SECURITY APPLICATIONS OF DYNAMIC BINARY TRANSLATION

by

DINO DAI ZOVI

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Bachelor of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

December, 2002

©2002, Dino Dai Zovi

Dedication

To all the independent, academic, and professional researchers who make computer security such an exciting field to work in.

Acknowledgments

This would not have been possible without the help and support of my advisor, Darko Stefanovic. Since this work began as a term project in his Spring 2001 Java Implementation seminar, he has encouraged me to continue exploring and working on the project. Professor Stefanovic allowed me to continue work on this project in his research group while I gained valuable writing and research experience. Finally, I would also like to acknowledge Professor Stefanovic for his detailed readings of my manuscripts that helped correct my sometimes creative abuses of the English language.

I would like to acknowledge my "partners in crime" Trek Palmer and David Worth. I would like to thank both Trek and David for the always enlightening brainstorming sessions and discussions which led to many of the ideas in this work.

Finally, I would like to thank family and friends for their support, especially when I was running around like Lewis Carroll's White Rabbit working on this project.

This work was supported in part by National Science Foundation ITR grants CCR-0219587 and CCR-0085792, and by Sandia National Laboratories.

SECURITY APPLICATIONS OF DYNAMIC BINARY TRANSLATION

by

DINO DAI ZOVI

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Bachelor of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2002

SECURITY APPLICATIONS OF DYNAMIC BINARY TRANSLATION

by

DINO DAI ZOVI

B.S., Computer Science, University of New Mexico, 2002

Abstract

The last 13 years have seen a large number of serious computer security vulnerabilities. Some of the most pernicious of these vulnerabilities have been buffer overflow and format string vulnerabilities in widely used software applications. A number of Internet worms have exploited these vulnerabilities to infect target hosts. The first part of this work introduces a framework for understanding and describing attacks that dynamically inject machine code into a process and the vulnerabilities that enable these attacks. The techniques used in these attacks are described in detail. The second part of this work describes the application of *dynamic binary translation*, previously a technique primarily for dynamic optimization, to stopping and mitigating these sorts of attacks. The implementations of several known techniques using a dynamic binary translation system are described in detail. Finally, some conclusions about the applicability of dynamic binary translation to computer security are made.

Contents

List of Figures x

1 Introduction 1

2 Memory Trespass Vulnerabilities 4

2.1 Buffer Overflow Vulnerabilities 5

2.2 Format String Vulnerabilities 6

2.2.1 Exploitation of Format String Vulnerabilities 7

3 Code Injection Exploits 12

3.1 Exploit Vectors 13

3.1.1 Stack Smash 13

3.1.2 Overwriting Stored Program Addresses 15

3.2 Code Injection Exploit Payloads 15

3.2.1 Unix Payloads 16

3.3 Defenses Against Code Injection Exploits 19

4 Dynamic Binary Translation **22**

4.1 SIND 22

4.2 DynamoRIO 25

4.2.1 DynamoRIO Interface 26

5 Security Approaches Using Dynamic Binary Translation **28**

5.1 Out-Of-Band Return Address Verification 29

5.1.1 Implementation 30

5.1.2 Conclusion 33

5.2 System Call Sandboxing 33

5.2.1 Implementation 35

5.2.2 Conclusion 36

6 Conclusions **38**

6.1 Future Work 40

References **42**

List of Figures

2.1 Vulnerable Invocation of `strcpy` 6

2.2 Potentially dangerous call to `printf` 7

2.3 Safe call to `printf` 7

2.4 Declaration of a function taking a variable number of arguments 8

2.5 Invocation of a function taking a variable number of arguments 8

2.6 Implementation of a function taking a variable number of arguments . . 9

2.7 Simulating writing a full 32-bit value 10

2.8 Format String to write `0xddccbbaa` to `0xbfffff10` 11

3.1 Buffer Overflow Vulnerability in BSD `fingerd` 14

3.2 Methods of Obtaining Address at Which Code is Executing 17

3.3 SPARC exploit payload to execute `/bin/ksh` 18

3.4 PowerPC exploit payload to execute `/bin/sh` 19

4.1 SIND modules 23

4.2 DynamoRIO Hook Functions 26

5.1 System Call Before Instrumentation 35

5.2 System Call After Instrumentation 36

Chapter 1

Introduction

On July 19, 2001, in the space of 14 hours 359,000 hosts were infected by the second incarnation of the "Code Red" Internet worm [18]. The worm spread by exploiting a vulnerability in Microsoft's IIS web server software. Almost 13 years earlier, Robert T. Morris' Internet worm had infected nearly 6,000 hosts on the early Internet [27]. Both worms spread by exploiting "stack-smashing" buffer overflow vulnerabilities in remote Internet hosts.

A security vulnerability is a software weakness that allows the program to enter unexpected states, possibly allowing malicious control. A general class of these vulnerabilities, which we call *memory trespass* vulnerabilities, occur when memory locations may be accessed outside of the semantics of a given programming language. These memory locations may contain data stored in other variables or even structures used by the programming language runtime environment. A knowledgeable and malicious user may overwrite values in these runtime structures with specially chosen values whereby the attacker may divert program control into dynamically injected machine code and take over the running process. The "stack-smashing" buffer overflow attack is an example of a *code injection exploit*, a class of attack that uses memory trespass vulnerabilities to overwrite

and corrupt data in stack activation records in order to cause a running program to execute attacker-supplied dynamically injected machine code.

There are several approaches to stopping or mitigating code injection exploits. The most natural approach is to address the memory trespass vulnerabilities by finding and patching these vulnerabilities through static source code analysis or by using stronger programming languages that do not permit unsafe memory accesses. While this approach is the most reliable, it is also the most labor-intensive. Automated tools may assist in the code auditing process, but they are not as thorough or accurate as a trained human auditor. Other approaches dynamically detect when an exercised memory trespass vulnerability may alter the execution path of the process and halt the execution of the running process. Similar approaches change the runtime memory layout of a process in order to make it more difficult to use a memory trespass vulnerability to divert control of the process. These approaches are typically implemented as compiler extensions requiring application source code access to build the security instrumentation into the executable. Finally, other approaches attempt to detect when control has been diverted by monitoring the execution of the process using operating system kernel extensions. However, many of these approaches are not possible when either the source code to the application or operating system is not available. An alternative implementation method for a variety of dynamic security mechanisms that operates without requiring application or operating system source code (on 90% of hosts, neither is available) is dynamic binary translation.

Current high-performance Java virtual machines such as IBM's Jikes Research Virtual Machine [2] and Sun's HotSpot Virtual Machine [17] make use of dynamic monitoring and statistic gathering to guide run-time optimizing code transformations. These techniques build upon earlier work on dynamic optimization of running binaries [3]. Similar systems perform run-time translation to execute foreign-architecture binaries ([6] [10]). These *dynamic binary translation* systems share a general approach where machine code fragments may be interpreted or framed for direct execution and where frequently-used

fragments are transformed and saved for later use in a fragment cache where they may be combined with other fragments. Such a lazy approach minimizes the time spent operating on the machine code fragments in order to maximize the time executing the transformed fragments. Currently available dynamic binary translation systems include SIND [23] and DynamoRIO [4].

