Red Worm" spread over the Internet and more than 359,000 computers connected to the Internet were infected with the worm in less than 14 hours. The damage caused by this worm was estimated to be $2.6 billion [35].

Software vulnerabilities have been prevalent in applications since 1960s [14]. Figure 1 shows the number of security alerts reported by CERT between 1995 and 2005. A total of 22,716 vulnerabilities were reported within the 10-year period [13].

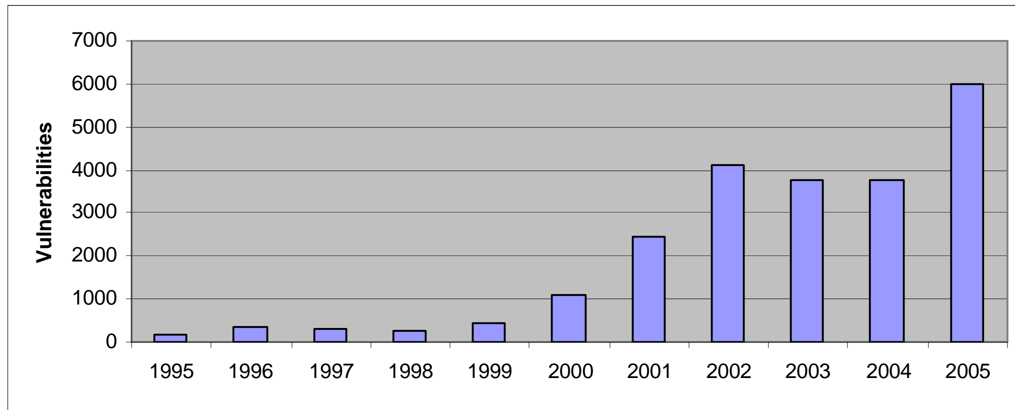

Figure 1: Software vulnerabilities reported to CERT

One common technique employed by these malwares is known as buffer overflows. The core of this security vulnerability lies in the use of programming languages such as C and C++. These programming languages tend to compromise safety of the application for efficiency. For instance, it is essential for any application to ensure that every access to an element in an array is safe. However, the compilers in these languages do not perform such checks automatically. In addition, the notions between arrays and pointers to data structures are used interchangeably. This makes it more difficult to monitor the violation of buffer limits in programs. Yet, due to legacy as well as the continued popularity of these programming languages, the problem cannot be solved easily by abandoning them in favour of safer ones.

Buffer overflow occurs when buffers allocated in a program overflow upon copying data into the buffer of a size larger than its capacity. This results in the excess data overwriting the adjacent memory locations. In this way, if critical control information was stored in the adjacent memory locations, a well-crafted data can overwrite the memory locations and cause the program to deviate from its execution flow and in turn, execute any malicious code. Return addresses, frame pointers,

Global Offset Table (GOT) entries, `longjmp()` buffer and function pointers are such vulnerable critical information that resides in the process memory.

A variant of buffer overflow attack is the format string attack. Format string vulnerability is based on the lack of safety checks by the C compilers (as mentioned earlier) and on the implementation of some functions in the standard C library. The standard C library provides a list of functions known as format functions, which accept a variable number of arguments. One of these arguments is the format string. These functions evaluate the format string and convert the arguments following the format string into a suitable form to be passed into the output stream. However, these functions do not check the attributes of the arguments. Moreover, they do not validate if the function parameters are indeed passed by the caller. There is also a lack of control mechanism to prevent any procedure that evaluates the format string from accessing erroneous memory locations. This vulnerability is manifested mainly in interactive computer systems. Apart from format functions, a number of other functions implemented in the standard C library are also vulnerable to buffer overflow attacks.

This situation raises a serious cause of concern because applications that utilize these library functions become vulnerable to attacks and this in turn compromises the security of the computer system, which executes these applications. Hence, methods that protect computer systems from becoming vulnerable because of such applications are vital. This research paper introduces a runtime security mechanism, which I refer to as Transparent RUntime Security Suite (henceforth, TRUSS). TRUSS provides a set of techniques to protect computer systems against common types of buffer overflow attacks and format string attacks in both Linux and Windows.

1.1 Background

Buffer overflow attacks fundamentally aim to achieve two goals. Firstly, an adversary has to inject a malicious code into the process memory. This is usually a small sequence of instructions that can invoke a shell on the system and pass control to the adversary with the privilege of the user. Secondly, the adversary has to change the execution flow to point to the start of the malicious code residing in memory. An attack is successful only when both the goals are achieved [57]. There is no major security concern if only one of the goals is pulled off by the adversary. Malicious code residing in the computer system does not cause any damage to the system unless it is invoked.

There are five main types of buffer overflow attacks. The most common way to perform a buffer overflow attack is by modifying return addresses. These addresses are usually modified via stack smashing – a method to overwrite a buffer with data more than the size of the buffer itself. Overflowed buffers will usually cause the program to crash. However, adversaries can use well-crafted data to modify specific location in the memory where the return address resides. This can change the execution flow of the program and make the program counter (also known as instruction pointer) to point to the start of the malicious code.

The second type of attack targets the stack frame pointer. This attack uses the stack-smashing technique as well. The adversary has to insert a fake stack frame into the process stack with a return address pointing to the start of the malicious code. The overflowing data has to overwrite the value of a stored frame pointer with the address of the fake stack frame. Hence, when a subroutine returns, control will be passed on to the fake stack frame and it will perform a return again directing the flow of control to the attack code.

The next type of attack aims to redirect a function pointer in the program to point to the attack code. This function pointer can be allocated in the stack or heap for the attack to succeed. A buffer is overflowed until the memory location of the function pointer is reached. The start address of malicious code is then copied to the function pointer. Thus, when the function pointer is used in the program, it will direct the execution flow to execute the attack code.

The fourth type of attack targets the GOT entries. The GOT is used by applications in Linux to redirect function calls between the executable and a shared object or between different shared objects. This is a point of control flow transfer and if an adversary manages to hijack this data and overwrite it with the start address of any malware, the malware will be successfully executed.

The last type of attack uses the `setjmp()/longjmp()` buffers. A `setjmp()` call saves the environment information in a buffer. This data includes the contents of the program counter, the stack pointer and the frame pointer. The program counter contains the address of the instruction to be executed next. If an adversary manages to modify the program counter to point to the start of attack code, control will be transferred to the attack code when `longjmp()` restores the environment information [14].

1.2 Objective

This paper presents TRUSS, which is to be used on applications that execute on Intel x86 architecture. The main objective of TRUSS is to provide a defensive mechanism that makes minimal modification to the original executable and incur low overheads during runtime. In order to efficiently work on an application, TRUSS employs

DynamoRIO, a binary instrumentation tool. DynamoRIO is a runtime code manipulation system and TRUSS functions within this system. DynamoRIO is an IA-32 implementation of the original PA-RISC based Dynamo Project [5]. It supports efficient, transparent and comprehensive manipulation of applications running on Windows or Linux operating systems. DynamoRIO provides APIs to hook each basic block in the application before the block gets executed. This provides ideal opening to analyze the instructions and to implement defensive measures.

The suite of safety techniques presented in this research paper is to provide efficient runtime protection for applications, inclusive of those without any source code. The primary aim of this research paper is to present a defense suite that protects existing applications from return address modification attacks, format string attacks, stack smashing attacks, GOT attacks, heap buffer attacks, `longjmp()` buffer attacks and attacks that take advantage of the vulnerable functions in the standard C library with considerably low overhead.

1.3 Roadmap

In Chapter 2, I will look at the two different types of executable file formats used in Linux and Windows operating systems. Chapter 3 will review some of the related work in this field. In Chapter 4, I will look at the underlying tool, DynamoRIO, upon which TRUSS is built. Chapter 5 will explain the security modules included in TRUSS. In Chapters 6 and 7, I will analyze the security performance and overhead incurred by TRUSS respectively and I will conclude in Chapter 8.

Chapter 2

Executable File Formats

In this chapter, I will briefly describe the Executable and Linkable File (ELF) format and the Microsoft Portable Executable (PE) file format utilized by binaries in Linux and Windows respectively. Understanding these formats is essential as most of the information used by TRUSS is extracted from the binaries.

*2.*1 Executable And Linkable File (ELF) Format

ELF [33] defines a binary interface that allows the linking of several object files and creates a process image during execution. There are three main types of object files.

- *A relocatable file stores code and data suitable for linking with other object files in order to create an executable or a shared object file.*
- *An executable file contains a program suitable for execution.*
- *A shared object file stores code and data suitable for two functions:*
    - *First, the link editor may process it with other relocatable and shared object files to create another object file.*
    - *Second, the dynamic linker may combine it with an executable file and other shared objects to create a process image.*

The ELF object file provides a dual view of the file's contents - linking view and execution view. While linking view is required to build the program and it divides the

file content into sections that contain the text, data and other information such as the symbol tables and relocation tables; execution view is needed to form the program image. It describes how the various parts of the file should be mapped into the memory to form the process image. Execution view divides the content into segments with differing permissions (e.g. read/write/executable) assigned to them.

The ELF file stores the ELF header at the beginning of the file and this header holds a roadmap that describes the file's organization. In addition, the ELF file contains a program header table and a section header table. The program header describes the ELF file's execution view while the section headers describe the ELF file's link view. The program header instructs the system on creating a process image. Only ELF files that are used to build a process image have a program header table. Relocatable ELF files do not need one. Each program header describes a segment in the ELF file. A segment can contain one or more sections. Sections with similar access permissions can be grouped into the same segments. Each segment corresponds to a segment in the virtual address space. The process image is made up of segments of memory that hold code, data and stack.

The section header table is an array of section header structures. Each section header contains information describing the file's sections. Every section in the ELF file occupies a contiguous block of memory and no two sections overlap. The sections in ELF are pre-defined and hold program and control information. These sections are used by the operating system and have different types and attributes for different operating systems. An illustration of the dual view of an ELF file is depicted in Figures 2 and 3.

ELF Header
Program Header Table Optional
Section 1
…
Section n
…
…
Section Header Table

Figure 2: Linking view of the ELF file

ELF Header
Program Header Table
Segment 1
Segment 2
…
Section Header Table Optional

Figure 3: Execution view of the ELF file

Linking two files essentially means to resolve the symbols defined in one object file and used in another. This process uses a number of sections. To facilitate linking, the ELF file contains two symbol tables that store a list of all the symbols used or globally defined in an object file. The dynamic linker exclusively uses the dynamic symbol table and the static linker uses the other symbol table. Each symbol entry contains information regarding the name, value and section index as depicted in Figure 4.

```
typedef struct {
      Elf32_Word st_name;
      Elf32_Addr st_value;
      Elf32_Word st_size;
      unsigned char st_info;
      unsigned char st_other;
      Elf32_Half st_shndx;
} Elf32_Sym;
```

st_name	This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names.
st_value	This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, and so on.
st_size	Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.
st_info	This member specifies the symbol's type and binding attributes.
st_other	This member currently holds 0 and has no defined meaning.
st_shndx	Every symbol table entry is defined in relation to some section. This member holds the relevant section header table index.

Figure 4: Symbol table entry information

Relocation is the process of connecting symbolic references to symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address during execution. In other words, relocatable files must contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. In the ELF file, relocations are needed because the virtual address of all the symbols is only available at runtime. The virtual address of a function or a data item inside a shared library is not known until the program starts to execute. The ELF file thus makes use of the GOT to store all the symbols that have to be resolved at runtime. Each entry in the GOT specifies a symbol and the first time a symbol is used in the program, the dynamic linker is invoked to go through all the loaded libraries and to perform the specified relocation [3].

2.1.1 Accessing Symbols In Shared Library

In Linux, instructions in dynamically linked libraries are not bound to the executable at link time. Dynamic linking defers much of the linking process until a program starts running. Therefore, it is not possible to know the addresses of functions defined in shared libraries before the program begins execution. In Linux, however, a programming interface is provided to dynamic linking loader. This interface allows shared libraries to be loaded explicitly via the `dlopen()` call.

`dlopen()` loads a dynamic library and returns a handle for it. If the absolute path for the library is not provided, the library is searched for in the following locations:

- *A list of directories in the user's LD_LIBRARY_PATH environment variable.*

- *The list of libraries cached in /etc/ld.so.cache.*
- */lib directory, followed by /usr/lib directory.*

If library filename is a `NULL` pointer, a handle for the main program is returned.

External references in the library are resolved using the libraries in the library's dependency list and any other libraries previously opened with the `RTLD_GLOBAL` flag. If the executable was linked with the flag `-rdynamic`, the global symbols in the executable will also be used to resolve references in a dynamically linked library.

`dlopen()` has to load the dynamic library with the flag `RTLD_NOW`. This will resolve all undefined symbols before `dlopen()` returns. Optionally, `RTLD_GLOBAL` may be added to flag, in which case, the external symbols defined in the library will be made available to subsequently loaded libraries. `dlsym()`accepts the handle of a dynamic library returned by `dlopen()` and the symbol name. It returns the address where that symbol is loaded. If the symbol is not found, `dlsym()` returns a null value. TRUSS employs this technique to obtain the address of functions defined in shared libraries in order to intercept them.

2.2 Microsoft Portable Executable (PE) Format

Microsoft PE files [36] are intended for a paged environment. Pages from a PE file are usually mapped directly into memory and executed, similar to an ELF executable. There are two types of PE files.

- *EXE programs*
- *DLL shared libraries (known as dynamic-link library)*

The format of the two files is the same. Only a status bit differentiates the two PEs. Both types of files can contain a list of exported functions and data that can be used by other PE files loaded into the same address space. A list of imported functions and data that has to be resolved from other PEs at load time can also be found in the PE files. However, the `EXE` files do not export functions.

In addition, there exists another type of file known as `COFF`. `COFF` is the Common Object File Format. This is the type of an object file under Microsoft Windows. The structure of the `COFF` is similar to the ELF relocatable file. The layouts of the Microsoft PE file and `COFF` file are illustrated in Figures 5 and 6.

MS-DOS 2.0 Compatible EXE Header
unused
OEM Identifier OEM Information Offset to PE Header
MS-DOS 2.0 Stub Program and Relocation Table
Unused
PE Header (aligned on 8-byte boundary)
Section Headers
Image Pages: import info export info base relocations resource info

Figure 5: PE file layout

Microsoft COFF Header
Section Headers
Raw Data: code data debug info relocations

Figure 6: COFF file layout

The PE file header consists of a Microsoft MS-DOS stub, the PE signature, the `COFF` file header and an optional header. A `COFF` object file header consists of a `COFF` file header and an optional header. In both cases, the section table follows the file headers immediately. Each row of the section table is a section header, which describes a particular section. Each section is physically aligned on a disk block boundary and

logically aligned on a memory page boundary (4096 on the x86). The linker creates a PE file for a specific target address at which the file will be mapped [36].

