Install-Time Vaccination of Windows Executables to Defend against Stack Smashing Attacks

Danny Nebenzahl, Mooly Sagiv, and Avishai Wool, *Senior Member*, *IEEE*

Abstract—Stack smashing is still one of the most popular techniques for computer system attack. In this work, we present an anti-stack-smashing defense technique for Microsoft Windows systems. Our approach works at install-time, and does not rely on having access to the source-code: The user decides when and which executables to vaccinate. Our technique consists of instrumenting a given executable with a mechanism to detect stack smashing attacks. We developed a prototype implementing our technique and verified that it successfully defends against actual exploit code. We then extended our prototype to vaccinate DLLs, multithreaded applications, and DLLs used by multithreaded applications, which present significant additional complications. We present promising performance results measured on SPEC2000 benchmarks: Vaccinated executables were no more than 8 percent slower than their un-vaccinated originals.

Index Terms—Computer security, buffer overflow, instrumentation.

1 Introduction

1.1 Background

Stack smashing attacks, which exploit buffer overflow vulnerabilities to take control over attacked hosts, are the most widely exploited type of vulnerability. About half of the CERT advisories in the past few years have been devoted to vulnerabilities of this type [9]. Stack smashing is an old technique, dating back to the late 1980's. For example, the Internet worm [44], [20] used stack smashing. However, this technique is still in current use by hackers: For instance, it is the underlying method of attack used by the MSBlast virus [10], [11]. Stack smashing attacks are not even unique to general purpose OSes like Unix or Windows: Successful attacks were reported against specialized operating systems and hardware platforms such as Cisco's IOS [12]. In general, stack smashing works against a program that has a buffer overflow bug: A malicious attacker inputs a string that is too long for the buffer, thereby overwriting the program's stack. Since the program keeps return addresses on the stack, the overwriting string can modify a return address—and when the function returns, the attacker's injected code gets control. A detailed description of the stack smashing attack mechanism can be found in [2], [15].

1.2 Classification of Anti-Stack-Smashing Techniques

Various techniques have been developed to defend against stack smashing attacks. One way to classify these techniques is by the method they use to handle the vulnerability:

- Techniques ensuring that software vulnerabilities exploitable by stack smashing attacks do not exist, i.e., they attempt to eradicate buffer overflows.
- Techniques that prevent an attacker from executing malicious code on the attacked host. These techniques assume that buffer overflows will continue to occur, and attempt to ensure that the attack code will not be executed successfully.

Our tool is of the latter type. It does not detect buffer overflows, but defends against their exploitation.

A second classification of anti-stack-smashing techniques is based on the stage in the software life cycle in which the countermeasures are deployed:

- Techniques that are deployed by the software *developer* at the software coding stage. These techniques include static code analysis and modified compilers.
- Techniques that are deployed by the software *user*, before, or while, using the vulnerable software. These techniques include wrappers, emulators, and binary code manipulations.

Our tool is a user tool: It does not require access to the source code.

Note that anti-stack-smashing developer tools (static checkers, compilers) have the advantage of working at a high level of abstraction, e.g., with access to the C source code. In contrast, user tools have little or no information about the language or techniques used to create the program—all they have to work with is the binary executable. However, we argue that user tools are extremely valuable: Typically, the user has no control over the bugs in the software. Thus, having the ability to *vaccinate* software, at the user site, at the user's discretion, is an important goal.

The vast majority of anti-stack-smashing tools are Unix-based. This is because source code is readily available for

- *D. Nebenzahl and M. Sagiv are with the Department of Computer Science, Tel Aviv University, Ramat Aviv 69978, Israel. E-mail: d.nebenzahl@gmail.com, msagiv@post.tau.ac.il.*
- *A. Wool is with the School of Electrical Engineering, Tel Aviv University, Ramat Aviv 69978, Israel. E-mail: yash@acm.org.*

Manuscript received 10 Nov. 2004; revised 7 Oct. 2005; accepted 9 Nov. 2005; published online 3 Feb. 2006.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0160-1104.

the operating system, the compilers, and application software. In contrast, our tool targets the proprietary Microsoft Windows operating systems.