The approach taken by dynamic binary translation systems can be used to perform security-related transformations to mitigate or stop code injection exploits. By using a dynamic binary translation system, these security augmentations can be implemented without source code access or modification to the application or operating system. This advantage enables dynamic security systems to be quickly deployed to protect a wider scope of applications. For example, an application security system using dynamic binary translation would only require restarting the application for it to be protected. In some cases, it will even be possible to attach the SIND binary translation system to a running process.

This work is presented in several chapters. The first introduces a new classification of computer security vulnerabilities, *memory trespass vulnerabilities*, which generalizes several already well-known classes of vulnerabilities. The second chapter develops a terminology and framework for describing and understanding code injection exploits while providing detailed descriptions and examples of the components of a code injection exploit. The fourth chapter discusses the available tools for dynamic binary translation. The fifth chapter discusses several approaches using dynamic binary translation to protect applications against code injection exploits.

Chapter 2

Memory Trespass Vulnerabilities

Memory trespass vulnerabilities are software weaknesses that allow memory accesses outside of the semantics of the programming language in which the software was written. Memory trespass vulnerabilities pose a serious threat to system survivability and security. For example, overwriting memory allocated for programming language variables may alter the execution of the process. Even worse, the corruption of programming language runtime structures may cause the program to crash or even enable a knowledgeable attacker to take complete control over the running process and cause it to execute supplied machine code.

Common memory trespass vulnerabilities include buffer overflow, format string, out-of-bounds array access, signed integer cast, integer overflow, and double-free vulnerabilities. The two most common types of memory trespass vulnerabilities, buffer overflow and format string vulnerabilities are discussed in detail below.

2.1 Buffer Overflow Vulnerabilities

Generally, a buffer overflow vulnerability occurs when too much data is written to a fixed-size buffer and values in memory following the space allocated for the buffer are overwritten. The overwritten data following the buffer may be other program variables or control structures used by the language runtime environment. By crafting the input sent to a vulnerable program, these control structures may be overwritten in such a way as to redirect execution of the program to a location of the attacker's choosing.

In the C programming language, character strings are represented as a byte array terminated by the `NULL` character, the character with integer value zero. The length of the string is not explicitly stored – it is calculated when needed by counting the elements in the string before the terminating `NULL`. Most of the string operations in the C standard library do not take an explicit length argument – they implicitly use the length of the source string. The unbounded string operations in the C standard library are the most common cause of buffer overflow vulnerabilities. String handling functions such as `strcpy`, `strcat`, and `sprintf` do not take into account the size of the destination buffer and instead use the implicit length of the source string, which may be larger than the destination buffer. When the source string is larger than the destination buffer, a buffer overflow may occur. If the source string may be controlled by the user, a malicious user may attempt to perform a buffer overflow attack.

For example, consider the invocation of `strcpy` in Figure 2.1. The call to `strcpy` copies successive characters from `str` into `buff` until it encounters a `NULL` byte in `str`, indicating the end of the string. If `str` is more than 256 bytes long, bytes from `str` will overwrite data stored in the memory locations following the storage allocated for `buff`. In this example, `buff` is allocated on the stack, but as shall be described below, this sort of vulnerability may be exploitable regardless of where the overflowed buffer is allocated.

```
void foo(char* str)
{
  char buff[256];
  ...
  strcpy(buff, str);
  ...
}
```

Figure 2.1: Vulnerable Invocation of `strcpy`

2.2 Format String Vulnerabilities

The C Standard Library includes a number of functions performing formatted input and output conversion. These facilities include the well-known `printf` and `scanf` family of functions. These functions take as an argument a *format string* consisting of literal characters and *format directives* that specify what input or output conversions are performed and a variable number of other arguments of number and type specified by the directives in the format string. A function using the C variable-length argument facilities (`stdarg(3)`) cannot determine the number or types of the passed arguments. In the formatted input and output functions, this information is encoded in the format string. A *format string vulnerability* ([20]) occurs when untrusted user input is present in the format string argument, allowing the user to influence the actions taken by the formatted input and output functions.

The most common instances of format string vulnerabilities are through careless use of the `printf` function. The `printf` function prints formatted output to the standard output stream taking a format string argument and a variable number of other arguments. Consider the invocation of `printf` in Figure 2.2. If the string variable `str` includes any input from the user, the user may insert format string directives into `str`. When `printf` evaluates its first argument as a format string, these format string directives will cause

```
char* str;
...
printf(str);
...
```

Figure 2.2: Potentially dangerous call to `printf`

```
char* str;
...
printf("%s", str);
...
```

Figure 2.3: Safe call to `printf`

`printf` to attempt to retrieve function arguments that were not supplied by the programmer. The C variable-length argument facilities are not able to determine that no arguments have been supplied and will return spurious values from the stack. A knowledgeable attacker may use this condition to output values from arbitrary memory locations or, through clever use of format directives discussed below, write arbitrary values to arbitrary memory locations. A corrected invocation of `printf` is listed in Figure 2.3.

2.2.1 Exploitation of Format String Vulnerabilities

The exploitation of format string vulnerabilities relies on the ability of the attacker to supply the variable arguments to the formatting function and clever use of the `%n` format directive. Upon examining the implementation of the C variable-length argument facility, it will become clear that an attacker can easily supply arbitrary arguments to the formatting function.

Under normal C calling conventions, a number of arguments will be passed in registers

```
int foo(int a, int b, ...);
```

Figure 2.4: Declaration of a function taking a variable number of arguments

```
foo(1, 2, 3, 0);
```

Figure 2.5: Invocation of a function taking a variable number of arguments

with the rest pushed onto the program stack. On some architectures, notably the IA-32 architecture, all function arguments may be passed on the stack. With a function taking a variable number of arguments, the compiler will store any of the optional arguments on the stack. For example, consider a function declared as in Figure 2.4 and invoked as in Figure 2.5. On an architecture where some arguments are passed in registers, the values 1 and 2 might be passed in registers while the values 3 and 0 would be passed on the stack. The definition of the function `foo` should use the macros `va_start`, `va_arg`, and `va_end` to step through the variable argument list. Note that there is no facility for the called function to retrieve the number or type of arguments supplied.

In our example, `foo` (Figure 2.6) is unable to determine when it has retrieved all the arguments supplied by the caller. Typically, the end of the list is determined by an explicit length argument, a special marker value, or is implicit in the structure or another argument. In our implementation `foo`, the value 0 serves as a special value marking the end of the argument list. For example, in the formatted input and output functions, the number of arguments the caller retrieves is determined by interpretation of the format directives in the format string. If the attacker can insert arbitrary format directives into the format string, they may direct the formatting function to use other values stored on the stack. Most format directives take an optional field, the argument selector, consisting of a decimal digit followed by a $ character that specify which argument to access. The attacker may

```
int foo(int a, int b, ...)
{
    va_list ap;
    int last = -1;

    va_start(ap, b);
    while (last != 0) {
        last = va_arg(va_list, int);
    }
    va_end(ap);
}
```

Figure 2.6: Implementation of a function taking a variable number of arguments

calculate an argument number that selects a value stored in other attacker-supplied input or even the format string itself. For example, the attacker may place a memory address in the format string and an argument selector pointing to it to output the value stored in any location in memory. In addition, as shall be shown, an attacker may use a similar technique to write any value to any location in memory.