2.2.1 Accessing Symbols In Shared Library

Unlike Linux, Microsoft Windows does not provide an interface that can be used to retrieve the address of imported functions. However, the PE file does contain an imported function table that stores the names of imported functions and the libraries where the functions are defined. This table is known as the Import Address Table (IAT). I will now explain the process of accessing the addresses of functions from the IAT.

In order to work with the Microsoft PE, a handle to the executable has to be obtained. The Win32 API provides a `GetModuleHandle()` call to retrieve a module handle, known as `HMODULE`, for the specified module. This function returns a handle to the file used to create the calling process when parameter is NULL. The base of the module, which contains the DOS header is then retrieved from the `HMODULE` handle. The optional header in the PE is then accessed to obtain the data directory, which contains the various sections in the PE. The import descriptor entry is then accessed from the data directory. This is the section that contains the information about the imported libraries.

The import descriptor entry contains two data thunks - `OriginalFirstThunk` and `FirstThunk`. The import information is in fact stored in two tables. The `OriginalFirstThunk` contains a reference to the Import Name Table for the module and `FirstThunk` contains a reference to the Import Address Table for the module. The two tables contain entries of `IMPORT_BY_NAME` structure. This structure

contains both the address and the name of the imported function and hence, the desired function address can be retrieved from the PE once the program is loaded. The Import Name Table and the Import Address Table are merely two copies of the same table. The PE loader, which maps the function names to addresses when loading the PE, uses these two tables. Since the addresses of the various functions are unknown at compile time, the loader performs the resolution and replaces each entry in the Import Address Table with the actual address of the function. If there arises a need to know which function an address corresponds to, the Import Name Table can then be used. TRUSS employs this technique to obtain the address of standard C functions defined in shared libraries in Windows.

Chapter 3

Previous Work

Security modules that provide protection to applications can be employed at three different levels in computer systems. Firstly, it is possible to modify the hardware of the existing systems or add hardware modules to the systems to monitor applications. Some systems include an additional processor solely to perform security checks. This is a fast but rather expensive method. Secondly, security measures can be included into compilers. Compilers can insert additional instructions into the applications to perform checks and prevent any violations. This would require recompilation of the source code but to most legacy applications, the source code is not available. Thirdly, defensive measures can be applied simultaneously on the applications that are executed in the system. This, however, requires additional tools to decode and analyze the instructions in the applications and it is likely to incur higher time overhead than the earlier methods.

In this chapter, I will briefly review the currently available tools that claim to protect systems against the attacks under discussion.

3.1 Techniques

I will review tools built for Linux operating system followed by tools built for Windows operating system. I will then review tools that are able to operate on both Linux and Windows.

3.1.1 Tools For Linux

SmashGuard [40] is a hardware solution that protects function return addresses in the process stack from buffer overflow attacks. This technique modifies the semantics of call and return instructions in the instruction set architecture. This modification enables functions to store a copy of the return addresses in a memory segment during calls and to perform comparison with the stored return addresses upon return instructions. In an event where the return address in the stack does not match with the stored copy, the processor raises a hardware exception and terminates the execution. Modern CPUs contain a considerable amount of memory on the chip itself. SmashGuard utilizes this memory space to create a data stack. This is used for storing function return addresses. Hence, by modifying the instruction set architecture, SmashGuard is able to provide protection to applications without modifying the application.

StackGuard [19] is a compiler extension that enhances the executable produced by the compiler so that the executable is protected against stack-based buffer overflow attacks. This technique specifically targets the return addresses of the functions in the stack. StackGuard can detect changes to active return addresses before a function returns. In order to prevent changes to active return addresses, StackGuard stores a canary (known 4-byte value) adjacent to the return address in the stack. When the function returns, checks are carried out to ensure that the canary is unaltered. The values of the canary are selected randomly so that adversaries will not be able to skip or simulate the canary. In addition, StackGuard uses of MemGuard to prevent any modification to the return address. MemGuard write-protects pages and generates exceptions when protected pages are accessed. Thus, StackGuard write-protects pages

containing the stack but restores write privilege to the topmost page to allow the program to access the stack variables.

StackShield [55] is a compiler extension implemented by Vendicator. It can work with GCC compiler to provide protection for applications that are compiled with it. During compilation, StackShield inserts instructions into the program to make copies of function return addresses and saves them in a data segment known as `Global_Ret_Stack`. These instructions are inserted after call instructions and before return instructions. During execution of the program, when a function call is invoked, the function return address is stored into `Global_Ret_Stack` and before a return instruction is executed, the return address in the process stack and the copy in the data segment are compared. An alert is raised if the addresses do not match. StackShield, however, will only protect an application if the application is compiled with it. Hence, the source code of the application is necessary in order to utilize StackShield. This, however, is not possible for many legacy applications where only the binaries are available.

Return Address Defender (RAD) [14] is another compiler extension which provides a compile-time solution to buffer overflow attacks targeting return addresses. Like StackShield, RAD automatically adds instructions into applications that are compiled with it. Protection code is inserted into the function prologues and epilogues. Hence, when a program is executed, any function invocation will copy the return address to a memory segment called `Return Address Repository`. When return instructions are executed, the return address on the process stack is compared against the stored copy. A mismatch would raise an exception. In addition, RAD marks the `Return Address Repository` as read-only to ensure the credibility of the return addresses stored in the memory segment. It also has the option of marking

only the neighbouring pages of the `Return Address Repository` as read-only; this causes less performance degradation compared to the previous method. However, like StackShield, RAD too requires the source code of the program in order to provide protection.

Libsafe [54] uses a technique that can intercept invocations of most standard C functions and performs safety checks on the arguments. If the checks pass, either the original functions or the equivalent alternatives get executed. Suppose the check fails, the Libsafe will log the necessary information and terminate the application. Libsafe functions as a dynamically linked library that is activated by explicitly specifying itself in the `LD_PRELOAD` environment variable. By doing so, the library is loaded even before the program begins execution. When Libsafe intercepts a C library function, it performs frame pointer checks and frame span checks. Frame pointer check ensures that "`%n`" format specifier does not modify any return addresses or frame pointers. Frame span check ensures that no buffer is overflowed beyond the current frame pointer in the process stack. The advantage of Libsafe is that it does not require the source code of the application; neither does it modify any part of the application.

Guarded Memory Move (GMM) [34] is a technique that functions in a similar way to Libsafe. GMM also functions as a dynamically linked library that is activated by explicitly specifying itself in the `LD_PRELOAD` environment variable. During dynamic symbol resolution, the loader checks libraries in the `LD_PRELOAD` variable and the functions found in GMM library are executed instead of the C library functions. GMM's alternative C functions store content of some memory locations above the current stack frame and three previous return addresses into private location during a call instruction. When the subroutine returns, the existing memory content is

compared with the stored data. A violation is signalled if a mismatch is found. In this way, the safety of the application is assured. Just like Libsafe, GMM does not require source code and it does not modify any part of the application. Both LibSafe and GMM provide safe versions of format functions. Hence, these applications can also be considered as format string defense tools.

Libverify [6] is a proposed solution that works on binaries at runtime to provide protection against buffer overflow attacks that target function return addresses. It works on Linux operating system. Libverify works as a dynamically linked library that is activated by explicitly specifying it in the `LD_PRELOAD` environment variable. By doing so, the library is loaded even before the program begins execution. The `_init()` function in Libverify will modify the application such that every function invocation and return instruction will invoke the checking functions in its library. Libverify copies every function instruction to the heap and appends a branch instruction to an entry wrapper function. This entry wrapper function stores a copy of the return address in a canary stack (which resides in the heap memory) and then branches back to the original function. Likewise, the return instructions are replaced with an exit wrapper function, which in turn verifies the return address in the process stack with the corresponding value stored in the canary stack. Upon a match, the process will execute the return instruction and continue with its execution flow. Any mismatch will create a syslog entry, output an error message and terminate the process. Libverify provides dynamic protection to executables. Its ability to work on binaries without the source code is a major advantage. However, this software has not been released.

FormatGuard [17] employs a defense mechanism that prevents exploitations due to format strings. In order to prevent format string attacks, FormatGuard compares the

number of actual arguments provided for format functions against the number of arguments specified in the format string. If the number of arguments called for were more than the number of arguments passed to the functions, FormatGuard would classify the case as an attack. It will log the attempt and abort the program. Counting the actual number of arguments that are passed to format functions is a difficult task. This is because the arguments are passed as a variable list that does not provide any counting mechanism. FormatGuard counters this problem by using CPP variable argument syntax for argument counting and this function is inserted into the `stdio.h` file. FormatGuard is packaged as a modified implementation of `glibc 2.2` libraries. Thus, applications have to be complied with the modified libraries to utilize FormatGuard's protection. In addition, the CPP syntax deployed in FormatGuard may cause compatibility problems when executing some C programs.

White listing [49] is a technique used to control memory modification via format functions. It maintains a list of memory address ranges that allow memory modification. This list is referred to as the white list. During runtime, the technique inserts and removes memory address ranges to the white list and from it. Hence, a format function can check with the white list to verify whether a valid integer pointer is being modified, when handling a '`%n`' format specifier. The white listing scheme proves to be rather flexible in that it is possible to enforce different policies - one at a time - using the same technique. Firstly, removing all memory addresses from the white list can enforce a 'no memory write' scheme. Secondly, 'write anywhere' can be put in place by specifying all possible address ranges in the white list. Lastly, adding a user specified range of addresses into the white list can enforce a 'restricted write'. White listing can automatically register the address range before a format function and unregister it after the function. Hence, when a format function

encounters a '`%n`' specifier, it will always scan the white list to check if the memory addresses to be written to is safe for a write operation. White listing uses the C Intermediate Language System (CIL) to implement its defense mechanism. CIL provides the tools that permit easy analysis and source-to-source transformation of C programs. Hence, the C program is transformed into the intermediate language, after which the additional instructions for protection are inserted. As such, this technique requires the source code of the application.

C Range Error Detector (CRED) [51] is a safe C compiler. It uses an object tree, containing the memory ranges occupied by all the buffers used in a program. When an object is created, it is added to the tree and when it is destroyed or goes out of scope, it is removed from the tree. A pointer operation is considered illegal if a memory location that is not within the tree is accessed. Moreover, CRED does not change the representation of pointers within the application. As such the instrumented code can interoperate with the unchecked code. Limitations in CRED include interchangeable use of structures and arrays and unverified accesses within library functions.

LibSafePlus [3] is a dynamically loadable library and it is an extension to LibSafe. LibSafePlus includes heap protection in addition to LibSafe's stack protection. It contains wrapper functions for unsafe standard C functions. These wrapper functions determine the target buffer sizes in any operation that writes to a buffer and ensures that the operation does not result in an overflow. Type Information Extractor and Depositor (TIED) is a tool that can extract debugging information from a program and can enhance the program with additional information regarding the sizes of all the buffers used in the program. LibSafePlus uses TIED to prevent buffer overflows. LibSafePlus requires the program to be compiled with the debugging option '`-g`'. This provides the executable with additional debugging sections. TIED can thus

examine this debugging information and extract the starting addresses and sizes of buffers used in the program. The wrapper function in LibSafePlus uses this information to protect the unsafe functions in standard C library so as to buffer overflows.

Transparent Runtime Randomization (TRR) [58] proposes an idea to randomize runtime locations of critical data in the application in order to make it difficult for an adversary to determine the location of the data through experimentation. TRR essentially modifies the dynamic loader to relocate the user stack, shared libraries, user heap and the GOT to different memory locations. This technique handles both position dependent memory region and position independent memory region. User stack, user heap and shared libraries are position independent, as these regions do not have complex inherent relationship with other parts of the application. The GOT is a position dependent region because this region is fixed inside the program's data segment and any uncoordinated relocation will break references within the application. The random offsets used to relocate the critical data regions are chosen in a way that the memory regions do not overlap but have sufficient space to grow. The randomizations in this technique do make it more difficult for attacks to succeed but do not completely thwart attacks because the relocated memory regions are not protected in any way.

PointGuard [18] is a pointer protection technique that encrypts pointers when they are stored in memory and decrypts them when they are loaded into CPU registers. PointGuard is implemented as a compiler extension that modifies the intermediate syntax tree to insert encryption and decryption code. However, encryption only provides confidentiality. It does not guarantee the integrity of the encrypted values. In addition, although PointGuard imposes very small performance overhead for most

applications, it only protects the code pointers (function pointers and pointers in `longjmp()` buffers) and data pointers. It does not offer any protection for other program objects.

Propolice [22] is a compiler extension that can protect executables from stack smashing attacks. It places canaries in memory location between the frame pointers and local variables and ensures that the memory locations adjacent to the canary are not altered. In addition, Propolice reorders the location of stack variables. It places local pointers below arrays. Pointers from the arguments are placed before the local variables. Having local pointers placed below arrays prevents attacks that attempt to overflow the arrays in order to modify the pointers. Placing pointers from the arguments before the local variables makes it more likely for the buffer overflow attacks to be detected.

One point to note is that GCC 4.1 incorporates a modified version of IBM 's ProPolice Stack Detector. This version includes more run-time and compile-time optimizations and function analysis. Hence, applications compiled with GCC 4.1 will include stack protection instructions by default.

Another class of defense techniques include those which support non-executable mappings on platforms. These techniques assign data segments such as the stack and heap as non-executable. In this way, any malicious code that resides in these data region will be denied execution privileges. Examples of such techniques are the Solaris/SPARC Non-Executable Stack Protection and NetBSD 2.0 Non-executable Stack and Heap Protection.

3.1.2 Tools For Windows

DOME [47] is a technique for detecting several classes of malicious code in applications. It uses static analysis to identify and to store locations of Win32 API calls within the application. It then monitors the executable to verify that all API calls are made from the observed addresses. DOME utilizes Detours [27] to intercept every API functions in an application. Detours is a library for intercepting Win32 functions on x86 machines. It intercepts functions by inserting additional code into the binary during execution. It can divert calls to Win32 functions so as to pass control to the user. It can also allow the original function to be invoked if needed. The underlying idea is to prevent any Win32 API invocation from unverified location, thus preventing the malicious code from invoking any API function.