The work closest to ours was independently suggested by Prasad and Chiueh [38], in parallel with early versions of our work [36], [37]. They too add anti-stack-smashing instrumentation into Windows binary files. However, they only handle very simple Win32 executables. Our work shows that advanced features like multithreading, DLLs (Dynamic Link Libraries), and their combination, significantly complicate the problem. Dealing with such features requires new and different methods. A more detailed comparison of our work and that of [38] can be found in Section 9.

1.3 Contributions

The contribution of our work is twofold. Our first contribution is that we created an anti-stack-smashing tool that works at *install time*, or whenever the software user wishes. Thus, our technique does not require access to the source code of the application and assumes nothing about the application, beyond it being written in a high-level compiled language. The main idea is to equip existing binary files with additional machine code that can detect a stack smashing attack.

The second contribution is that our tool is a "system-wide" solution. Our tool handles simple applications, multithreaded applications, and DLLs. Thus, the user can instrument a vulnerable binary while keeping its interoperability: A DLL can be instrumented while the applications using it are not instrumented, and an application can be instrumented with or without instrumenting DLLs it uses.

We have built a working prototype implementing our approach, that instruments Win32 executables running on an x86 Intel Pentium platform. We vaccinated several Windows executables with known buffer overflows, and successfully defended them against real exploits. Our approach enjoys minimal overhead: In standard benchmarks, we have not observed more than an 8 percent slowdown in the vaccinated program.

An extended abstract of this work appears in [37].

Organization: In Section2, we describe our solution architecture. Section 3 describes the implementation of our technique to vaccinate simple Windows applications. Section 4 describes the techniques used to vaccinate Windows DLLs. Section 5 describes the techniques used to overcome the challenges imposed by multithreaded applications. Section 6 describes the techniques used to vaccinate DLL's used by multithreaded applications. In Section 7, we evaluate our solution. In Section 8, we provide a proof sketch for the correctness of our technique. In Section 9, we describe alternative approaches to stack-smashing protection, and we conclude in Section 10.

2 Solution Architecture

2.1 Design Choices

Our first choice was to use a separate stack (as done by LibVerify [2] on Linux), rather than insert so-called "canaries" into the stack, as done by StackGuard [15]. We believe that a separate stack offers better protection than canaries—e.g., [50] shows how an attacker can overcome a simple canary-based mechanism. Furthermore, the separate stack approach can be modified to support any mechanism that requires additional memory to be allocated and used in runtime (such as encrypted per-thread canaries).

We chose to insert all the instrumentation code into the executable file itself, rather than to rely on load time instrumentation code (see [22]). This is a somewhat arbitrary choice, and we believe that load-time instrumentation is a viable option.

Furthermore, we chose not to use the Detours library [22], and to implement our own instrumentation mechanism. This gave us the ability to instrument functions that Detours skips (such as short functions, and functions with jumps into entry or exit code).

We implemented our approach using static instrumentation: Our vaccination tool instruments the executable file when it is not used, not its image in memory during runtime. It should be noted that while this choice limits our solution to "normal" (i.e., non-self-modifying) programs, it results in better performance than dynamic instrumentation.

Finally, we do not assume the presence of any debug symbols, map files, or any information beyond what is available in the raw Windows binary file—simply because such information is typically only available at compile time and rarely available to users at install time.

2.2 The Basic Method

Our anti-stack-smashing mechanism is based on instrumenting existing software. The instrumentation code is added at the function level—each function is instrumented with additional entry and exit code. The added entry code records the return address from the stack by pushing it onto a private stack. The added exit code tests whether the return address found on the stack just before returning from the function is identical to the one at the top of the private stack, the one that has been recorded upon the function entry.

If our instrumentation detects that the stack has been smashed, i.e., the return address has been overwritten, we halt the program by using a deliberate illegal memory access. Halting the program is not the only possible option. Since we have the original return address in the private stack, we could return to the correct caller of the function. However, we believe this to be an inferior choice because continuing to run after detecting that the stack is corrupt will result in unpredicted behavior. Implementing a messaging mechanism to inform the user about the attack, or to log data about the attack, can also be dangerous, as noted in [40].