The `%n` directive instructs the formatting function to store the number of characters written so far into the address pointed to by the next argument. The attacker can control this number by specifying minimum field widths in the format directives. For example, consider the format string `%478d%n`. The format directive `%478d` will convert an integer to a string 478 characters long containing a decimal number padded on the left with zeros. The following `%n` directive, will write the integer 478 to the address pointed to by the next argument. As discussed above, the attacker may use the optional argument selector to control the value used by the format directive. Using these two tricks, the attacker may write a small integer to an arbitrary memory location. However, on architectures permitting byte-aligned word writes, four overlaid writes may be used to simulate writing an arbitrary word value to an arbitrary location. For example, consider Figure 2.7, depicting

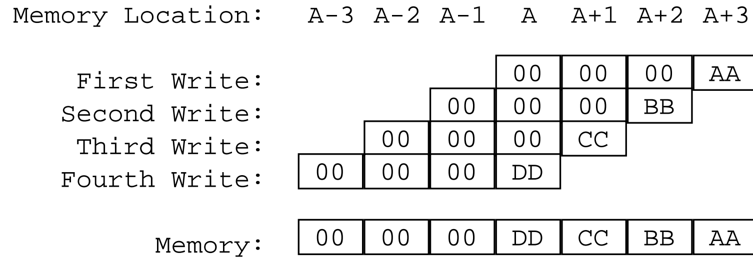

Figure 2.7: Simulating writing a full 32-bit value

how the hexadecimal value `0xddccbbaa` may be written using four overlaid writes of the values `0xaa`, `0xbb`, `0xcc`, and `0xdd`. In our running example, we assume a big-endian architecture that allows unaligned writes.

We will now consider how a format string may be used to write an arbitrary value to an arbitrary memory location in more detail. Consider the format string in Figure 2.8. This format string will write the hexadecimal value `0xddccbbaa` to memory location `0xbfffff10`, assuming the format string itself may be considered the 12th argument when the argument selector is used (if it is not, this value may be changed appropriately). The first 16 bytes are the memory addresses where the byte-sized values will be written to simulate the full word value. The following format directive prints a decimal number zero-padded to 154 bytes. This raises the number of characters output so far to 170 bytes (`0xaa` in hexadecimal). The next format directive uses the argument selector to select the first memory address in the format string and writes this value to that location. The following format directives do the same for the remaining bytes. As depicted in Figure 2.7, this simulates a write of the full 32-bit value to our chosen memory location. Careful construction of format strings using this technique allows an attacker to write an arbitrary value to an arbitrary location.

```
\xbf\xff\xff\x10\xbf\xff\xff\x11
\xbf\xff\xff\x12\xbf\xff\xff\x13
%154d%$12n%17d%$13n%17d%$14n%17d%$15n
```

Figure 2.8: Format String to write `0xddccbbaa` to `0xbfffff10`

Chapter 3

Code Injection Exploits

A code injection exploit consists of several components: vulnerability, vector, payload, string, and delivery.

- For exploitation to be possible, there must be a *vulnerability* present in the program allowing a malicious user to corrupt program memory.
- The exploit *vector* is the mechanism by which the exploit diverts control of the vulnerable program into the payload, executing the attacker's code with the capabilities and privileges of the vulnerable program [1]. Several of the more common types of code injection vulnerabilities and exploitation methods are described below.
- The exploit *payload* is the machine code to be injected into the process address space for execution.
- The payload and vector are encoded together into the exploit *string*, the sum of the input sent to exploit the vulnerability. The exploit string satisfies the constraints

[1]Note that a code injection exploit is different from a computer *virus*. A virus infects and alters the stored executable file of the program, whereas code injection exploits infect a program while it is running.

of communication with the vulnerable program required to reach the vulnerable portions of the code. For example, the exploit string may encode the payload and vector in the headers and filename, respectively, of a well-formed HTTP request.

- This entire sequence of input is sent to the vulnerable program by means of the exploit *delivery*. The exploit delivery may be through a remote network connection, local command-line argument, or local environment variable.

3.1 Exploit Vectors

A code injection exploit uses an *exploit vector* to direct execution of the process to a location of the attacker's choosing. The most common exploit vector is the *stack smash*, where the return address in a stack frame is overwritten through exploitation of a buffer overflow. In general, memory trespass vulnerabilities that allow arbitrary memory writes may be used to overwrite any of a multitude of stored program addresses in the process' address space including return addresses or other addresses. Both the stack smash and some commonly targeted stored program addresses are discussed below.

3.1.1 Stack Smash

In compiled C code, whenever a procedure is called, an activation frame is allocated on the stack segment. The called procedure uses the activation frame to store the old values of used registers and to reserve space for data local to the execution of the procedure. The activation frame also stores the *return address*, the memory address at which execution will resume when the procedure has finished executing. When the called procedure completes execution, the saved registers are restored and control resumes at the *return address*.

A *stack smashing* [22] attack is a buffer overflow that overflows a buffer allocated

```
char line[512];
...
gets(line);
...
```

Figure 3.1: Buffer Overflow Vulnerability in BSD `fingerd`

on the vulnerable program's stack segment, causing data in the activation record to be overwritten. A knowledgeable attacker may overwrite the saved return address with an address within user input, causing the program to "return" into attacker-supplied machine code.

This form of attack was first publicly identified in the 1988 Morris Worm. The Morris Worm exploited a stack overflow in the BSD `fingerd` daemon on the VAX architecture. The BSD `fingerd`, running from the Internet "Super Server" `inetd`, accepted user input via a call to the C standard library function `gets`, storing the input in a fixed-length buffer on the stack. The library function `gets` reads a line of input from standard input into the passed buffer until a terminating newline or *end-of-file* character is encountered in the input. The vulnerable code resembled the fragment in Figure 3.1.1.

The call to `gets` reads a line of input and stores it in the variable `line`, which is declared as an automatic variable allocated in the current stack frame. If the input is longer than 512 characters, other information on the stack will be overwritten by the call to `gets`. The Morris Worm sent an exploit string 536 bytes long containing VAX machine code and replacement values for the activation record [19]. This overwrote the return address with the address of the exploit payload on the `fingerd` process' stack, causing execution to divert into the supplied machine code which spawned a `root` (the Unix Super User) privileged shell.

Stack smashing buffer overflow attacks and their causes are well understood. However,

they are still one of the most common form of security vulnerabilities.

3.1.2 Overwriting Stored Program Addresses

Several types of memory trespass vulnerabilities allow the attacker to write a chosen value to a chosen location. Vulnerabilities that allow this include the previously discussed format string vulnerabilities as well as heap corruption and double-free vulnerabilities. The typical target of these arbitrary writes are stored program addresses used by the programming language runtime environment.