Binary rewriting defense [44] is a binary level solution to foil buffer overflow attacks. This technique does not require the source code of the application. It protects the function return addresses by adding protection code at every function invocation in the binaries. This is done by static analysis without disturbing the procedure's execution flow. This technique requires tools to analyze the binary in order to identify each instruction. Binary rewriting method uses dissemblers to accurately trace the location of function invocations in the binary. In order to store copies of the return addresses, the binary rewriting method employs similar technique used in RAD. Function return addresses are stored in a repository upon function calls and a comparison is done before return instructions are executed. However, in contrast to previously mentioned techniques, binary rewriting method inserts additional protection code only for 'interesting functions'; these functions contain instructions to allocate and to free the memory for local variables. Thus, functions that do not contain any local variables are considered safe functions as stack based buffer overflow

cannot succeed in functions without local variables. However, static analysis of binaries cannot provide protection for dynamically linked libraries and Position Independent Code (PIC). Moreover, static analysis of binaries using dissemblers is not 100% accurate and it is still possible to miss vulnerable functions even after using it.

3.1.3 Tools That Operate In Linux And Windows

HeapShield [8] is a memory management approach that can prevent heap overflows. It basically modifies the free-list based heap as typically used in Linux and Windows to segregated-fits 'Big Bag of Pages' (BiBOP) style heap. The heap allocator divides memory into chunks that are multiples of the system page size. Objects of different sizes are allocated in different chunks. Object sizes and other metadata are stored either at the beginning of the chunks or in a page directory. This type of heap organization allows efficient calculation of available free space in allocated buffers and thus thwarts heap overflow attacks.

Secure execution via program shepherding [29] is a software technique that thwarts all attempts to hijack a program's control flow using security policies and binary rewriting techniques. It monitors control flow transfers during program execution. It employs the use of three techniques to enforce security policies. Program shepherding can restrict execution privileges based on code origins. It can restrict control transfer based on instruction class, source and target. In addition, program shepherding can place a sandbox on any type of application. Program shepherding has been built as an extension to a dynamic optimizer called RIO. RIO is built upon an IA32 version of

Dynamo. RIO is implemented for both Linux and Windows and it can execute large desktop applications and multi-threaded applications.

The No eXecute (NX) bit is a technology used to distinguish areas of memory for instructions storage and data storage. Any memory segment assigned with the NX bit will mean that it can only be used for storing data. No instructions can be executed from regions assigned with the NX bit. The general technique is known as executable space protection. It is used to prevent malicious software from taking over computers by inserting their code into another program's data storage area and running their own code from within this section.

In this thesis, I am introducing a new tool called TRUSS. TRUSS is a software technique that operates in both Windows and Linux. It aims to thwart common forms of buffer overflow attacks and it includes novel techniques to protect the GOT and heap buffers without modifying any part of the application. I employ a dynamic binary rewriting tool, DynamoRIO, to implement TRUSS. It is to be noted that this is the same tool that has been used in Program Shepherding. However, the DynamoRIO application used in this project has been modified to support self-modifying code.

Chapter 4

DynamoRIO Operations

The protection provided by TRUSS consists essentially of rather straightforward ideas. However, the major challenge is to provide an efficient method to perform binary instrumentation and to insert minimal checking instructions while bringing down the performance overhead. TRUSS uses DynamoRIO as its implementation platform.

DynamoRIO is a runtime code manipulation tool that supports code transformations in an application during the application's execution. Its operating procedure is illustrated in Figure 7. DynamoRIO maintains a code cache where it stores a copy of the application instructions. These instructions are stored in units of basic blocks such that each basic block ends with a control transfer instruction. The basic blocks in the code cache are used for execution. Hence, DynamoRIO constantly transfers control between instrumentation of basic blocks from the application code and execution of the basic blocks. DynamoRIO includes an important optimization technique to improve the application's performance. It contains a cache that stores a copy of contiguous sequences of basic blocks known as traces. These are basic blocks that are executed more than a default number of times. The control transfer instructions in these blocks are replaced with frequently used targets of indirect branches (inlined into the traces); these also include a check to verify the target of the branch instruction [11]. Traces improve the application's performance by allowing a processor's instruction decoder and a branch predictor to work more efficiently.

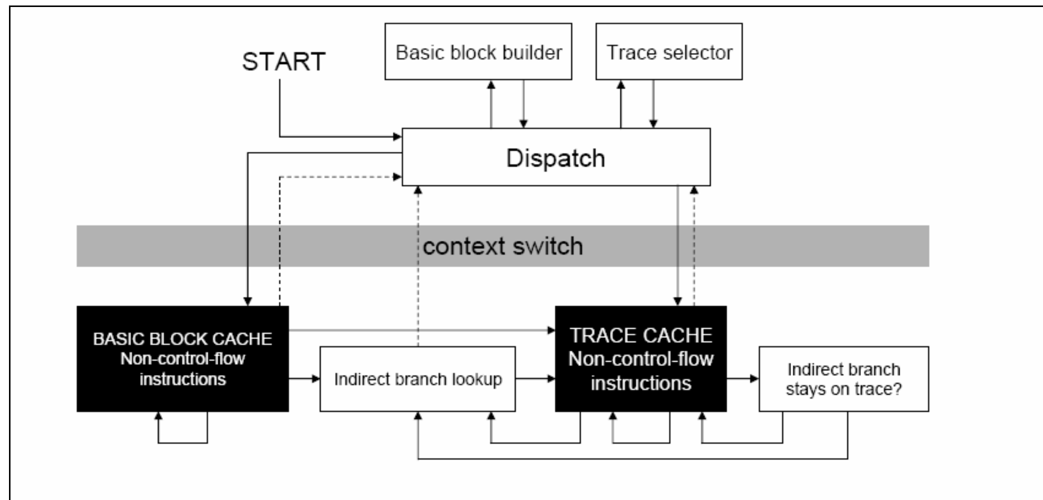

Figure 7: Operations of DynamoRIO [11]

4.1 Basic Block Cache

DynamoRIO begins its execution by copying the first basic block of an application into its basic block cache. This basic block extends across unconditional branch instructions and call instructions. The block ends with an indirect branch instruction or a conditional instruction. This block is then executed until the target address of the indirect branch or conditional instruction is resolved. Upon resolving the target address, the cache is searched for basic blocks with starting address corresponding to the target address. If a match is found, the target address is replaced with the start of that basic block. If no match is found, the target address is modified so as to return control to DynamoRIO. DynamoRIO would then continue to build a new basic block, starting at the target address. When this new block is added to the cache, the target address of the indirect branch or the conditional instruction is modified to point to the start of this basic block.

4.2 Trace Cache

Traces provide optimization within the DynamoRIO framework and also organize a sequence of commonly executed basic blocks into a single contiguous piece of code. This improves the efficiency of indirect branches and achieves a better code layout. The creation of a trace begins by looking for a suitable trace head. A trace head is either the target basic block of a backward branch or an exit from an existing trace. Each trace head is associated with a counter that gets incremented each time the block is executed. When the counter exceeds a threshold number, the sequence of basic blocks that follows the trace head is concatenated to the trace head. The trace terminates when a backward branch or a basic block that is part of another trace is reached [11]. During execution, the trace cache is searched first for matching traces before the basic block cache is.

4.3 DynamoRIO Interface

DynamoRIO provides client hooks like `dynamorio_basic_block()` and `dynamorio_trace()`, which are invoked whenever a basic block or a trace is to be added to the basic block cache or the trace cache respectively. It also incorporates a set of APIs that allows the basic blocks in the cache and the traces in the trace cache to be analyzed and manipulated. It also allows a user to build a client program, using the APIs, which can be attached to DynamoRIO so as to work on the application. The client program is compiled as a shared library and is loaded before DynamoRIO begins its routines. Hence, the defensive mechanism in the client program is able to intercept the application at the appropriate instances. All these features come at a cost

ranging from zero to thirty percent of time and memory overhead on both Windows and Linux [11].

Chapter 5

Security Modules

In this chapter, I discuss some common buffer overflow exploits and the defensive mechanisms incorporated in TRUSS to detect and to prevent such vulnerabilities and their implementation details.

5.1 Return Address Defense

A common form of attack is by means of modifying control information in the application address space and transferring the program control to any malicious code. The function return address is among the most vulnerable control information that resides in the process stack. Its role is to facilitate a function to continue executing the correct instructions following a subroutine. It thus plays a vital role in ensuring correct control flow within an application. However, no form of protection is provided by the application or the operating system in order to prevent any unauthorized modification of return addresses.

During a program execution, when a call instruction gets executed, the process will evaluate the address of the instruction that follows the subroutine. This evaluated address will then be pushed into the process stack. Following this operation, most subroutines will save the value of the frame pointer into the stack and assign it to the location pointed by the stack pointer register. This location is the start of the subroutine's active stack frame. Moreover, if the subroutine requires arguments, they

are saved in the process stack before the return address. Figure 8 shows the layout of the stack frame in the process stack.

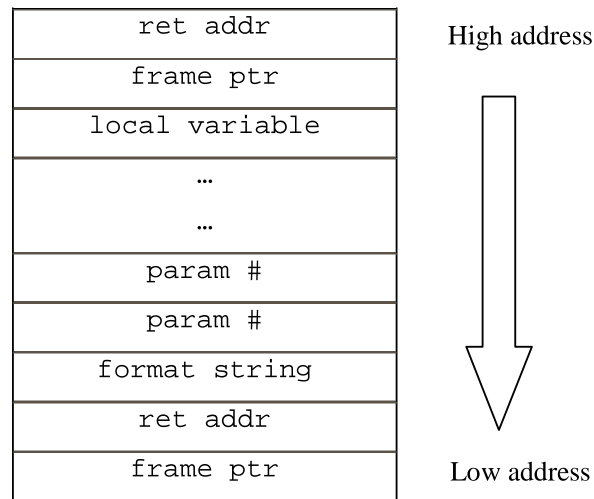

Figure 8: Layout of an x86 process stack frame

Upon completion of the subroutine, the previous value is restored in the frame pointer. This is usually performed explicitly by a `leave` or `pop` instruction. Consequently, the top of the stack now contains the return address. Hence, when a return instruction is executed, the instruction pointer is replaced with the value of the return address and execution control is passed back to the caller.

5.1.1 Return Address Modification Exploit

The above described method of calling subroutines and returning from subroutines assumes that the integrity of the stored return address was never violated. However, this is a naive assumption because the subroutine could have altered the return address illegally. This would in turn allow any malicious code (to which the return address was altered) to be executed when the subroutine returns. To illustrate how this might happen, consider the C function in Figure 9.

```
int foo(int a, int b)
{
      char homedir[100];
      ...
      strcpy(homedir,getenv("HOME"));
      ...
      return(1);
}
```

Figure 9: Vulnerable C function - 1

If the result from `getenv()` is of a size that is larger than the local variable `homedir`, then `strcpy()` will overwrite the memory locations adjacent to `homedir`, including the return address of the subroutine `foo()`. Consequently, when the subroutine is completed and it performs a return to pass control back to the caller function, the execution flow will be passed to the value that lies in the return address storage location. Usually, such situations will result in "segmentation fault" or "bus error" messages. However, adversaries can place a well-crafted value to replace the return address in order to redirect execution to the location of their choice.

A buffer overflow in a stack requires data that contains executable code, followed by enough repetitions of its address (in memory). The purpose of this is to redirect the program's execution flow to the inserted code. This is with the assumption that the inserted code is small enough to fit into the local buffer. However, such an attack requires the adversary to predetermine the exact starting address of the inserted code to replace the return address. Adversaries can get around this requirement by appending a sequence of `NOP` instructions before the executable code. Concatenating the sequence of `NOP` instructions creates a ramp. The modified return address, by pointing anywhere in the ramp, can enable a successful attack. While it still takes some effort to find the proper range, an adversary only needs to make a close guess.

A successful return address modification attack allows an adversary to execute instructions with the same privileges as that of the compromised program. If the

compromised program was running with the super-user privilege, the adversary can inject code to spawn a super-user shell and subsequently take control of the machine. In the case of worms, a copy of the worm program can get installed with super-user privilege and the system begins looking for more machines to infect [31].

5.1.2 Preventing Return Address Attack In TRUSS

TRUSS intercepts every call instruction and return instruction in a program to insert protection code. It uses the `dynamorio_basic_block()` function to interrupt the application. This is performed after DynamoRIO creates a basic block and before the block gets executed. In this function, TRUSS scans through every instruction in the basic block to identify the call and return instructions. When call and return instructions are encountered, TRUSS redirects the basic blocks to two different modules where additional instructions are inserted into them.

Following call instructions, TRUSS inserts instructions to calculate the return address from the basic block and store it into a separate memory segment. The return address is calculated by adding the address of the call instruction to the length of the instruction. This addition will give the address of the instruction that follows the call instruction in the caller function. The memory segment is dynamically allocated and is known as the `shadow_stack`. TRUSS also stores the location of the return address in the stack into the `shadow_stack`. Both the location and the value of the return address are stored in the same memory segment for space efficiency. Upon encountering return instructions, TRUSS inserts instructions before these to retrieve the return address and its locations. It also retrieves the corresponding addresses stored in the `shadow_stack`, after which, it performs a comparison. The comparison

is to ensure that the return addresses and the locations match. Otherwise, an error signal is raised and the application is terminated.

For instance, if an adversary attempts to overwrite a function return address with the start address of a malicious code, he will have to inject such a code into the stack or the heap. When a subroutine is called, `shadow_stack` will store a copy of both the subroutine's return address and its corresponding location. During the subroutine's execution, the adversary by some means modifies the return address in the process stack. In a native program, such a scenario will cause the instruction pointer to be set to the modified return address in the process stack and this will result in the execution of the malicious code. However, with TRUSS in place, before a return instruction is executed, the return addresses in the process stack and the `shadow_stack` will be compared. In a case where the locations match and the return addresses do not match, an error will be signalled and the application will be terminated. In this way, TRUSS protects every function return address used in an application. This technique is similar to those discussed in other tools such as SmashGuard, StackShield and RAD. However, TRUSS differs from these tools because unlike the other tools which add additional checking instructions during compilation or through hardware modification, TRUSS inserts the checking instructions at runtime. This is clearly an advantage when protecting legacy applications without source code.