2.3 Instrumentation of Binaries

The process of instrumenting a binary consists of the following steps:

1. binary disassembly,
2. function discovery,
3. function analysis and classification, and
4. function modification (the actual instrumentation).

A diagram of the instrumentation process is shown in Fig. 1. In the next sections, we describe each of these steps in some detail.

2.4 Disassembly

Since the disassembly process is not in the core of our research, we chose to use a commercial disassembly

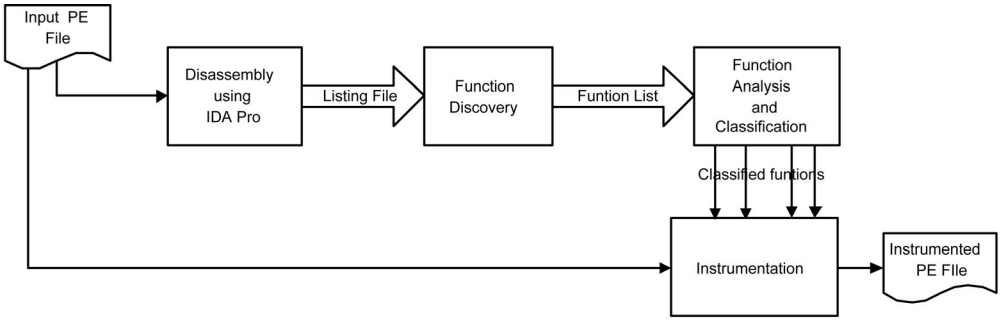

Fig. 1. The PE instrumentation process.

package, the IDA Pro [24]. We chose IDA Pro since it has been recognized as a very accurate disassembler that is capable of distinguishing between code and data [8], [31]. Our own experience showed IDA Pro to be more accurate than a number of shareware and free open-source disassemblers (e.g., [39], [13]). The input to IDA Pro is the binary to be disassembled and the output is a listing file of the disassembled program.

2.5 Function Discovery

The next step in our process is the discovery of function boundaries. To do this, we wrote a parser for the IDA Pro listing file. We identify a function entry when we find an address that is called, using the CALL machine instruction, from some other address. Thus, the listing file is scanned to detect calls, and the called addresses are marked as function entry addresses. Each function is then scanned, building a tree emulating all possible branches in the function, until all RET commands (exit addresses) of the function are detected. Note that a function can have more than one entry point, and more than one exit point. Note also that our function discovery will miss "nonstandard" functions—e.g., functions that are not called by the CALL instruction, or that do not return by the RET instruction. We believe that this is not a significant issue since compilers, in order to create high-performance binaries, generate standard call sequences. Nevertheless, our method can be extended to detect and handle other function call and return mechanisms.

2.6 Function Analysis and Classification

When instrumenting a function, it seems attractive to add additional entry-code before the entry address of the function, and additional exit-code after the end of the function. Unfortunately, in general, this method cannot work. One cannot assume that the addresses before or after a function are not used. Inserting code between functions is also a complicated solution since it may require the modification of all memory references in the binary. Therefore, our solution is to instrument a function by overwriting the entry-code of the function with a jump instruction to an area that is not used in the binary. The jump-to area includes:

- our instrumentation code,
- the original entry code instructions that were overwritten by our jump instruction, and
- a jump back to the rest of the original function.

A similar solution is done for the instrumenting of the function's exit-code. This instrumentation method is shown in Fig. 2.