Stored program addresses are used in many different contexts in compiled C programs. As mentioned above, when a stack activation record is created, the *return address* of the caller is stored in it. Return addresses on the stack are frequent targets of arbitrary writes, but in some cases they may be too dynamic and their exact location may be difficult to predict. Shared library function addresses used by the runtime linker are a frequent target with a predictable location. For example, on Unix systems that use the Executable and Linking Format (ELF) [7], the addresses of imported shared library functions are stored in the Global Offset Table. By overwriting an entry in the Global Offset Table, an attacker may cause the next call to a commonly used function to instead jump into the attacker's exploit payload. Finally, if the program makes use of them, stored function pointers or `setjmp(3)/longjmp(3) jmpbufs` may also be overwritten.

3.2 Code Injection Exploit Payloads

Code injection exploit payload is most commonly referred to as *shellcode*, as the most common task is to execute an interactive shell. However, many other more specialized functions are both possible and common, so we will discuss the exploit payload in the general sense. Generally, the exploit payload is the machine code fragment injected into

the vulnerable program for execution.

Although the payloads are typically reusable and interchangeable, there are several constraints on their construction and execution that make the task of writing exploit payloads non-trivial. The code fragments must be completely position independent, making little or no assumptions about their execution environment. In addition, because the code fragments are encoded within `NULL`-terminated strings, they must avoid having `NULL`-bytes in their instruction encodings or else the exploit string may be truncated. This makes it impossible to encode certain instructions. For example, this often presents difficulties in the encoding of forward branches and instructions to clear registers. The code fragments must be written in hand-coded assembly language specific to the target platform and clever tricks must often be employed to avoid `NULL`-bytes.

There are several tricks commonly employed to facilitate construction of complex payloads within the previously mentioned constraints. The payload may use the inherited value of the stack pointer for temporary storage as long as its use does not overwrite the executing payload. An ad-hoc data section may be included in the exploit string if the executing payload can determine the memory address at which it is executing. This can be done in a variety of clever ways, some of which are demonstrated in Figure 3.2 [21]. With a pointer to the executing code, the payload can load values from the ad-hoc data section at the end of the payload. Many payloads store strings or other data structures in this manner. For example, the path to an executable or a shell command to execute may be stored there. To avoid the difficulty of `NULL`-byte free encoding, the bulk of the payload may be *exclusive-or*ed with a constant and stored in the payload data section.

3.2.1 Unix Payloads

The assembly language interface for performing system calls under Unix-like operating systems makes construction of exploit payloads for these systems relatively straightfor-

Moving value of program counter to register `i7` on SPARC:

```
a: nop
b: bn,a a
c: bn,a b
   call c
```

Moving value of program counter to register `r31` on PowerPC:

```
a: xor. r5, r5, r5
   bnel a
   mflr r31
```

Moving value of instruction pointer to register `eax` on IA-32:

```
      call next
next: pop %eax
```

Figure 3.2: Methods of Obtaining Address at Which Code is Executing

ward. Under Unix-derivative operating systems, a system call is performed by initiating a software interrupt or system trap. The specific system call is selected by placing the system call number, usually an unsigned integer, in a register. Under most operating systems, the system call numbers remain constant under every release, as this is what permits statically linked executables to run under more than one release of the operating system. However, some operating systems such as AIX change system numbers with each operating system release to discourage the use of statically linked executables. System call parameters are pushed onto the stack or stored in registers. Upon completion, the result of the system call is typically returned in a register.

For local Unix exploits, the typical exploit payload simply performs the `exec` system call to execute `/bin/sh`, giving the attacker a shell with the privileges of the vulnerable program. For remote exploits, the typical payload listens on a TCP port for a connection

```
set  0x2f62696e, %l0     ! (void*)sh = "/bin";
set  0x2f6b7368, %l1     ! (void*)sh + 4 = "/ksh";
std  %l0, [%sp - 24]
st   %g0, [%sp - 16]
sub  %sp, 24, %o0        ! %o0 = "/bin/ksh";

xor  %sp, %sp, %o1
std  %o0, [%sp - 8]
sub  %sp, 8, %o1         ! %o1 = {NULL};

xor  %sp, %sp, %o2       ! %o2 = NULL;

mov  59, %g1
ta   8                   ! execve(sh, argv, NULL);
```

Figure 3.3: SPARC exploit payload to execute `/bin/ksh`

upon which it executes `/bin/sh`, allowing the attacker to connect to an interactive shell running on the remote machine. A close variant makes an outbound TCP connection on which it runs `/bin/sh`. In this scenario, the attacker starts a process listening on a TCP port and when the payload is executed a connection is made to the attacker's listening process and the attacker is presented with an interactive shell on the remote machine.

Two example Unix payloads are listed in Figures 3.3 and 3.4. The payloads execute an interactive shell under the Solaris operating system on the SPARC architecture and under the Mac OS X operating system on the PowerPC architecture, respectively. These payloads were written to ensure a `NULL`-byte-free encoding and have been tested in simple dynamic machine code injection exploits.

```
execsh:
        ;; Don't branch, but do link.  This gives us the
        ;; location of our code.  Move the address into
        ;; GPR 31.
        xor.   r5, r5, r5        ; r5 = NULL
        bnel   execsh
        mflr   r31

        ;; Use the magic offset constant 268 because it
        ;; makes the instruction encodings null-byte free.
        addi   r31, r31, 268+36
        addi   r3, r31, -268     ; r3 = path

        ;; Create argv[] = {path, 0} on the stack.
        stw    r3, -8(r1)        ; argv[0] = path
        stw    r5, -4(r1)        ; argv[1] = NULL
        subi   r4, r1, 8         ; r4 = {path, 0}

        ;; 59 = 30209 >> 9 (trick to avoid null-bytes)
        li     r30, 30209
        srawi  r0, r30, 9        ; r0 = 59
        sc                       ; execve(path, argv, NULL)
path: .asciz "/bin/sh"
```

Figure 3.4: PowerPC exploit payload to execute `/bin/sh`

3.3 Defenses Against Code Injection Exploits

A successful code injection attack requires the existence of an exploitable memory trespass vulnerability, successful exploitation of the memory trespass vulnerability in order to create an exploit vector, sufficient knowledge of the runtime memory image to create the exploit vector, the ability to execute the payload, and the capability of the payload to perform actions of use to the attacker. Defenses against code injection exploits stop or restrict one or more of these phases.

Static defenses against code injection exploits ([14] [30] [25]) target the first phase of the attack, the existence of the vulnerability. Static source code analysis tools may use supplemental code annotations or declarations to aid in the operation of automated tools. Such automated tools have been successful in detecting many classes of buffer overflow and format string vulnerabilities. However, the adoption of such tools has not been widespread.

Dynamic defenses such as StackGuard [8] and Stack Shield [29] detect and prevent the operation of the exploit vector. StackGuard places a *canary* value before the return address in the stack activation record which is verified before the subroutine returns. If the canary value was overwritten by an attempted stack smashing attack, an executable compiled with the StackGuard compiler extensions will halt execution and log the exploitation attempt. Stack Shield takes a different approach by storing the return addresses out-of-band so that they may not be overwritten by an overflowed stack buffer. Both systems require application source code for recompilation using static compiler extensions.