5.2 Base Pointer Defense

The base pointer (also known as frame pointer) is another piece of vulnerable information that also resides in the process stack. The main function of the frame pointer is to represent the start of each stack frame. In x86 architecture, whenever a

subroutine is invoked, a new stack frame is allocated in the process stack to the subroutine and when the subroutine completes, the process stack is freed. The frame pointer is saved in the process stack during every call instruction and is restored during each return instruction. This is illustrated in Figure 8. The frame pointer facilitates efficient access to current stack frame and keeps track previous stack frames. The value is maintained in a register for fast access. In addition, when applications are complied without the GCC option `-fomit-frame-pointer`, subroutine parameters and local variables are usually referenced relative to the frame pointer.

5.2.1 Base Pointer Modification Exploit

One technique to attack a system by compromising the frame pointer is via a dummy stack frame. As illustrated in Figure 8, when a function is invoked, the return address of the function is pushed into the stack. This is followed by saving the frame pointer value in the stack and then updating the frame pointer register with the stack pointer register. When an adversary attempts to modify the frame pointer, it will be rather easy for the adversary to access the frame pointer if parameters are passed to the function. In x86 machines, the frame pointer will be at an offset of 8 bytes with respect to the last parameter.

For instance, consider the vulnerable code fragment in Figure 9. The exploit illustrated in Section 5.1.1 modifies a return address to execute a sequence of malicious instructions. Although it will be easier to modify the return address directly, some protection techniques prevent direct modification of return addresses. Hence, using a dummy stack frame is a way to bypass those protections. The local buffer

(used in Figure 9) is overflowed up to and including the previous frame pointer. The data used for overflowing will be constructed in a manner resembling a stack frame. It will contain the start address of a sequence of malicious instructions followed by a memory location with an arbitrary stack address. The current frame pointer is overwritten with the address of the location of the arbitrary stack address. The purpose of such an action is for the frame pointer to be replaced with modified frame pointer when the subroutine completes and performs a return. Consequently, when another return instruction is executed, the address of the malicious code will replace the instruction pointer and in turn gets executed. Since none of the return addresses are altered directly, protection techniques that prevent return address modifications will not be able to detect such an attack.

5.2.2 Preventing Base Pointer Attack In TRUSS

The protection of the frame pointers is similar to the technique carried out for the return addresses. Following a call instruction, the frame pointer is saved in a buffer and before a return instruction, the frame pointer value is compared with the corresponding value in the stack. The storage of the frame pointer values is in fact done with the same `shadow_stack` (as mentioned in Section 5.1.2) for efficiency.

5.3 Global Offset Table Defense

Dynamic linking is used by applications to resolve shared symbols. In order to carry out dynamic linking, the dynamic linker primarily uses two processor-specific tables, the Global Offset Table (GOT) and the Procedure Linkage Table (PLT) as mentioned in Section 2.1. The dynamic linkers support position-independent code through the

GOT in each shared library. The GOT contains absolute addresses to all of the static data referenced in the program and it provides direct access to a shared symbol without compromising position-independence. Since the executable file and shared objects have separate GOTs, a symbol may appear in several tables. The dynamic linker processes all the GOT relocations before giving control to any code in the process image, thus ensuring that the absolute addresses are available during execution. The PLT is used to redirect function calls between the executable and a shared object or between different shared objects. It converts position-independent function calls to their absolute locations. The PLT contains many entries and allows procedure addresses to be resolved when they are called for the first time.

Suppose, there is a call to `printf()` in an application, this will correspond to a call to the PLT entry of `printf()` in the executable. This call will then make an indirect branch to the `printf()` entry in the GOT. If the GOT entry contains the absolute address of `printf()`, the instructions of `printf()` are executed. If the GOT entry has not been resolved, the dynamic linker is then invoked to resolve the absolute address of `printf()`.

```
PLT0:   pushl GOT + 4
        jmp *GOT + 8

PLTn:   jmp *GOT + m
        push #reloc_offset
        jmp PLT0
```

Figure 10: Procedure linkage table

The GOT stores pointers to all the global data that is addressed by the executable file. At load time, the dynamic linker stores two values at the memory locations *GOT + 4 and *GOT + 8. These two addresses, in fact, refer to the second and third

word in the GOT respectively. In the second word, the dynamic linker stores a code that identifies a particular library. In the third word, the dynamic linker stores the address of the symbol resolution routine.

As shown in Figure 10, the first entry in the PLT is PLT0. This is a routine to call the symbol resolution routine. In each of the other entries in the PLT, the instructions begin with an indirect jump to the GOT. Before any function is resolved, the target of the GOT entry refers to the next instruction in the PLT entry. When a function is called for the first time, the PLT routine is invoked and because the actual address of the function is yet to be resolved, the indirect jump executes the next instruction. The push instruction saves an offset into the process stack. This offset value is obtained from the executable's relocation table and it identifies both the symbol to be resolved and its corresponding GOT entry. The next jump instruction calls PLT0. Here the library identifier is pushed into the process stack and the dynamic linker's symbol resolution routine is invoked. Upon resolving the symbol, the linker stores the function's absolute address in the GOT entry. Hence, subsequent calls to the PLT entry will jump directly to the function itself without invoking the dynamic linker [33].

5.3.1 Global Offset Table Modification Exploit

The GOT entry is a point where a transfer in the program's execution flow occurs. This information is not protected in the executable and is vulnerable to attacks. One way to exploit the GOT entry is to overwrite the entry with the address of a sequence of malicious instructions. Thus, when the entry is utilized by the application, the

malicious code gets executed. In this section, I will explain how such an attack can be carried out. Consider the vulnerable code in Figure 11.

```
int main(int argc, char* argv[]) {
     int* ptr;
     char homedir[100];
     ...
     ptr = homedir;
     ...
     strcpy(ptr,argv[1]);
     ...
     strcpy(ptr,argv[2]);
     printf("Hello World!\n");
     ...
     return(1);
}
```

Figure 11: Vulnerable C function - 2

If the result from `argv[1]` is of a size that is larger than the local variable `homedir`, then, the first `strcpy()` will overwrite the memory locations adjacent to `homedir` including the integer pointer `ptr`. However, before a successful attack can be carried out, the adversary has to obtain some information from the system.

Firstly, the memory location of the GOT entry for `printf()` has to be determined. It is relatively easy to deduce this information from an unstripped binary. Suppose an executable is named `example1`. Then, one can utilize the `objdump` package in Linux to dump the dynamic relocations of the binary as follows:

```
objdump --dynamic-reloc ./example1 | grep printf
```

This command will output the desired address. Secondly, malicious code appended with `NOP` instructions has to be prepared. This piece of malicious code is usually referred to as shellcode. This data has to be constructed in a manner such that it overflows `homedir` and overwrites the memory location of `ptr` with GOT entry of `printf()`.

During the second strcpy(), an approximate start address of the shellcode has to be provided. This will, hence, be written to the memory location pointed to by ptr – in this case, the GOT entry of printf(). Therefore, when the program executes printf() after this strcpy(), the malicious code will be executed.

5.3.2 Preventing Global Offset Table Attack In TRUSS

TRUSS can protect applications that are vulnerable to the above-mentioned attack. It directs the dynamic linker to resolve all GOT entries during the initial start-up operations, after all the shared libraries have been loaded into the memory but before transferring control to the main program. This is achieved by declaring the environment variable LD_BIND_NOW. When DynamoRIO begins execution, it invokes dynamorio_init() hook function. In this function, the application's executable is examined and the addresses and sizes of the GOT and PLT are extracted. Following this, each entry of the GOT is stored into a separate buffer, GOT_BUF, with the corresponding GOT address. When DynamoRIO copies the application code into a basic block, the basic block is scanned for a call to a PLT entry and an indirect branch to a GOT entry following immediately. Upon detection, the indirect branch target address is used to retrieve the actual address of the function from GOT_BUF and a direct branch to the actual address replaces the indirect branch instruction. In this way, the application does not have to use the GOT during its execution and thus will bypass any GOT modification attack. An example involving a call to printf() is depicted in Figure 12. The basic block on the left illustrates the usual way of invoking a printf(). The basic block on the right is one which has been modified by TRUSS.

It has to be noted that GOT attacks may be detected via other techniques as well because it involves overflowing data buffers .

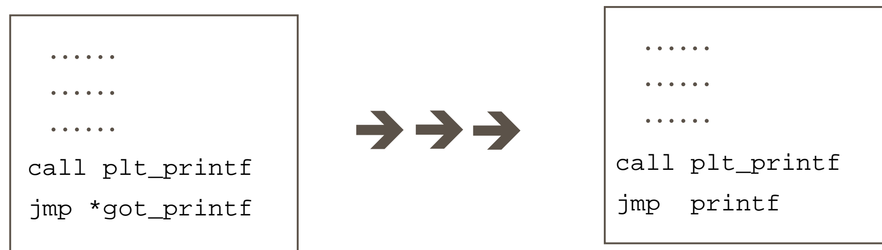

Figure 12: Copy of basic block in code cache

5.4 Format String Defense

The essence of the format string attacks utilizes the vulnerability in some standard C functions that use the format string argument. In C programming language, it is possible to declare the functions that take in a variable number of arguments. A format function is a special kind of C function taking in variable number of arguments, of which one is a format string. This function evaluates the format string argument by accessing the parameters that follow. These parameters, which could be of varying types, are converted into a suitable form and then, passed to the output stream. The signatures of widely used format functions are as follows:

- int printf(const char* format, ...)
- int fprintf(FILE* stream, const char* format, ...)
- int sprintf(char* str, const char* format, ...)
- int snprintf(char* str, size_t size, const char* format, ...)
- int vprintf(const char* format, va_list ap)
- int vfprintf(FILE* stream, const char* format, va_list ap)

- int vsprintf(char* str, const char* format, va_list ap)
- int vsnprintf(char* str, size_t size, const char* format, va_list ap)

The format string can contain additional information that determines the format of the output. These include flag characters, field width, precision and length modifiers.

The format string also carries information on the number of parameters that supposedly follow it. This information can be observed by counting the number of format specifiers present in the format string argument. Format functions become vulnerable due to the fact that when the format functions evaluate the format string, it has to access the parameters that follow. However, these functions do not ensure that the parameters are indeed sent by the caller function. Their evaluation is based on the assumption that the caller function has pushed the parameters into the process stack and that the evaluation routine is accessing a valid argument. This lack of validation on the memory locations accessed by format functions might result in a series of security breaches.

5.4.1 Reading The Stack

Many applications commonly use a user-supplied input as format string argument when format functions are invoked. However, when a call to a format function contains format specifiers without the corresponding argument, the function will continue to read values from the stack and print them to the output stream. The format function will not check if the argument accessed on the process stack is within a caller's stack frame. This makes the format function vulnerable because an adversary can supply a bogus format string argument as input. An example is illustrated in Figure 13.

```
printf("%08x.%08x.%08x.%08x.%08x.%08x\n");
```

Figure 13: Bogus format string

The `printf()` function in Figure 13 will not check whether any parameter is passed. Instead, it will assume that all memory accesses are valid and will read values from the stack. Hence, return addresses and other such critical values in the program memory can be displayed.

Moreover, it is also possible to use of the format string vulnerability to read values from any arbitrary memory location in the text segment, data segment or the heap of the process. When string arguments are passed to a subroutine, only a reference to the string is pushed into the stack. The actual string can be located in the heap, the text segment or the data segment. Suppose an adversary wants to read a string stored at the address (eg. 0xbfffff00), he has to start by finding the location of the format string in the process stack. Only a reference to the format string is pushed into the stack; so the adversary has to read down the stack to find the actual location of the format string. A number of '`%x`' specifiers can do this.

```
printf("AAAA_%x_%x_%x_%x_%x_%x_%x);
```

Figure 14: Detecting position of format string in x86 process stack

The `printf()` function shown in Figure 14 will output the content stored in the process stack. By adding sufficient '`%x`' to the format string, the function will

eventually output the format string. When the output starts with ‘41414141_...’[1], it is an indication that the format string argument has been reached. Now, the adversary can read the data at the address 0xbfffff00 by replacing AAAA with \x00\xff\xff\xbf (little endian format) and the last ‘%x’ specifier with ‘%s’ specifier in the format string as illustrated in Figure 15.

```
printf("\x00\xff\xff\xbf _%x_%x_%x_%x_%x_%x_%s);
```

Figure 15: Reading content at arbitrary memory location

5.4.2 Writing Into Arbitrary Memory Locations

The format string argument, moreover, gives the format function the privilege to write to memory locations with the use of ‘%n’ format specifier. This format specifier writes the number of bytes output by the format function to a memory location that is specified in the argument. Figure 16 illustrates this.

```
int i;
printf("12345%n", &i);
```

Figure 16: Proper use of ‘%n’ format specifier

The code fragment portrayed in Figure 16 will write the value 5 into the memory location referred to by the integer i. However, the ‘%n’ format specifiers can be abused to perform a write operation to any arbitrary location in memory. With such capability, an adversary can overwrite critical values like return addresses and function pointers to refer to a location of his choice. The use of the code segment

[1] 41 is ascii code for ‘A’

illustrated in Figure 16 can achieve such an attack. Say an adversary wants a memory address (eg. 0xbfffff00) to contain the value 0x44f660, he has to execute the code illustrated in Figure 14 to find the memory address of the format string argument. Subsequently, performing the code in Figure 17 will overwrite the desired memory location with the desired value.

```
printf("\x00\xff\xff\xbf_%.4519518x_%n")
```

Figure 17: Overwriting content of desired memory location

The two underscores will be output in addition to 0x44f65e symbols, which in total will write the integer 0x44f660 to the memory location 0xbfffff00. Thus, the '%n' specifier can result in serious security implications.

5.4.3 Format String Exploit

Format functions have the ability of reading and writing into arbitrary memory locations. This poses a threat to critical program information that resides in memory. Format functions that accept user-supplied input do not have the ability to discern between a normal input and a malicious input. The absence of such a check allows an adversary to capitalize on this vulnerability. An attack using format strings is described in Figure 18.

```
foo(){

      1.char text[200];
      2.FILE *input;

         ...
      3. input = stdin;
      4. fgets(text,sizeof(text),input);
      5. printf(text);
         ...
}
```

Figure 18: Vulnerable C function - 3

Consider the code fragment in Figure 18. Such a code fragment will usually be used to print out the user inputs. The user input is read-in via `stdin` and the input is copied to a local buffer before it is displayed on the screen. The problem with this code fragment is due to the `printf()` function. The correct use will be as follows:

```
printf("%s", text);
```

However, by forgoing the format specifiers, the program can be compromised by means of a format string attack.