As an alternative approach to detecting and preventing exploit vectors, the runtime memory image of the executable may be altered in such a way as to make construction of a correct exploit vector infeasible. Forrest, Somayaji, and Ackley [12] describe several techniques for introducing diversity into computer programs at both compile-time and load-time. The techniques described involve adding or deleting nonfunctional code, reordering code, or changing memory layout. As a sample implementation, they modified the GNU C Compiler (`gcc`) to add a random amount of stack padding to any stack allocations larger than a chosen threshold.

The ability to execute payload may be hampered by making writable segments of the process address space non-executable. The Sun Solaris Operating Environment on SPARC hardware is able to make the stack segment non-executable. With the option turned on, any process attempting to execute code on the stack is terminated (and optionally logged to `syslog`). In programs conforming to the 64-bit SPARC ABI, a non-executable stack

is the default. Future pure 64-bit releases of the operating system will implement this for all system binaries. The PaX [28] project is an attempt to bring hardware-enforced non-executable pages to the IA-32 platform. Intel Pentium and newer IA-32 processors contain separate Translation Lookaside Buffers for both instructions and data. These two buffers are usually kept consistent as they are both loaded from the same page table, but they may be manipulated such that they contain different entries and page protections. PaX uses this architecture feature to implement non-executable pages on IA-32 Linux and Windows systems.

Finally, some novel approaches involve detecting and hampering the execution of foreign code. Process Homeostasis [26] is a Linux kernel extension that detects anomalous behavior in running processes and creates exponential delays in system calls when sequences of system calls do not match the normal execution profile of the process. Such an approach globally makes it difficult for an attacker to use even a successful exploit to her advantage.

Chapter 4

Dynamic Binary Translation

A *Dynamic Binary Translation* system dynamically decomposes an executable into machine code fragments that may be executed directly or in a software interpreter. While executing in an interpreter, statistics may be gathered on how often the fragment is executed and how often specific branches in the fragment are taken. The most frequently used fragments are transformed and cached for later execution. Once fragments are in the cache, they may be merged with other fragments when they are found to be separated by a direct unconditional branch. This general architecture is used for dynamic program optimization, introspection, instrumentation, and architecture translation.

Several notable projects using dynamic binary translation are SIND, DynamoRIO [4], FX!32 [6], and DAISY [10]. SIND and DynamoRIO are described in more detail below.

4.1 SIND

SIND is an multi-platform open-source framework for binary translation currently in development. SIND uses the general dynamic binary translation system architecture de-

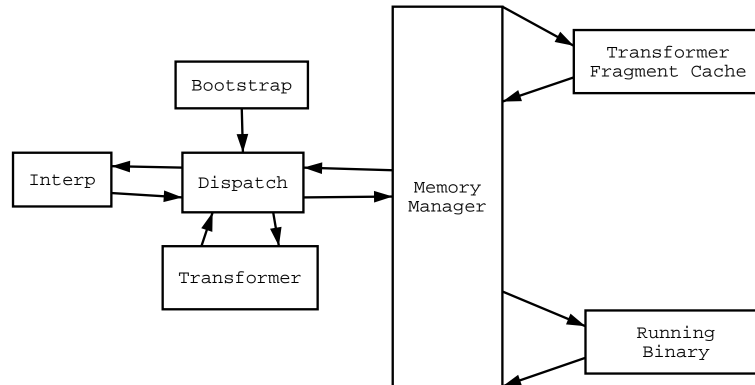

Figure 4.1: SIND modules

scribed above, but applies modular and object-oriented design to provide an easily extensible and flexible framework for the research and development of applications and techniques for dynamic binary translation. The current target platforms are Solaris on SPARC and Mac OS X on PowerPC.

The SIND system consists of several modules, depicted in Figure 4.1. The primary modules of interest are the bootstrapper, dispatcher, interpreter, memory manager, code transformers, and fragment cache. One of the goals in the development of SIND is the evaluation of techniques for dynamic binary translation. Therefore, many of the modules are implemented using several approaches of varying complexity and efficiency. This will allow systems built using SIND to use the fastest implementations except where a more complicated implementation is needed for functionality required by the dynamic binary translation system. This approach also allows optimized module implementations using platform-specific features to be used instead of the more portable but slower implementations where available.

The bootstrapper is the mechanism by which application control is assumed by the SIND system. Currently, three techniques are implemented using the `ptrace` [16] process debugging facility, the `procfs` [15] process filesystem, and the dynamic linking

facilities of the Solaris operating system. The `ptrace` and `procfs` implementations set a breakpoint trap at the initial execution point of the executable. When the breakpoint trap is triggered, the application context is saved and restored in the software interpreter where execution is resumed. The third technique uses a shared library which upon initialization instruments the executable to hand over control to SIND when execution starts at the entry point `_start`. Currently, this is done by mapping the page containing the `_start` symbol non-executable and registering a signal handler to catch when execution in that page is attempted.

The dispatcher controls context switches between the application code executing directly on the processor and in the software interpreter. The dispatcher also is responsible for implementing trace identification to create fragments, determining which fragments are to be placed in the fragment cache, and when frequently-executed fragments are to be linked into a super-fragment. The dispatcher is also responsible for choosing the appropriate module implementations based on the features in the host platform.

The SIND interpreter is a software-based instruction set architecture interpreter. Unlike architecture emulators, the interpreter simply interprets the instruction stream as correctly and efficiently as possible without simulating the low-level details of the emulated architecture. This is simplified by only interpreting user-mode instructions. During interpretation, the interpreter collects statistics and profiling data on the code fragment for potential use by the code transformers.

The memory manager abstraction controls memory reads and writes in the address space of the application when it is running in the interpreter. This abstraction allows SIND to operate in a different address space from the application if desired where mutual access to data could be provided through shared memory. Doing so would protect SIND structures and the fragment cache from a potentially malicious application. Alternatively, the memory manager could transparently allow SIND and the application to share the same address space.

Using data gathered during code trace interpretation, the SIND code transformers rewrite the instruction stream for increased performance, monitoring, or application security. Multiple transformers may be chained together to perform several operations on the code fragments. An abstracted instruction set interface may be used by the transformers to implement instruction-set independent transformations where possible.

The fragment cache manages and stores the most frequently executed application code fragments. In addition, the fragment cache is responsible for inserting prologue and epilogue code to manage the SIND context switch when control enters and leaves the fragment. The fragment cache will also support fragment linking whereby fragments that branch to each other will be linked together into a super-fragment, eliminating unnecessary SIND context switching.

The design of SIND is described in further detail in [23].

4.2 DynamoRIO

DynamoRIO [4] is an extension of the earlier Dynamo [3] system. Dynamo was a dynamic binary optimization system for HP-PA binaries on the HP-UX operating system. DynamoRIO incorporates an updated version of Dynamo that operates on IA-32 binaries under both Windows and Linux. Although still in development, the RIO runtime introspection and optimization architecture has been used to implement a secure program execution method called *program shepherding* [13].