When the program reaches line 3, it will wait for the user to provide it with some input. An adversary can take advantage of such a situation and provide a bogus format string to take control of the system. An example of a bogus format string that will work in this situation is as follows:

```
"%33\$344p%34\$n%33\$386p%35\$n%33\$277p%36\$n%33\$464p%37\$nA
A\x01\x00\x00\x00\xec\x98\x04\x08\xed\x98\x04\x08\xee\x98\x04\
x08\xef\x98\x04\x08"
```

The core idea of the attack is to save the shellcode in an environment variable, A (0xbfefda58), and overwrite the starting address of the DTOR deconstructor section – found within all GNU compiled binaries – with the address of A. In this case, the address of DTOR is 0x080498ec.The DTOR section contains a pointer to a function

that will be called when the program exits. This section allows write operations. Hence, when the program exits, the DTOR deconstructor will be invoked and thus the shellcode will get executed. The beginning of the bogus format string –

```
%33\$344p%34\$n%33\$386p%35\$n%33\$277p%36\$n%33\$464p%37\$n
```

– is derived from the address of environment variable, A. The address of A is firstly converted into little endian format. A saturating counter approach is then employed to convert each byte of the address to a value such that `printf()` outputs the correct number of bytes to leave the lower byte equal to the memory address of A. The saturating counter approach is illustrated in Figure 19. The specifier, %33\$344p, will print out the 33rd argument in `printf()`'s stack frame with padding that is of size 344 bytes and the specifier, %34\$n, will write that number to the location of `printf()`'s 34th argument.

```
mem1p =  mem1 + 256
mem2p = ((256 - mem1) + mem2)%256 + 256)
mem3p = ((256 - mem2) + mem3)%256 + 256)
mem4p = ((256 - mem3) + mem4)%256 + 256)
```

Figure 19: Saturating counter approach

Following the pairs of '%p' and '%n', the format string contains some padding. The padding is necessary to ensure that the following values and addresses get copied to the 33rd and subsequent arguments. The padding used here is the character 'A'. The value 0x00000001 follows the padding. This value is included to represent a dummy argument. This will be the 33rd argument. Following this value, the 4-byte addresses of the DTOR deconstructor section is appended. It is necessary to include all 4 byte-addresses because the '%n' specifier writes to 1 byte memory. Consequently, when the bogus format string is passed as input, the address of the DTOR deconstructor section

gets overwritten and when the program exits, the shellcode stored in the environment variable A gets executed [43].

5.4.4 Preventing Format String Attack In TRUSS

In order to monitor an application, TRUSS passes control to the user via the `dynamorio_init()` function. This function is part of the client program that has to be loaded before DynamoRIO begins execution. The `dynamorio_init()` function allows the user to set up the environment to facilitate the monitoring of the application. In TRUSS, the `dynamorio_init()` function resolves and stores the addresses of the format functions even before they are invoked in the application. These addresses are subsequently used to intercept every invocation of the functions. The interception is done at the basic block level. The interception details are as discussed in Section 5.1. The `dynamorio_basic_block()` function allows the manipulation of a newly created basic block before it is executed. Hence, every instruction in the block is scanned for calls to format functions. Each time a call to a format function is identified, a call to a hook function is placed before the original call. In this way, the hook function can check the arguments of the format function before the format function uses them.

In addition, a parallel stack is maintained to keep track of the previous frame pointer values when a function is invoked. This list is updated dynamically at every call and return instruction. A stack pointer address is pushed into the parallel stack during a call and an entry is popped from the stack before a return instruction. For efficiency, information in the `shadow_stack` is used.

During the execution of the application, the hook functions are called before the format functions. The core idea in the hook functions is to prevent reading from or writing to any memory location including and beyond the current stack frame pointer. The hook functions access the arguments (including the format string argument) of the format functions from the process stack. The format string argument is parsed to observe the number of format specifiers present in it. This number must not be more than the total memory space between the address that stores the format string in the process stack and the stack frame pointer. This policy ensures that the safety of the critical information such as the frame pointers and return addresses are not compromised.

The above-mentioned technique will work for `printf()`, `vprintf()`, `fprintf()` and `vfprintf()` functions. The `sprintf()`, `vsprintf()`, `snprintf()` and `vsnprintf()` functions, however, do more than just to access arguments in the stack to evaluate the format string. These functions store the evaluated output in a buffer. If the output data is larger than the buffer's size, the buffer will be overflowed.

In order to prevent buffer overflow due to `sprintf()`, `vsprintf()`, `snprintf()` and `vsnprintf()` functions, a separate parser is used to evaluate the format string argument. This parser scans the format string for format specifiers and accumulates the total size of the parameters accessed. Integer and double arguments will respectively add four and eight to the accumulator. String arguments, however, will continuously increase the accumulator by one until a '`/0`' escape character is encountered. In addition, the precision and the field width options in the format string have to be taken into account. While the precision option defines the maximum number of characters to be printed to the output, the field width defines the minimum

number of characters. This value has to be added to the accumulator, which will then be checked against the memory space between the memory location (where the buffer resides) and the stack frame pointer. This technique ensures that calls to `sprintf()`, `vsprintf()`, `snprintf()` and `vsnprintf()` functions are safely executed.

5.5 Vulnerable C Library Functions Defense

In the standard C library, there exist a few functions (other than format functions) that are vulnerable to buffer overflow as well. These functions also handle buffers in an insecure manner and they do not have any mechanism to prevent buffer overflow. The signatures of the commonly used functions that face this problem are as follows:

- char *strncat(char *, const char *, size_t)
- char *strcat(char *, const char *)
- char *stpcpy(char *dest, const char *src)
- char *strncpy(char *, const char *, size_t)
- char *strcpy(char *, const char *)
- void *memcpy(void *s1, const void *s2, size_t n)
- wchar_t *wcscpy(wchar_t *, const wchar_t *)
- wchar_t *wcscat(wchar_t *, const wchar_t *)
- wchar_t *wcpcpy(wchar_t *dest, const wchar_t *src)
- int *_IO_vfscanf(_IO_FILE *s, const char *f, _IO_va_list argptr, int *errp)

In Section 5.1.1, I discussed a return address exploit. Although, the target of the exploit was the return address, it was through the vulnerability of the

strcpy() function that such an exploit was possible. Hence, it is necessary to provide some form of security mechanism to such standard C functions.

5.5.1 Preventing Vulnerable C Library Functions Attack In TRUSS

The security mechanism for the list of functions basically employs a technique similar to the one used for the security of sprintf(), vsprintf(), snprintf() and vsnprintf(). However, unlike those functions, this group of functions does not have to handle any format string argument.

Functions like strcpy(), stpcpy(), wcscpy() and wcpcpy() are invoked with two buffers. These functions aim to copy the content of a source buffer to a destination buffer. To prevent any buffer overflow, the content of the input buffer is measured using the strlen() function. This size will be considered against the memory space between the memory location where the destination buffer resides and the stack frame pointer. If the size of the source buffer is larger than the space allocated to the destination buffer, then an error message is signalled and the application will be terminated.

Functions such as strncpy() and memcpy() perform the same work as strcpy(), stpcpy(), wcscpy() and wcpcpy(). But these functions accept an additional argument that specifies the size of the content that has to be copied from the source buffer to the destination buffer. Hence, here, the size comparison (as mentioned earlier) is used again.

strcat() and wcscat() also copies the content of the source buffer to the destination buffer. However, for these functions, the content of the source buffer is

appended to the destination buffer. Hence, to prevent any overflow, the size of the destination buffer and the size of the source buffer must be less than the permissible size. `strncat()` accepts an additional argument that specifies the size of the source buffer to be copied onto the destination buffer. As before, the size of the destination buffer and the size passed by the application must be less than the permissible size.

Unlike the functions discussed up to this point, the `scanf()` family of functions works in a different manner. It accepts the input from the user and then, writes the received input to the corresponding arguments (that are passed as parameters). These functions cannot be intercepted before the routine's execution because the safety of these functions depends on the user inputs. Furthermore, it can be observed that the `scanf()` family of functions invokes the `_IO_vfscanf()` subroutine to perform its task. Thus, in order to check the functions for buffer overflows, these are intercepted upon their return from the subroutine. Although by this time, the buffer might have already overflowed, the effect can only be felt when one of the `scanf()` family of functions returns to the caller.

Before the subroutine `_IO_vfscanf()` is invoked, the process stack content, referred by the addresses stored in the parallel stack, is saved. Upon returning from the subroutine, the content of the process stack is compared with the saved values. Any mismatch would signal an error and terminate the application. This technique will effectively prevent any form of stack smashing attacks. TRUSS is therefore able to protect applications from being violated via unsafe standard C library calls.

5.5.2 Vulnerable C Library Functions BSS/DATA/HEAP Overflow Exploit

The techniques in Section 5.5.1 are effective only if the buffer, handled by the unsafe functions, is allocated in the stack. The defense mechanism basically prevents any data beyond the frame pointer to be modified. In addition to this, the buffers used in the list of functions can be global buffers or buffers allocated in the heap. In this section, I will describe one such instance - a heap overflow exploit.

The GNU standard C library employs a memory allocator, dlmalloc, implemented by Doug Lea [32]. This memory allocator handles applications' dynamic memory requests and freed memory. dlmalloc's memory management is based on chunks – memory blocks that consist of usable regions and additional information. The structure of such a chunk is shown in Figure 20.

```
    struct malloc_chunk {
            INTERNAL_SIZE_T prev_size;
            INTERNAL_SIZE_T size;
            struct malloc_chunk *bk;
            struct malloc_chunk *fd;
     };
```

Figure 20: Heap memory chunks

The additional information is stored at the beginning of each chunk and it stores the sizes of the current and the previous chunks. This information facilitates coalescing of two sequential and unallocated chunks into one larger chunk. Moreover, all chunks contain two pointers. These pointers are utilized when the chunks are unallocated.

Memory chunks that are unallocated in the application are maintained in bins according to their sizes. Bins that store chunks of sizes smaller than 512 bytes hold chunks of exactly one size. Bins that store chunks larger than 512 bytes hold chunks of a size range that increase logarithmically. When a process requests for dynamic memory, the search for the suitable chunk starts at the appropriate bin depending on

the memory size requested and then, it is processed in smallest-first, best-fit order. Within each bin the chunks are maintained as a doubly linked list known as free-list. Two pointers – forward (fd) and backward (bk) – are used to traverse this list in both directions.

However, the use of the fd and bk pointers to link available chunks in bins exposes libc's memory management routines to security vulnerability. If an adversary is able to overflow a dynamically allocated block of memory, he can effectively overwrite the next contiguous chunk header in memory. When the overflowed chunk is freed and stored in a bin's double-linked list, the adversary can control the values of that chunk's fd and bk pointers. Consider the macro in Figure 21.

```
#define unlink(P, BK, FD) { \
        [1] FD = P->fd;  \
        [2] BK = P->bk;  \
        [3] FD->bk = BK; \
        [4] BK->fd = FD; \
  }
```

Figure 21: Memory management macro

This macro is used to remove a chunk from the free list. P is the chunk to be removed. In order to carry out an attack, an adversary has to store the address of a function pointer in P→fd and the desired value in P→bk. The function pointer is with a minus 12 bytes so that FD→bk points to the function pointer. This macro will cause the function pointer to point to the desired value. When the function pointer is invoked in the program, the code at the desired location gets executed [50]. An analogous attack is illustrated in Figure 22.

```
main(int argc, char* argv[]) {
        1. char *buf0 = (char*)malloc(16);
        2. char *buf1 = (char*)malloc(16);
        3. char *buf2;

        4. ...
        5. free(buf1);
        6. ...
        7. strcpy(buf0, argv[1]);
        8. ...
        9. buf2 = (char*)malloc(16);
        10. printf("\bin\sh");
 }
```

Figure 22: Vulnerable C program - 4

In Figure 22, lines 1 and 2 allocate 2 buffers in the heap. These buffers are of the same size and hence, they will be allocated in contiguous memory blocks. In line 5, when the buffer, `buf1`, is unallocated, it is placed back in its free bin. The `strcpy()` function in line 7 can overflow `buf0` and overwrite the header information of chunk `buf1`. For instance, if an adversary provides an input as follows –

```
"AAAAAAAAAAAAAAAA\x00\x00\x00\x00\x00\x00
\x00\x10\x64\x96\x04\0x08\xb0\x65\x69\x00"
```

– the sequence of 'A' will overflow the `buf0`. The following values are 0 and 16 in 4 byte hexadecimal format. These values will overwrite the previous and current chunk sizes respectively. Following these sizes is the GOT location corresponding to `printf()`. This address is subtracted by 12 bytes. The next address is the absolute address of `system()`. Hence, the `buf1`'s fd and bk pointers are overwritten with these addresses respectively. When line 9 is executed, the macro in Figure 21 is executed before the overflowed chunk is allocated. This macro will, therefore, copy the address of `system()` to the GOT location of `printf()`. This will, in turn, execute a `system("\bin\sh")` instead of `printf("\bin\sh")` and spawn a shell with the privilege of the vulnerable program.

5.5.3 Preventing BSS/DATA/HEAP Overflow Attack In TRUSS

In Linux, initialized data buffers are allocated in the .DATA segment and the uninitialized data buffers are allocated in the .BSS segments. These segments are present in the ELF file. The symbol table present in the ELF file holds an array of symbols used in the executable. For each entry, the name, size, and section numbers are stored. The data structure of the ELF symbol table is shown in Figure 23.

```
typedef struct {
   Elf32_Word    st_name; /* Symbol name */
   Elf32_Addr    st_value;/* Symbol value */
   Elf32_Xword   st_size; /* Size of object (e.g., common) */
   unsigned char st_info; /* Type and Binding attributes */
   unsigned char st_other;/* Reserved */
   Elf32_Half    st_shndx;/* Section table index */
} Elf32_Sym;
```

Figure 23: Symbol structure

In order to ensure that the vulnerable C functions do not overflow BSS/DATA/HEAP buffers, TRUSS maintains a list of global variables with its corresponding sizes. When dynamorio_init() is invoked by DynamRIO, the binary file is analyzed to retrieve the .DATA and .BSS sections. The limits for these sections and the section numbers are then stored. The symbol table is then scanned through for symbols belonging to the .DATA and .BSS sections. The symbols' addresses and the respective sizes are saved in a global buffer in DynamRIO. Thus, when a buffer from either section is utilized by the vulnerable functions, the size of the buffer can be obtained from the global buffer and this aids in ensuring that the limits of these buffers are not overflowed.

Buffers allocated in the heap can also be used by the vulnerable functions. In order to prevent the limits of these buffers from being violated, the sizes of these buffers have to be stored. It is safe to assume that buffers in the heap are allocated via calls to malloc(), calloc(), realloc() and the memory is released via call to free().

Hence, every call to malloc(), calloc(), realloc() and free() are intercepted using the technique that was used to intercept the format functions. The requested sizes are stored in a buffer and upon completion of the malloc(), calloc() and realloc() functions, the start address of the memory space allocated is stored in the same buffer. During a free() call, the buffer is removed from the list. In this way, I maintain a list of buffer addresses and sizes. This list can be used to verify whether any buffer in the heap is overflowed. In C++ program, the new operator can also allocate heap buffers. In addition, functions such as brk(), sbrk() and mmap() can be used to allocate memory in the heap. Hence, in TRUSS, I restrict the defense to heap buffers allocated via malloc(), calloc() and realloc().

This technique is only implemented in Linux and protection of buffers in the .DATA and .BSS sections are only applicable if the executable is not stripped. A stripped executable will not contain the symbol table. Thus, it is impossible to retrieve the information regarding the global buffers. In Microsoft PE files, the symbol information is present in the COFF file, .obj, and is not found in the executable, .exe. Thus, it will be difficult to monitor these buffers in Windows.

5.6 Longjmp Buffer Defense

Another avenue through which control transfer information can be altered is via setjmp() and longjmp() functions. These are standard C functions that typically provide a crude form of exception handling. setjmp() function stores the context information for the current stack frame into a jmp_buf buffer. By using longjmp(), a program can jump out of many levels of nested function calls. The data structure of the jmp_buf buffer is shown in Figure 24.

```
#define _JBLEN  9
typedef struct { int _jb[_JBLEN + 1]; } jmp_buf[1];

jmb_buf[] = {ebx, esi, edi, ebp,
             esp, eip, return_val}
```

Figure 24: Setjmp/Longjmp buffer

The context information of an application depends completely on the contents of its memory and the contents of its registers. The content of the registers includes the stack pointer (esp), frame pointer (ebp), and program counter (eip). `setjmp()` essentially saves the contents of the registers. When a `longjmp()` function is invoked and a `jmp_buf` buffer is passed as a parameter, this function restores the context of the application to the state of the program when `setjmp()` was called. An example of how a pair of `setjmp()` and `longjmp()` works is depicted in Figure 25.

```
1.  /* Global "environment" variable; this must be in scope if longjmp is to be called. */
2.  char buf [16];
3.  jmp_buf g_env;


4.  /* A function that exits abnormally via longjmp */
5.  void function(void){
6.      printf("(calling longjmp)\n");
7.      longjmp(g_env,1);
8.      printf("This is not reachable because of the longjmp above.\n");
9.  }

10. int main(int argc, char *argv[])
11. {
12.     int i = setjmp(g_env);
13.     /* This is where execution resumes when longjmp is called: */
14.     ...
15.     strcpy(buf, argv[1]);
16.     ...
17.     /* setjmp returns 0 only when it is first called; never after a longjmp */
18.     if( i = = 0 ){
19.         printf("(calling function)\n");
20.         function();
21.         printf("This is never reached!\n");
22.     }
23.     return 0;
24. }
```

Figure 25: Setjmp/Longjmp example

In the code fragment in Figure 25, when `setjmp()` function is invoked in line 12, the context information is saved in `g_env`. The `setjmp()` instruction returns a 0 when it is called the first time. Subsequent calls to `setjmp()` after a `longjmp()` will return a non-zero value (which depends on the value passed to `longjmp()`). The execution follows from line 12 until line 20 where `function()` is invoked. The execution continues until the `longjmp()` invocation. Here, the program context information, which was stored in line 12, will be restored and line 12 will be executed again. Now, `setjmp()`will return a non-zero value (1 in this case) and thus the program terminates after line 23.

5.6.1 Longjmp Buffer Modification Exploit

It is essential to note that the `setjmp()` function stores the program counter and the frame pointer in the `jmp_buf` buffer. Thus, the integrity of the contents in `jmp_buf` buffer is vital because these are the information used during a `longjmp()`.

In Figure 25, the code fragment contains a `strcpy()` function that accepts a user input (in line 15). This is the program's vulnerable point. The technique used to attack such a program is to save a shellcode in an environment variable, A, and overwrite the value of the program counter in the `jmp_buf` buffer with the address of A. In order to overwrite the program counter in the `jmp_buf` buffer, sufficient padding followed by the address of the environment variable, A, has to be provided at line 15 by the adversary. The padding is to fill up `buf` and the other members of the `jmp_buf` buffer until the program counter. Following this, the program counter gets overwritten with the value of A. Hence, when a `longjmp()` function is invoked at line 7, the shellcode gets executed.

5.6.2 Preventing Longjmp Buffer Modification Attack In TRUSS

An adversary can perform a return address modification by altering the program counter information or perform a frame pointer modification by altering the frame pointer value in the `jmp_buf` buffer. In order to prevent such attacks, TRUSS stores the `jmp_buf` buffers in a buffer named `setjmp_buf` in DynamoRIO. When the application executes a `setjmp()` instruction, the address `jmp_buf` buffer is recorded. Upon completion of the `setjmp()` function, the content of the `jmp_buf` buffer at the recorded address is stored in `setjmp_buf`.

When a `longjmp()` function is invoked, this function is intercepted using the technique mentioned in Section 5.4. The address of the parameter (`jmp_buf` buffer) is retrieved from the stack and the current content of the `jmp_buf` buffer is compared with the saved contents. Any alteration to content will signal an error and the application will be terminated.

5.7 Function Pointer Defense

A function pointer is a type of pointer in the C and C++ programming languages. It points to a function. Function pointers are used to eliminate giant switch statements and in addition, they allow a programmer to dynamically modify a function to be called. Memory space for the function pointers can be allocated in stack or heap. If an adversary is able to identify a function pointer in a program, then he can modify the address referenced by the function pointer to some malicious code. The use of function pointers can be identified in the assembly code by a special characteristic in applications compiled with GCC. When a function is invoked via a function pointer, the address referenced by the function pointer is loaded to a general register followed

by a call instruction whose target address is the one in the register. An example is shown in Figure 26. Overflowing adjacent buffers and modifying the address referred to by the pointer can alter function pointers.

```
mov  0xbffffff0, %eax //0xbffffff0 is the target address
call %eax
```

Figure 26: Function pointer assembly code

5.7.1 Function Pointer Modification Exploit

Function pointers are used to store control transfer information and thus its credibility is vital during a program execution. However, when function pointers are declared adjacent to any buffers, it will be possible to overwrite the location referenced by the function pointer via overflowing the adjacent buffer. Figure 27 illustrates such an example.

```
main(int argc, char* argv[])
{
        1. static char buf [16];
        2. //declare function pointer
        3. static void (*funptr) (void);

        4. ...
        5. funptr = ( void (*) (void)) goodfunction;
        6. ...
        7. strcpy(buf, argv[1]);
        8. ...
        9. (void) (*funptr) (void);
        10. ...
}

void goodfunction()
{
    ...
}
```

Figure 27: Vulnerable C program - 5

In the code fragment depicted in Figure 27, a function pointer is declared adjacent to a buffer. Similar to the exploit mentioned in Section 5.6.1, a shellcode can be stored in an environment variable, A. In order to carry out the attack, a user-input consisting of 16 character ‘A’s followed by the address of A (little endian format) has to be provided. This will overwrite the address referenced by the function pointer. Subsequently, when the function pointer is used in the application, the shellcode will get executed.

5.7.2 Preventing Function Pointer Modification Attack In TRUSS

To prevent function pointer modifications, when there is an indirect call instruction that uses a register, the checking function is inserted prior to the call. The checking function retrieves the address in the register and ensures that this does not lie within any data region. In this way, the malicious code injected into data buffers will not be executed. This technique prevents any execution of data. Currently, this technique is only implemented for Linux.

Chapter 6

Security Evaluations

This chapter discusses the security tests that have been used to evaluate TRUSS.

6.1 Security Evaluation With John Wilander's Testbed Of Twenty Buffer Overflow Attacks

TRUSS's ability to prevent buffer overflow attacks has been tested using John Wilander's testbed of twenty buffer overflow attacks [56]. This testbed of attacks works on both Linux and Windows. Attacks illustrated in the testbed will either overflow the buffer all the way to the attack target or overflow a buffer to redirect a pointer to the target. The attacks are targeted at the stack, heap, `.BSS` and `.DATA` sections. The main targets in the testbeds are the return addresses, old frame pointers, function pointers and function parameters. The evaluation of TRUSS is shown in Table 1 and 2.

No	ATTACKS	TRUSS
1	Buffer overflow on stack all the way to parameter function pointer	DETECTED
2	Buffer overflow on stack all the way to parameter longjmp buffer	DETECTED
3	Buffer overflow on stack all the way to return address	DETECTED
4	Buffer overflow on stack all the way to old base pointer	DETECTED
5	Buffer overflow on stack all the way to function pointer	DETECTED
6	Buffer overflow on stack all the way to longjmp buffer	DETECTED
7	Buffer overflow on heap/BSS all the way to function pointer	DETECTED
8	Buffer overflow on heap/BSS all the way to longjmp buffer	DETECTED
9	Buffer overflow of pointer on stack and point to parameter function pointer	DETECTED
10	Buffer overflow of pointer on stack and point to parameter longjmp buffer	DETECTED
11	Buffer overflow of pointer on stack and point to return address	DETECTED
12	Buffer overflow of pointer on stack and point to old base pointer	DETECTED
13	Buffer overflow of pointer on stack and point to function pointer	DETECTED
14	Buffer overflow of pointer on stack and point to longjmp buffer	DETECTED
15	Buffer overflow of pointer on heap/BSS and parameter function pointer	DETECTED
16	Buffer overflow of pointer on heap/BSS and point to parameter longjmp buffer	DETECTED
17	Buffer overflow of pointer on heap/BSS and point to return address	DETECTED
18	Buffer overflow of pointer on heap/BSS and point to old base pointer	DETECTED
19	Buffer overflow of pointer on heap/BSS and point to function pointer	DETECTED
20	Buffer overflow of pointer on heap/BSS and point to longjmp buffer	DETECTED

Table 1: Security performance on John Wilander's testbed of twenty buffer overflow attacks in Linux

No	ATTACKS	TRUSS
1	Buffer overflow on stack all the way to parameter function pointer	DETECTED
2	Buffer overflow on stack all the way to parameter longjmp buffer	DETECTED
3	Buffer overflow on stack all the way to return address	DETECTED
4	Buffer overflow on stack all the way to function pointer	UNDETECTED
5	Buffer overflow of pointer on stack and point to parameter function pointer	UNDETECTED
6	Buffer overflow of pointer on stack and point to parameter longjmp buffer	DETECTED
7	Buffer overflow of pointer on stack and point to return address	DETECTED
8	Buffer overflow of pointer on stack and point to function pointer	UNDETECTED
9	Buffer overflow of pointer on stack and point to longjmp buffer	DETECTED
10	return-to-libc system("echo Attack Successful")	DETECTED

Table 2: Security performance on John Wilander's testbed of buffer overflow attacks in Windows

The undetected cases occur because function pointer monitoring is not included in TRUSS for Windows.

6.2 Security Evaluation With BASS

Moreover, I have also used the Benchmark Suite for Evaluating Architectural Security Systems, BASS [43] to test TRUSS's ability to prevent buffer overflow attacks. The

attacks in this suite target the stack, heap, BSS and dtor sections. The main targets in this suite are the return addresses, old frame pointers, function pointers and function parameters. This test suite is implemented only for Linux. The result of TRUSS's performance on these benchmarks is summarized in Table 3.

No.	Benchmark Program	Vulnerability	Attack / End Result	Location	TRUSS
1.	Lottery	Buffer overflow	overwrite function pointer /manipulate instruction flow	BSS	DETECTED
2.	Lottery	Buffer overflow	overflow variable / modify bank account total	BSS	DETECTED
3.	Message_wall	Buffer overflow	overflow file pointer / add malicious root account	Heap	DETECTED
4.	Small_finger	Buffer overflow	overflow return address / spawn root shell	Stack	DETECTED
5.	Secure_log	Format string	read memory location / access cryptographic key	Data	DETECTED
6.	Secure_log	Format string	write memory location / modify cryptographic key	Data	DETECTED
7.	Secure_log	Format string	overwrite deconstructor / spawn root shell	Dtors	DETECTED

Table 3: Security performance on BASS in Linux

As it can be seen from Table 3, TRUSS is able to successfully detect and prevent all the attacks in BASS.

6.3 Security Evaluation With Libsafe Exploits

In addition, TRUSS in Linux was tested with the exploit code provided by Libsafe in its distribution package. The test cases were included in the Libsafe package to verify the functioning of Libsafe. Table 4 shows how TRUSS and Libsafe perform against the test cases. It can be observed that TRUSS successfully detects all the attacks. The surprising result is that Libsafe is unable to detect its own exploit code.

Further inspection of Libsafe showed that the application is based on the assumption that every global C function will be dynamically resolved via the PLT. However, this is not the case at all times. This oversight can be observed during the

execution of `canary-exploit`. Libsafe protects the `fprintf()` function by intercepting the `vfprintf()` function. `vprintf()` is invoked by `fprintf()`. In newer Linux versions, the `fprintf()` functions make direct internal call to `vfprintf()` without calling the PLT entry. Hence, Libsafe's version of safe `vfprintf()` is never invoked. This is why the `canary-exploit` is successful on Libsafe. `t6` succeeds because of a similar reason. TRUSS does not face this problem because it intercepts the function `fprintf()` only after the first basic block for the function is built by DynamoRIO.