RIO takes a slightly different approach to dynamic binary translation from SIND and the earlier Dynamo. Instead of interpreting the binary before gathering fragments, all application code is executed out of code caches. The application code is broken up into *basic blocks*, instruction sequences ending with a single control transfer instruction, which are executed directly out of the basic block cache. At the end of a basic block, RIO resumes

```
void dynamorio_init();
void dynamorio_exit();
void dynamorio_basic_block(void*, app_pc, InstrList*);
void dynamorio_trace(void*, app_pc, InstrList*);
void dynamorio_fragment_deleted(void*, app_pc);
```

Figure 4.2: DynamoRIO Hook Functions

control and examines the target of the control transfer instruction. If the target is not in the basic block cache, RIO will place it in the cache and resume execution in the new basic block. If the target is already in the cache, the two basic blocks will be linked together with a direct jump. During execution, DynamoRIO identifies potential *trace heads*. These are targets of backward branches or exits from existing traces. Execution counts are maintained for each of these trace heads. When a predefined execution threshold is reached, the *hot trace* is placed in the trace cache. These collected hot traces are the most frequently executed code fragments in the executable and should be the targets of dynamic optimizations.

4.2.1 DynamoRIO Interface

DynamoRIO exports an Application Programming Interface (API) to enable construction of custom optimization and instrumentation utilities, called *DynamoRIO clients*. The API includes well-structured abstractions for creating and manipulating IA-32 instructions as well as DynamoRIO-specific data structures. This allows the DynamoRIO client programmer to operate on instruction streams without detailed knowledge of IA-32 instruction encoding, only an understanding of the IA-32 instruction set is necessary. The API also defines several hook functions that will be called by DynamoRIO if they are defined by the user in a shared library. The declarations of these functions are listed in Figure 4.2.1.

The `dynamorio_init` and `dynamorio_exit` hook functions are called when DynamoRIO is initialized and about to terminate, respectively. When a basic block is created, the hook function `dynamorio_basic_block` is called to allow it to transform the code fragment before it is placed in the basic block cache. Similarly, the `dynamorio_trace` function is called when a trace is created to allow the trace to be transformed before it is placed in the trace cache. Whenever a fragment is deleted from either cache, the DynamoRIO client is notified through invocation of `dynamorio_fragment_deleted`.

Chapter 5

Security Approaches Using Dynamic Binary Translation

Many defenses against code injection exploits are implemented as compiler extensions. For example, StackGuard and Stack Shield are implemented as extensions to the GNU C Compiler. However, for a number of reasons, re-compilation of security-critical applications may not be feasible. This may be due to a lack of available source code, time, or expertise. Fortunately, many of the techniques used by these protection mechanisms may be performed using dynamic binary translation.

The following sections discuss the use of dynamic binary translation to implement several techniques to defend against code injection exploits. *Out-of-Band Return Address Verification* is a code rewriting technique that is able to stop all stack smashing attacks. *System Call Sandboxing* is a technique to limit the effects of untrusted and potentially malicious code.

5.1 Out-Of-Band Return Address Verification

Stack Shield [29] is a tool to protect applications against stack smashing attacks. It requires no source code modifications. One of the techniques implemented by Stack Shield modifies the subroutine calling sequence to store return addresses in a dedicated stack rather than in the activation records on the program stack segment. The implementation adds a fixed-size array to the data segment of the program for storage of the global return address stack. The subroutine epilogues are modified to push their return address onto this stack. Before returning, the subroutines compare the return address from the activation record with the return address popped off the global return address stack. If they match, the subroutine returns normally. However, if they do not match, the program can (depending on a compile-time option) either halt or continue execution using the return address from the global return address stack.

We implemented a generalization of the technique used by Stack Shield, which we call *Out-Of-Band Return Address Verification* as a DynamoRIO client. We instrumented each call instruction to push the expected return address onto an out-of-band return address stack and instrumented each return instruction to compare the target location with the address on the top of the out-of-band stack. If the return address from the activation record on the program stack does not match the address on the out-of-band return address stack, this indicates that the return address has been overwritten and several actions will be taken: the intrusion attempt will be logged and execution can be halted or allowed to continue using the out-of-band return address.

This technique is effective against all *stack smashing* buffer overflow exploits and other exploitation methods that target return addresses on the stack. In a stack smashing attack, only data following the overflowed buffer can be overwritten, so the out-of-band stack stored in heap memory cannot be overwritten and the overwritten return address will be detected. Format string or heap corruption attacks that target the return address on the

program stack will similarly be detected and prevented.

5.1.1 Implementation

We implemented Out-Of-Band Return Address Verification as a DynamoRIO client using the DynamoRIO interface. The implementation is able to execute any binaries that DynamoRIO executes and successfully detects attempted modifications of the return address without noticeable performance degradation.

To ensure that all code is captured before execution, instrumentation is applied at the basic block level using `dynamorio_basic_block`. Before being placed into the code cache, each block of code is instrumented to call a monitoring function just before executing each direct call, indirect call, or return instructions. These monitoring functions store the correct return address out-of-band on function calls and verify that the return address agrees with the out-of-band value on function returns.

The current implementation uses a fixed-size array treated as a stack for return address storage. On a function call, the address of the instruction following the call instruction is pushed onto the stack. On a function return, the target address of the return is compared to the address stored on the out-of-band return address stack. If the addresses do not match, the event is logged as a possible intrusion attempt. At this point, several actions may occur (depending on system administrator preference). The process may be halted either by a call to `_exit` to terminate or `abort` to terminate and dump core for later examination. Alternatively, the out-of-band return address may be used to attempt to continue execution. Although program execution will continue, the stack smash attempt may have overwritten other data structures on the stack, which may cause the program to crash.

Challenges

The implementation posed some engineering challenges. In particular, there was some difficulty ensuring that each subroutine call had a corresponding subroutine return and vice-versa. Some special cases were needed to identify spurious calls or returns that should be ignored as described below.

Due to some aspects of run-time linking, not every call results in a recorded return address. For example, position-independent code often needs to determine the memory location at which it is executing. User-mode code cannot directly access the instruction pointer register `EIP` (the program counter register on the IA-32 architecture) so position-independent code often uses a trick to retrieve the value of the instruction pointer using one of the methods show in Figure 3.2.

In code that uses this method, there is no corresponding `ret` instruction, so such "calls" are not recorded in the out-of-band return address stack.

All `call` instructions with both source and target addresses in the memory region of the runtime linker executable code are ignored because they are frequently the results of other spurious calls or returns and may be safely ignored, as described below. For similar reasons, returns with both source and target addresses in the runtime linker code are also ignored. Returns from the runtime linker into a shared library are the result of imported symbol resolution. When the imported shared library function is called for the first time, a runtime linker function is called to look up the address of the desired function. This function writes the resolved function address into the ELF Global Offset Table and "returns" into the resolved function. These return instructions have no corresponding call instruction, so they are also ignored.

Similar special cases are also needed for calls and returns involving the DynamoRIO shared libraries. When a program is linked with the DynamoRIO libraries, DynamoRIO takes over execution of the program during shared library initialization. The first basic

block placed into the cache and executed contains the return from the DynamoRIO initialization and must be ignored. Similarly, there are returns into DynamoRIO at the end of program fragments for which no matching calls exist. For these reasons, calls or returns with a source or target address in the text segment of a DynamoRIO library are ignored.

Identification of calls and returns to or from the runtime linker involves correlating the source and target addresses to mapped memory regions in the process. The current implementation reads `/proc/self/maps`[1] at initialization time and stores the address range and pathname of the object mapped into that range in a linked list. For each control transfer instruction source or target address, the address is correlated to the object mapped at that address by traversing the list. This is then used to identify whether the call or return should be ignored as previously described.

Ignoring these call and return instructions is safe because the return addresses are overwritten by either application or shared library code. The runtime linker does not deal with user input[2] so an overflow may not be induced within it. Additionally, even though program control may have gone through the runtime linker in resolving a shared library routine, no addresses within the runtime linker will be in the call chain; the shared library function returns directly to where it was called in the application code. Therefore, when an unbounded string operation occurs, only the return addresses of functions in the application or shared library may be overwritten, and these returns are all validated. Also for this reason, all the return addresses in the call chain that can be specifically targeted by exploitation of a format string vulnerability will be validated.

[1] `/proc/self/maps` is a pseudo-file on the `procfs` filesystem that contains a textual listing of the mapped memory segments of the current process.

[2] The linker's primary interaction with the user is through environment variables such as `LD_PRELOAD` which are ignored when running `setuid` root executables.

5.1.2 Conclusion

After identifying the special cases and spurious calls that should be ignored, implementation of the technique proved very straightforward. The technique and implementation successfully detect and prevent several exploitation techniques without noticeable performance degradation.

However, there are several limitations. The current implementation uses a fixed-size array for storing return addresses. This unnaturally limits the maximum call depth. A dynamically resized array could provide constant amortized accesses while allowing the maximum call depth permitted by available memory. In addition, while the technique described successfully stops all "off-the-shelf" attacks against stack return addresses, it is vulnerable to a targeted attack using vulnerabilities that allow multiple arbitrary addresses to be overwritten (these include format string and heap corruption vulnerabilities). If the out-of-band return address stack is stored in a known or predictable location, the attacker may overwrite both the return address in the stack segment and in the out-of-band stack. However, by using a dynamically resized array, the return address stack would not remain at a fixed location and would make this targeted attack infeasible.

Several approaches could be taken to improve performance. The current use of a linked list to store memory region mappings could be improved to minimize the lookup time by a variety of techniques. A simple hash table indexed by the first 22 bits of the address would speed up lookup times considerably.

5.2 System Call Sandboxing

A *sandbox* is a restricted execution environment for running potentially untrusted code. Programming language environments typically use sandboxes to ensure secure execution of mobile code. For example, the Java programming environment uses a sandbox when

executing untrusted Java Applets. The Java sandbox limits the access of untrusted code to potentially harmful class methods. A similar approach may be used to control the execution of any system process by restricting the use of specific UNIX system calls.

A system call is a request for service from the user to the operating system. The operating system kernel acts as a monitor between the user and system resources. In this way, user-mode code does not directly interact with resources such as files or devices, but instead makes system calls to request that the operating system kernel perform the desired operations. Typically, system calls provide the functionality to interact with files, devices, other processes, or the network. The system calls enforce the UNIX security model by only permitting authorized actions as determined by file and resource permissions. The actions of a process are effectively constrained by restricting which system calls may be performed, and how the allowed system calls may be used. For example, by limiting certain system calls, an untrusted process may be restricted from using the network or performing any changes to the file system even though file access permissions alone may allow it to.

On Linux/IA-32, the system call convention is the following. The system call number is placed in register `EAX` and up to six arguments are placed in registers `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, and `EBP`, in that order. If there are more than six arguments to the system call, they are stored in a structure in memory and the address of that structure is passed as the first and only argument in `EBX`. The system call is finally initiated by performing a software interrupt instruction, `int $0x80`. After this instruction, the return value of the system call is contained in register `EAX`. If there was an error executing the system call, the return value is `-1` and the global variable `errno` is set to an error code representing the error that occurred.

```
...
mov    %edx, 0x10(%esp)
mov    %ecx, 0xc(%esp)
mov    %ebx, 0x8(%esp)
mov    %eax, $0x00000005
int    $0x80
...
```

Figure 5.1: System Call Before Instrumentation

5.2.1 Implementation

The approach we take is to replace system call interrupts with calls to a monitor function that performs the system calls on behalf of the application. This monitor function traces system calls (similar to the common Unix utilities `strace(1)` and `truss(1)`) and selectively executes system calls based on examination of the specific system call requested or the arguments to the call. This mechanism is used to implement a sandboxing scheme based on restricting use of system calls.

The implementation, a DynamoRIO client, examines each basic block before execution and replaces system calls, the IA-32 instruction `int $0x80`, (Figure 5.2.1) with a sequence of instructions that save the application's execution state, call the monitoring function `chk_syscall`, restore application state, and resume execution (Figure 5.2.1). The system call monitoring function `chk_syscall` takes as arguments the requested system call number and parameters. The return value of `chk_syscall` will be interpreted as the return value of the system call by the application. For example, if the system call is denied, `chk_syscall` may set `errno` to `EPERM` (signaling that the operation was not permitted or permission was denied) and return `-1`.

An arbitrary static or dynamic sandbox policy may be implemented in `chk_syscall`. The prototype currently implements a static policy denying all use of network sockets.

```
...
mov   0x4002079c, %esp
mov   %esp, 0x400207d4
pushf
pusha
push  %ebp
push  %edi
push  %esi
push  %edx
push  %ecx
push  %ebx
push  %eax
call  $0x40023f98
mov   0x40020780, %eax
add   %esp, $0x1c
popa
popf
mov   %esp, 0x4002079c
mov   %eax, 0x40020780
...
```

Figure 5.2: System Call After Instrumentation

However, simple extensions may yield policies restricting file accesses to a specified area of the file system (simulating the `chroot` system call) or network communication to a trusted subset of hosts.

5.2.2 Conclusion

The small number of system calls in Unix operating systems make them an ideal interface to restrict for the implementation of advanced security models or policies. Using dynamic binary translation to rewrite system calls allows these policies to be safely enforced in user code. In addition, because the policy may be selected when the application is launched, different applications may be restrained by different security policies or models. Another

benefit of implementing the security policies in user-mode code is simplified development. Security policy mechanisms may be written and debugged without the added complexity of kernel-level development.

The performance overhead of system call sandboxing is primarily dependent on the complexity of the enforcement of the security policy. In the prototype implementation, the system call policy check is a simple integer comparison. Therefore, the performance penalty is minimal. More complex security policies may add more performance overhead. However, efficient implementation strategies may minimize this overhead.

Due to the design of the DynamoRIO dynamic binary translation system, this form of sandboxing is inescapable. For example, a malicious user may attempt to bypass the system call instrumentation by jumping to code that executes the interrupt instruction. In this case, the target of the jump would be treated as a new basic block and the system call instrumentation code would be inserted. However, malicious user code may attempt to modify the code in the DynamoRIO basic block and trace caches. An attack of this style is acknowledged by the DynamoRIO authors and they contend that the attack may be stopped by write-protecting the code caches and other DynamoRIO data while application code is being executed.