No	**Attack**	**TRUSS**	**Libsafe**
1	canary-exploit	DETECTED	UNDETECTED
2	exploit-non-exec-stack	DETECTED	DETECTED
3	t1	DETECTED	DETECTED
4	t1w	DETECTED	DETECTED
5	t3	DETECTED	DETECTED
6	t3w	DETECTED	DETECTED
7	t4	DETECTED	DETECTED
8	t4w	DETECTED	DETECTED
9	t5	DETECTED	DETECTED
10	t6	DETECTED	UNDETECTED

Table 4: Security performance on Libsafe exploit code in Linux

6.4 Security Evaluation With cOntext's GOT Attack

The GOT defense in TRUSS has been tested with the exploit code depicted in [16]. This code attacks an application that contains

printf("Array …");

The exploit creates an executable named `Array`, which contains one instruction:

system("/bin/sh");

This executable will invoke a new shell. The exploit code subsequently overwrites the GOT entry of `printf()` with the absolute address of `system()`. Hence, when the

application is executed, the `printf()` statement will execute the `Array` executable and a new shell is invoked with the privilege of the application.

When the application is executed with TRUSS's defense in place, the attack does not succeed because the modified entry in the GOT is never used. Thus, TRUSS is able to thwart any attack that uses GOT modification to execute malicious code.

6.5 TRUSS vs Four Different Tricks To Bypass StackShield And StackGuard Protection

StackShield and StackGuard, as reviewed earlier, protect applications against stack smashing attacks. However, in [48], four techniques that can bypass the protection provided (StackShield and StackGuard) have been described. I will analyze these techniques against TRUSS in this section.

- Technique 1

In standard compiled C code, functions' arguments are pushed into the stack before the return address as shown in Figure 8. When a stack based buffer overflows, an adversary may be able to control the function's arguments and this can turn a protected program into a vulnerable program. If StackGuard is used, the canary death handler will be called and several library functions such as `openlog()` or `_exit()` will be invoked. Overwriting these functions' GOT entries will allow the adversary to hook the execution flow.

TRUSS's solution

In TRUSS, the GOT technique as described in Section 5.3 will bypass any modification made to the GOT entries and hence will foil Technique 1.

- Technique 2

This technique is one version of frame pointer attack. Upon a return instruction, the frame pointer gets modified and before the second return, control over the stack pointer is gained. Hence, the adversary can control the location where the function returns.

TRUSS's solution

This technique attempts to modify both frame pointers and return addresses. However, TRUSS can successfully detect any alteration of frame pointers and return addresses and thus will thwart this kind of attacks.

- Technique 3

In standard C code, compiled without the GNU-equivalent of the `-fomit-frame-pointer` option, all local variables are accessed relative to the frame pointer. Thus, if an adversary has control over the frame pointer, he will be able to manipulate the caller's local variables and arguments. In this attack, an 'off-by-one' overflow, where the least significant byte of the saved frame pointer is altered to 0, is used. It then goes on to overwrite the GOT entries.

TRUSS's solution

TRUSS can detect any alteration of frame pointer attacks and GOT entries. Therefore, it will prevent any attacks that utilize Technique 3.

- Technique 4

This technique extends Technique 3 and performs a few different kinds of attacks. Firstly, it exploits a `printf()` to show the memory content. It can make a pointer refer to some critical data such as the environment variables or modify the content of some variables. This is typically a format string attack. This technique can also modify GOT entries.

TRUSS's solution

The format string protection provided by TRUSS will not allow the adversary to access any memory content beyond the stack frames. Although the local variables will be vulnerable to alterations, this is unlikely to have any serious effect. In addition, the GOT protection will bypass any GOT entry modification.

Chapter 7

Experiments

This chapter discusses the performance tests that have been used to evaluate TRUSS.

7.1 Experimental Setup

All experiments were executed on a Dell Optiplex GX280 Pentium 4 530 running at 3.0 GHz with 1 GB RAM. The operating systems used are Microsoft Windows XP Professional SP2 and Linux Fedora Core 3.

7.2 Performance Test With DynamoRIO's Profiling

Firstly, profiling was carried out to measure the overhead caused by the additional protection code. Profiling facilities provided by DynamoRIO in Linux showed that more than 95% of the time, the application code and the protection code are being executed. The execution of SPEC CINT2000 [53] programs `crafty`, `twolf` and `parser` with profiling showed that protection code was present in 14%, 9% and 17% of the profiling samples collected respectively. This implies that TRUSS does incur moderate overheads.

7.3 Performance Test With SPEC CINT2000 Benchmark Programs

SPEC CINT2000 programs were used on both Windows and Linux to examine the performance of TRUSS. In order to evaluate the performance, I collected three sets of

execution times in each operating system. First, the benchmark programs were executed natively with no modifications. These results serve as a baseline to measure TRUSS's performance. Then, the benchmarks were executed on DynamoRIO without any client programs. This will measure the overhead due to DynamoRIO's instrumentations. Lastly, TRUSS is loaded and its performance is recorded. The results of the tests are shown in Figure 28 and Figure 29. The overheads shown in the results include the time for analysis of the binaries, insertion of the protection code and execution of the protection code.

BenchMark	Native	DynamoRIO	TRUSS	%overhead
Gzip	180.336	199.129	211.742	17.415
Vpr	207.795	218.077	241.471	16.206
Gcc	80.929	208.766	287.211	254.893
Mcf	192.308	193.163	193.782	0.766
Crafty	118.144	169.434	209.785	77.567
Parser	218.046	268.165	326.996	49.967
Eon	212.979	261.007	369.775	73.620
Perlbmk	164.114	255.084	334.879	104.053
Gap	91.037	122.692	153.889	69.040
Vortex	159.618	317.307	363.284	127.596
Bzip	183.703	197.595	227.042	23.592
Twolf	329.834	372.113	411.403	24.730
AVERAGE	178.237	231.878	277.605	55.750

Table 5: Performance of SPEC CINT2000 benchmark programs on Linux

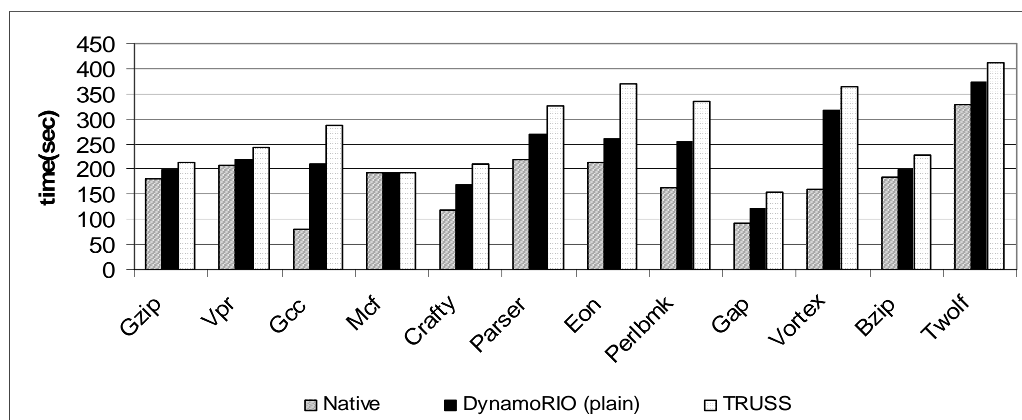

Figure 28: Performance of SPEC CINT2000 benchmark programs on Linux

BenchMark	Native	DynamoRIO	TRUSS	%overhead
Gzip	183.718	193.516	295.39	60.784
Vpr	192.515	197.538	365.765	89.993
Gcc	84.077	179.104	308.781	267.26
Mcf	201.171	202.514	324.343	61.228
Crafty	125.453	189.406	410.156	226.94
Parser	221.812	258.468	738.015	232.721
Eon	141.952	168.937	474.5	234.268
Perlbmk	148.283	271.843	569.125	283.81
Gap	102.093	125.687	370.265	262.674
Vortex	141.218	217.64	602.86	326.9
Bzip	226.64	240.596	415.046	83.13
Twolf	332.187	362.622	520.781	56.773
AVERAGE	175.093	217.323	449.586	156.769

Table 6: Performance of SPEC CINT2000 benchmark programs on Windows

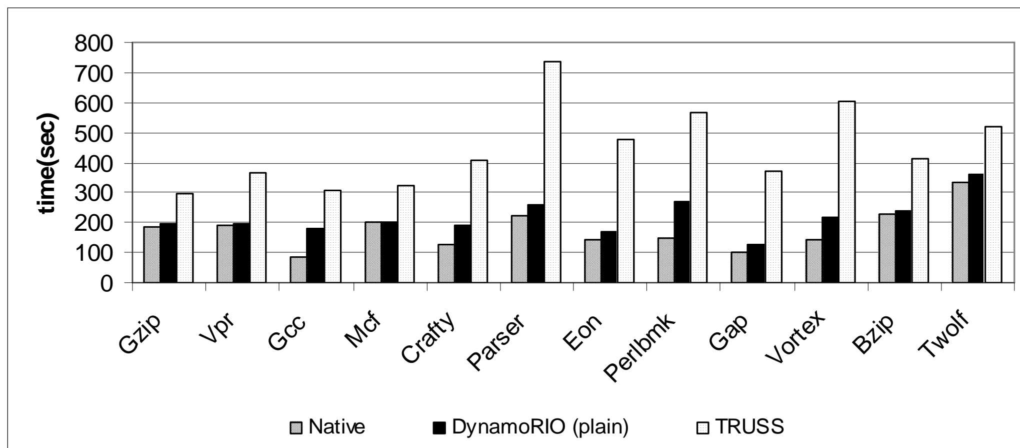

Figure 29: Performance of SPEC CINT2000 benchmark programs on Windows

In Windows, TRUSS incurs an average overhead of 156% in execution time. In Linux, TRUSS incurs an average overhead of 55% in execution time. One reason for such a significant variation in Linux and Windows is that in Linux the eflags were not saved and restored at every check. The eflags essentially contain the zero, carry, sign, parity, adjust, trap, interrupt, direction and overflow flags. In Linux the applications can execute correctly without saving and restoring the eflags. However, in Windows, few applications produced incorrect results when the eflags were ignored. Hence, in Windows the eflags were saved and restored at every check. The SPEC CINT2000 benchmarks `gcc`, `vortex` and `perlbmk` exhibited significant overheads in both operating systems. But it has to be noted that `gcc`, `vortex` and `perlbmk` perform

badly on DynamoRIO without TRUSS. These are programs which contribute to the exceptionally high overhead in Windows.

7.4 Performance Test With Bapco Sysmark Benchmark Programs

In addition, I have used 4 benchmarks from the Bapco Sysmark to evaluate the performance in Windows. These 4 benchmarks use popular Microsoft Office applications such as WinWord, Excel, Access and Powerpoint. The results of these tests are summarized in Figure 30.

Benchmark	Native	DynamoRIO	TRUSS	%overhead
Access	302.18	307.69	311.99	3.246
Excel	569.58	566.24	573.49	0.686
Powerpoint	365.22	364.25	369.9	1.281
WinWord	260.65	263.44	265.35	1.803
Average	374.407	375.405	380.182	1.542

Table 7: Performance of Sysmark benchmark programs on Windows

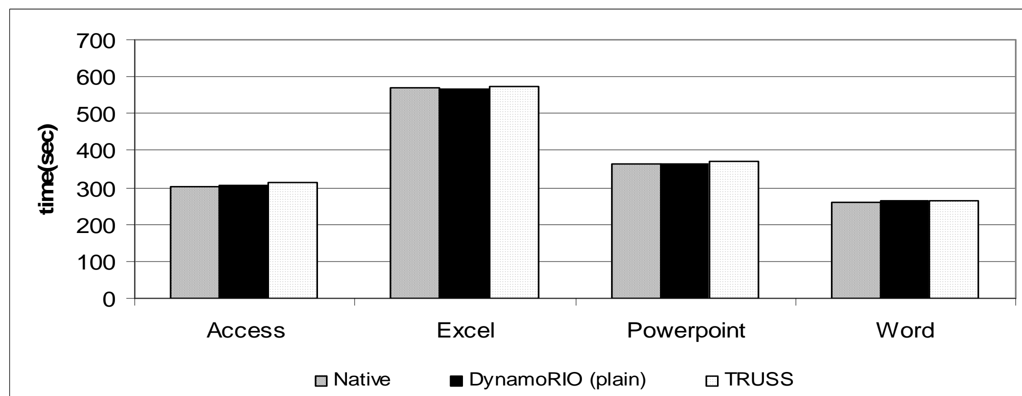

Figure 30: Performance of Sysmark benchmark programs on Windows

Figure 30 showed no significant slowdown due to TRUSS on the Microsoft Office benchmarks. One possible reason why the overheads differed so drastically between the SPEC CINT2000 benchmarks and the Bapco Sysmark benchmarks is that the number of call and return instructions in the SPEC CINT2000 benchmarks are very much larger than those in Bapco Sysmark benchmarks. This means that fewer

instructions are added and executed for return address checks and frame pointer checks. The number of call and return instructions observed in both benchmarks are shown in Tables 8 and 9. Note that in the Microsoft Office suite, calls and returns do not match up well. Many of the returns were replaced by jump instructions.

Benchmark	***Call-return* pairs**
Gzip	1,970
Vpr	2,589
Gcc	1,393
Mcf	2,053
Crafty	3,958
Parser	4,021
Eon	5,320
Perlbmk	5,755
Gap	3,980
Vortex	6,747
Bzip	3,209
Twolf	2,308

Table 8: Number of call and return pairs (in millions) in SPEC CINT2000 benchmark programs for Windows

Benchmarks	***call* instructions**	***return* instructions**
Access	1.81	1.72
Excel	4.76	4.25
Powerpoint	2.55	2.43
WinWord	1.27	1.20

Table 9: Number of call and return instructions (in millions) in Sysmark benchmark programs for Windows

7.5 Performance Of LibSafe And StackShield With SPEC CINT2000 Benchmark Programs

As a comparison, Table 10 shows the overhead of Libsafe on the SPEC CINT2000 benchmarks running in Linux. The average overhead incurred is negligible. It is rather obvious that Libsafe performs very much better than TRUSS. This is because Libsafe does not maintain a dynamic list of frame pointers. Instead, it invokes a GCC inbuilt function, `__builtin_frame_address`. The signature of this function is:

void * __builtin_frame_address (unsigned int LEVEL)

This method accepts an integer input that represents the level of the nested functions. However, the method is not safe for checking purposes because it assumes that the frame pointer will be in the memory location pointed to by the register %EBP. Some applications do not save %EBP in the process stack but rather use this register for their computational purposes. This means that the checking done using `__builtin_frame_address` does not guarantee that any buffer overflow will not exceed the frame pointer.

Table 11 shows the overhead of StackShield (the compiler approach) on the SPEC CINT2000 benchmarks running in Linux. The benchmark programs – `gcc`, `perlbmk`, `gap` – crashed. Moreover, `eon` could not be compiled with `shieldg++` compiler as this program requires newer version of `g++` compilers. The average overhead was 60%. Furthermore, StackShield does not support the GCC compiler optimization options such as `-fomit-frame-pointer` and `-funroll-all-loops`.

Benchmarks	**Native**	**Libsafe**	**%overhead**
Gzip	180.336	182.933	1.44
Vpr	207.795	205.523	-1.09
Gcc	80.929	81.864	1.155
Mcf	192.308	193.147	0.436
Crafty	118.144	119.092	0.802
Parser	218.046	220.82	1.272
Eon	212.979	214.884	0.894
Perlbmk	164.1136	165.937	1.111
Gap	91.037	91.388	0.386
Vortex	159.618	165.608	3.753
Bzip	183.703	184.258	0.302
Twolf	329.834	346.692	5.111
AVERAGE	178.237	181.012	1.557

Table 10: Performance of LibSafe with SPEC CINT2000 benchmarks in Linux

Benchmarks	Native	StackShield	%overhead
Gzip	180.336	298.607	65.584
Vpr	207.795	287.484	38.350
Gcc	80.929	Crash	-
Mcf	192.308	239.384	24.479
Crafty	118.144	180.9	53.118
Parser	218.046	233.191	6.946
Eon	212.979	Cannot compile	-
Perlbmk	164.1136	Crash	-
Gap	91.037	Crash	-
Vortex	159.618	251.211	57.383
Bzip	183.703	315.318	71.646
Twolf	329.834	487.221	47.717
AVERAGE	178.237	286.6645	60.833

Table 11: Performance of StackShield-0.7 with SPEC CINT2000 benchmarks in Linux

Chapter 8

Conclusion

Today's world is at a stage where it cannot function without computer systems. And a computer system loses its viability when its security is compromised. As such, the field of computer security has attracted great interests and investments. My work on TRUSS adds on to current research in the area of IT security and specifically, in the field of systems security.

The purpose of this thesis was to present a transparent, efficient and unified runtime solution for preventing a wide variety of known buffer overflow attacks, namely the return address attacks, format string attacks, vulnerable C function attacks, stack smashing attacks, heap overflows and GOT modifications. These attacks allow adversaries to intrude into systems and either steal critical information from the systems illegally or take control of the systems at privileged levels to perform unauthorized operations. Therefore, it is essential for computer systems to incorporate a defensive mechanism to thwart such attacks. Furthermore, it will be ideal if the mechanism makes minimal modification to the original executable and allows the application to execute normally. TRUSS is such a runtime security tool. It makes minimal alteration to the binary to ensure safe execution of the application. TRUSS has been implemented as a client program in DynamoRIO. The techniques incorporated in TRUSS are simple, robust and furthermore portable to other binary rewriting tools. Working only with binary executables, TRUSS can protect code running on both Linux and Windows without requiring any special hardware, access to the source code or even patches to the operating systems.

The performance overhead involved is among the major concerns of any runtime scheme. The performance evaluation of TRUSS has shown that its overhead is dependent on the application and operating systems and it is within a range that would be deemed acceptable to most users.

8.1 Limitations

Nevertheless, no single method of security is omnipotent. TRUSS, for instance, is not effective against buffer overflow of local variables allocated in the stack. It is rather difficult to extract the size of a local buffer from the binary. Thus, monitoring such buffers at runtime is not an easy task. Similarly, the Windows PE files do not carry information about the global variables. These are instead stored in the `COFF` file, which is usually not provided with the executables. Therefore, TRUSS does not have sufficient information from the binary to monitor these buffers as well.

Function pointer defense in TRUSS makes use of GCC specific code sequence to identify function invocations via function pointers. This is a limitation as this technique will be effective only in applications compiled with GCC compilers. Another limitation to be noted is that TRUSS does not allow instructions that are stored in the heap to be executed. Storing instructions in the heap and subsequently executing them may be legal in some applications. In such applications, TRUSS will raise false alarm.

8.2 Future Research

Future research can focus on eliminating the current limitations that TRUSS faces. In addition, more defense techniques can be included into TRUSS to strengthen its ability to protect applications. Such additional techniques can be incorporated into TRUSS to make it a comprehensive runtime security tool.

References

1. ANDREWS, M. HEAT: Runtime Interception of Win32 Functions. Technical Report CS-2003-1, 2003.

2. ARORA, D., RAVI, S., RAGHUNATHAN, A., AND JHA, N., K. Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Volume 1. Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, pages 178-183, 2005.

3. AVIJI, K., GUPTA, P., AND GUPTA, D. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, pages 45-55, 2004.

4. BAIN, C., FAATZ, D. B., FAYAD, A., AND WILLIAMS, D. Diversity as a defense strategy in information systems. Does evidence from previous events support such an approach?. In *Proceedings of the IFIP TC11/WG11.5 Fourth Working Conference on Integrity, Internal Control and Security in Information Systems: Connecting Governance and Technology* (November 15 - 16, 2001). M. GERTZ, E. GULDENTOPS, AND L. STROUS, Eds. IFIP Conference Proceedings, Volume 211. KLUWER B.V., Deventer, The Netherlands, pages 77-94. 2001.

5. BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000*

Conference on Programming Language Design and Implementation (Vancouver, British Columbia, Canada, June 18 - 21, 2000). PLDI '00. ACM Press, New York, NY, pages 1-12. 2000.

6. BARATLOO, A., SINGH, N., AND TSAI, T. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, CA, June 2000.

7. BARRANTES, E. G., ACKLEY, D. H., PALMER, T. S., STEFANOVIC, D., AND ZOVI, D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (Washington D.C., USA, October 27 - 30, 2003). CCS '03. ACM Press, New York, NY, pages 281-289. 2003.

8. BERGER, D., E. HeapShield: Library-Based Heap Overflow Protection for Free. UMass CS TR 06-28. 2006.

9. BHATKAR, S., DUVARNEY, D. AND SEKAR, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium*. USENIX. 2003.

10. BHATKAR, S., DUVARNEY, D. AND SEKAR, R. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, August 2005.

11. BRUENING, D. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. PhD thesis, Massachusetts Institute of Technology, September 2004. http://www.cag.csail.mit.edu/rio/.

12. Business Applications Performance Corporation. SYSmark 2004 SE. http://www.bapco.com/products/sysmark2004se.

13. CERT/CC.CERT Advisories. http://www.cert.org/stats/cert_stats.html

14. CHIUEH, T. AND HSU, F.H. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, Mesa, AZ, April 2001.

15. CIFUENTES, C., WADDINGTON, T., AND MIKE VAN EMMERIK. Computer Security Analysis through Decompilation and High-Level Debugging. In *Proceedings of the Eighth Working Conference on Reverse Engineering* (Wcre'01) (October 02 - 05, 2001). WCRE. IEEE Computer Society, Washington, DC, 375. 2001.

16. C0NTEXT. How to hijack the global offset table with pointers for root shells, September 2005. http://www.opensecurity.org/texts/6

17. COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., LOKIER, J. FormatGuard: Automatic Protection From printf

Format String Vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC. 2001.

18. COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

19. COWAN, C., PU, C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX Security Conference*, San Antonio, TX, January 1998.

20. Determina Inc. http://www.determina.com/

21. ENBODY, J., R. AND PIROMSOPA, K. Secure Bit: Transparent, Hardware Buffer-Overflow Protection. *IEEE Transactions on Dependable and Secure Computing,* Volume 3, No. 4. pages 365-376. October 2006.

22. ETOH, H. AND YODA, K. ProPolice: Improved stack-smashing attack detect on. *IPSJ SIGNotes Computer Security*, 014(025), October 2001. http://www.trl.ibm.com/projects/security/ssp

23. FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*

(Hotos-Vi) (May 05 - 06, 1997). HOTOS. IEEE Computer Society, Washington, DC, 67. 1997.

24. FOSTER, J. C., VITALY, O., NISH, B., NIELS, H. Buffer overflow attacks: Detect, Exploit, Prevent. Syngress Publishing, Inc, USA, 2005.

25. FRANTZEN, M. AND SHUEY, M. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C., August 2001.

26. HASTINGS, R. AND JOYCE, B. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, Usenix Association, pages 125-136, 1992

27. HUNT, G. AND BRUBACHER, D. Detours: Binary Interception of Win32 Functions. USENIX Technical Program - Windows NT Symposium 99, 1999.

28. JESSE, R., ROGER, K., SCOTT, L., ROBERT, C. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the 2003 ACM workshop on Rapid malcode*, October 27, 2003, Washington, DC, USA, 2003.

29. KIRIANSKY, V., BRUENING D., AMARASINGHE, S.P. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 05-09, pages 191-206, 2002.

30. KOCHER, P., LEE, R., MCGRAW, G., AND RAGHUNATHAN, A. Security as a new dimension in embedded system design. In *Proceedings of the 41st Annual Conference on Design Automation* (San Diego, CA, USA, June 07 - 11, 2004). DAC '04. ACM Press, New York, NY, pages 753-760, 2004.

31. KUPERMAN, B. A., BRODLEY, C. E., OZDOGANOGLU, H., VIJAYKUMAR, T. N., AND JALOTE, A.. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM* 48, 11, pages 50-56. November 2005.

32. LEA, D. A memory allocator. http: //gee.cs.oswego.edu/dl/html/malloc.html.

33. LEVINE, J. R. Linkers and Loaders. 1st. Morgan Kaufmann Publishers Inc. 1999.

34. LIBENZI, D. Guarded Memory Move (GMM) Buffer Overflow Detection And Analysis. http://www.infosecwriters.com/text_resources/pdf/gmm.pdf

35. MOORE, D., SHANNON, C., AND CLAFFY, K. Code-Red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on internet Measurment* (Marseille, France, November 06 - 08, 2002). IMW '02. ACM Press, New York, NY, 273-284. 2002.

36. Microsoft Portable Executable and Common Object File Format Specification.

37. MYERS, D. AND BAZINET, A. Intercepting Arbitrary Functions on Windows, UNIX, and Macintosh OS X Platforms. Technical Report, CS-TR-4585

38. NANCE, J. Product Review: Insure++. Linux J. 1998, 51es, 14. July 1998.

39. NECULA G. C., MCPEAK, S., WEIMER, W. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, January 16-18, 2002, Portland, Oregon, pages 128-139, 2002.

40. OZDOGANOGLU, H., BRODLEY, C., VIJAYKUMAR, T., JALOTE, A., AND KUPERMAN, B. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. Tech. Rep. TR-ECE 03-13, Purdue University School of Electrical and Computer Engineering, November 2003. www.smashguard.org/.

41. PIROMSOPA, K. AND ENBODY, R. Buffer-Overflow Protection: The Theory. In *Proceedings of the 6th IEEE International Conference on Electro/Information Technology*, East Lansing, Michigan, 2006.

42. PIROMSOPA, K. AND ENBODY, R. Arbitrary Copy: Bypassing Buffer-Overflow Protections. In *Proceedings of the 6th IEEE International Conference on Electro/Information Technology*, East Lansing, Michigan, 2006.

43. POE, J. AND LI, T. BASS: A benchmark suite for evaluating architectural security systems. *SIGARCH Computer Architecture News*, Volume 34, No. 4, pages 26-33. September 2006.

44. PRASAD, M. AND CHIUEH, T. A binary rewriting defense against stack-based buffer overflow attacks. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

45. PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. Optimistic incremental specialization: streamlining a commercial operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, pages 314-321. 1995.

46. PU, C., BLACK, A., COWAN, C., AND WALPOLE, J. A specialization toolkit to increase the diversity of operating systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan, December 1996.

47. RABEK, J. C., KHAZAN, R. I., LEWANDOWSKI, S. M., AND CUNNINGHAM, R. K. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode* (Washington, DC, USA, October 27 - 27, 2003). WORM '03. ACM Press, New York, NY, pages 76-82. 2003.

48. RICHARTE, G., Four different tricks to bypass StackShield and StackGuard protection. 2002. Tech. rep., Core Security Technologies, April 2002.

49. RINGENBURG, M.F. AND GROSSMAN, D. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the 12th ACM conference on Computer and communications security*, November 07-11, 2005, Alexandria, VA, USA, 2005.

50. ROBERTSON, W., KRUEGEL, C., MUTZ, D., AND VALEUR, F. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, San Diego, California, U.S.A., USENIX Association. pages 51-60, Oct. 2003.

51. RUWASE, O. AND LAM, M. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium* (NDSS), pages 159-169, February 2004.

52. SHANKAR, U., TALWAR, K., FOSTER, J. S., WAGNER D. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*. USENIX, Washington, DC, pages 201-220. 2001.

53. Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite. http://www.spec.org/osg/cpu2000/.

54. TSAI, T. AND SINGH, N. Libsafe 2.0: Detection of format string vulnerability exploits. White paper, Avaya Labs, February 2001.

55. VENDICATOR. Stackshield: A "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/.

56. WILANDER, J. AND KAMKAR, M. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, In *Proceedings of the 10th Network and Distributed System Security Symposium.* San Diego, California, February 6-7, 2003. Reston, VA: Internet Society, pages 149-162. 2003.

57. XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. Architecture support for defending against buffer overflow attacks. In *Proceedings of the 2002 Workshop on Evaluating and Architecting System dependability* (EASY-2002) University of Illinois at Urbana-Champaign, October 2002.

58. XU, J., KALBARCZYK, Z., AND IYER, R. K. Transparent runtime randomization for security. Technical report, Center for Reliable and Higher Performance Computing, University of Illinois at Urbana-Champaign, May 2003.

59. XU, J. AND NAKKA, N. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the 2005 international Conference on Dependable Systems and Networks* (Dsn'05) - Volume 00 (June 28 - July 01, 2005). DSN. IEEE Computer Society, Washington, DC, pages 378-387, 2005.

60. YOUNANA, Y., POZZAB, D., PIESSENSA, F. AND JOOSENA, W. Extended Protection against Stack Smashing Attacks without Performance Loss. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (ACSAC'06). Miami, pages. 429-438, December 2006.

61. ZHANG, C. AND JACOBSEN, H. A. TinyC^2 - towards a dynamic weaving aspect language based on C. In *Foundation of Aspect Oriented Languages (FOAL) jointly held with the 2nd International Conference on Aspect Oriented Systems and Design*, Boston, MA, March 2003.

62. ZHIVICH, M., LEEK, T. AND LIPPMANN, R. Dynamic Buffer Overflow Detection. *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.