Chapter 6

Conclusions

We introduced a new classification of computer security vulnerabilities, *memory trespass vulnerabilities*. Memory trespass vulnerabilities are software weaknesses that allow memory accesses outside of the semantics of the programming language in which the software was written. Common memory trespass vulnerabilities include buffer overflow, format string, signed integer cast, integer overflow, out-of-bounds array access, and double-free vulnerabilities. Memory trespass vulnerabilities may be used to perform a *code injection* exploit whereby an attacker may divert program control into dynamically injected machine code.

We also introduced a decomposition of code injection exploits. A code injection exploit consists of the vulnerability, vector, payload, string, and delivery. The vulnerability, a memory trespass vulnerability, makes exploitation possible. The exploit vector is the mechanism by which control of the program is diverted to a location of the attacker's choosing. The exploit payload is the crafted machine code injected into the process. The sum of the input to the vulnerable program, including the encoded exploit vector and payload is the exploit string. Finally, the exploit delivery is the method by which the input is sent to the program. The exploit delivery may be through a network connection, local

environment variable or command-line argument. The decomposition of code injection exploits into these components facilitates the classification of defenses against these attacks.

Computer security, especially application security, is an arms race between attackers and defenders. Since the first publicly identified buffer overflow vulnerability, both attacks and the vulnerabilities they exploit have grown in sophistication. For example, the difference in sophistication between the exploitation method used by the Morris worm and the creative exploitation of the recent Apache Chunked Encoding vulnerability [9] is dramatic. The use of non-reentrant functions in signal handlers has even proved to be exploitable remotely in some cases [32]. In addition, the capability to discover vulnerabilities in applications has grown significantly through the use of advanced reverse engineering [11] and automated black-box testing [1]. Although the number of trivially exploitable memory trespass vulnerabilities in major applications is going down, the vulnerabilities that do remain are increasingly difficult to spot for both attackers and defenders. For example, the Apache Chunked Encoding vulnerability existed in the extremely popular open-source web server for over 4 years before it was disclosed. Subtle vulnerabilities even have been found in the OpenBSD operating system, whose code base undergoes regular manual source code audits for potential security vulnerabilities.

In the same span of time, much work has been done in creating defenses against code injection exploits. These defenses may target one or more of the phases of a successful code injection exploit attempt. In this work, there has been a wealth of creative solutions, many of them described in Section 3.3. However, none of these systems is perfect, and there have been many successful attacks against them ([24] [5] [31]). In order to keep up with attack technology, defense systems must be more dynamic in both their design and implementation. Systems using a variety of techniques and using diversity in their implementation may fare much better than specialized static solutions. Dynamic binary translation has proven itself to be a very viable solution for implementing dynamic defense

systems.

As shown by the two implemented approaches, dynamic binary translation can be used to implement approaches that would typically be implemented as a static compiler or operating system extension. Using dynamic binary translation to rewrite application code at run-time obviates the need for application source code or permanent changes to the application. This enables the application changes to be performed randomly at run-time, making the application a moving target. The System Call Sandboxing implementation showed how dynamic binary translation may be used to implement a system that would typically be implemented in the operating system kernel. This enables specialized systems to be built to protect applications without requiring global changes to the operating system (which may not be possible without operating system source code access). However, dynamic binary translation is not just a tool for re-implementing existing systems; its use also enables a new generation of dynamic defense systems. The investigation of the systems made possible by dynamic binary translation remains an area of promising future research.

6.1 Future Work

Dynamic binary translation is a useful tool for several areas of future research. One area of potential research is techniques of introducing dynamic diversity into executables. Building upon earlier work describing diversified compilation [12], dynamic binary translation can enable diversification to be performed at run-time. Using static diversification successfully stops mass exploitation of vulnerabilities, but a targeted attack by a knowledgeable attacker may succeed. Diversifying a program at run-time would make it difficult for a determined attacker to tune their attack when the parameters needed to successfully exploit the vulnerability will differ with every execution of the program. Dynamic diversification will also have the same benefits of static diversification.

Another interesting area of future research is in adaptive and self-regenerative systems. A long-running server application under dynamic binary translation may initially run with code transformations intended to detect attempted attacks. When the system determines that it is under attack, the system may enable more secure, albeit expensive, code transformations until the attack subsides.

References

[1] Dave Aitel. Spike. http://www.immunitysec.com/spike.html.

[2] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño - a compiler-supported java virtual machine for servers.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Impleme ntation*, pages 1–12, 2000.

[4] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.

[5] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[6] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin a nd T. Tye, S. Yadavalli, and J. Yates. Fx!32 a profile-directed binary translator, 1998.

[7] Tool Interface Standards Committee. Executable and linking format. http://x86.ddj.com/ftp/manuals/tools/elf.pdf, 1995.

[8] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zang. Automatic detection and prevention of buffer-overflow attacks. *7th USENIX Security Symposium*, 1998.

[9] Mark J. Cox. Apache httpd: vulnerability with chunked encoding. http://online.securityfocus.com/archive/1/277268.

[10] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[11] Halvar Flake. Graph-based binary analysis. In *Blackhat Briefings 2002*.

[12] Stephanie Forrest, Anil Somayaji, and David. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[13] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, 2002.

[14] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, pages 177–190, 2001.

[15] Sun Microsystems. proc - /proc, the process file system. In *SunOS 5.8 Manual*, chapter 4.

[16] Sun Microsystems. ptrace - allows a parent process to control the execution of a child process. In *SunOS 5.8 Manual*, chapter 2.

[17] Sun Microsystems. The Java Hotspot performance engine architecture, 1999.

[18] David Moore. The spread of the code-red worm. http://www.caida.org/analysis/security/code-red/coderedv2_ana lysis.xml.

[19] Robert Morris. Morris worm source code. http://sunsite.bilkent.edu.tr/pub/security/cerias/doc/morris_worm/worm-src.tar.gz.

[20] Tim Newsham. Format string attacks. Technical report, Guardent, Inc., September 2000.

[21] Last Stage of Delirium Research Group. Unix assembly codes development for vulnerability illustration purposes. http://www.lsd-pl.net/documents/asmcodes-1.0.2.pdf.

[22] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[23] Trek Palmer, Dino Dai Zovi, and Darko Stefanovic. SIND: A framework for binary translation. Technical Report TR-CS-2001-38, University of New Mexico, 2001.

[24] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. http://www.corest.com/files/files/11/StackguardPaper.pdf, June 2002.

[25] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–220.

[26] Anil Somayaji and Stephanie Forrest. Automated response using System-Call delays. pages 185–198.

[27] E. Spafford. The internet worm: Crisis and aftermath, 1989.

[28] PaX Team. Nonexecutable data pages. http://pageexec.virtualave.net/pageexec.txt.

[29] Vendicator. Stack shield. http://www.angelfire.com/~sk/stackshield/info.html.

[30] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[31] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[32] Michal Zalewski. Delivering signals for fun and profit. http://razor.bindview.com/publish/papers/signals.txt, 2001.