However, in a CISC architecture, implementing this solution is not so simple. In a CISC architecture, different instructions may have different lengths. Replacing the original entry (or exit) code with a jump may replace several instructions of the original code with the one instruction of the added jump. Furthermore, the original program may include jumps to one of the instructions replaced by the added jump. Therefore, simply replacing the original code with a jump may result in an erroneous program that includes jumps to illegal instructions, as shown in Fig. 3. In order to avoid this problem, we classify functions into three categories:

- Simple functions. Simple functions have one entry point, one exit point, and do not include jumps into the first or last instructions of the function. These functions are handled as mentioned above.
- Complicated functions. These functions may have more than one entry or exit point, and may have jumps into entry code or jumps into exit code. To instrument complicated functions, we copy them in full to an unused area. Their entry code areas are replaced with jumps to the new, instrumented copy of the whole function. If the function has more than one entry point, multiple instances of the function will be created, one for each entry point. Each copy is instrumented to handle a call sequence that enters through *its* entry point.
- Unhandled functions. Functions that include indirect or computed jumps (jumps whose destination address is determined by a value in a register or by data). The difficulty with such jumps is that, a priori, the jump destination is not known and, thus, determining the function's boundaries with certainty is impossible. Compilers use indirect jumps to efficiently implement C `switch` statements as jump tables. Although discovering jump tables in binary files is feasible, and done quite well by IDA-Pro [8], in our prototype, we decided to not instrument such functions. In a real-world product, one should of course implement jump table discovery methods, such as the ones mentioned in [8], [29]. In all of the programs we have instrumented, indirect jumps were used in less than 4 percent of the functions.

2.7 Updating the Binary

After detecting and classifying the binary's functions, the binary is instrumented. The PE file is modified to include the bigger address space needed for the instrumentation code, the new entry and exit codes of each function are

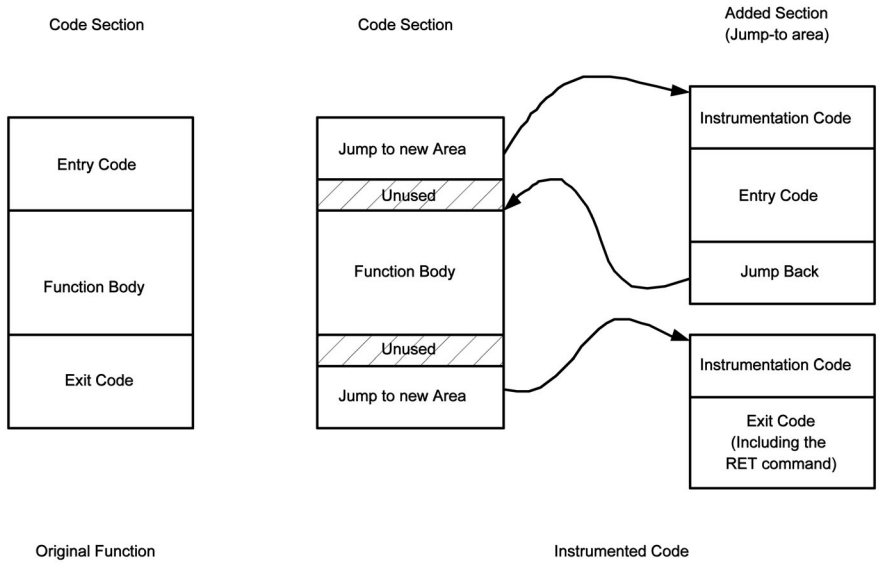

Fig. 2. Instrumentation of simple functions.

added, and the entry point of the binary is changed, so that the binary's first instructions will consist of the initialization of the instrumenting code (initializing the private stack in which copies of return addresses are saved). Methods for adding code to a Win32 binary are described in detail in [6]. In general, we either extend the last section of the binary, or add a new section to the binary. These methods allow for adding significant and predetermined amounts of code to an existing binary.

3 Instrumenting a Simple Win32 Application

In our terminology, a *simple Win32 application* is a windows application that is not multithreaded. Examples of such applications are Notepad, RegEdt32, Calc, etc. These applications are loaded into fixed virtual memory addresses, and the memory allocated to them is used solely by them. As mentioned above, our instrumentation code manages a private stack. In simple Win32 applications, we allocate the memory used for this stack, statically, at the end of the original binary. Thus, the address of the private stack is known at instrumentation time, and the instrumentation code can directly access the private stack. The initialization code added at the beginning of the program initializes the private stack, and the instrumentation code of each function manages the private stack.

In our prototype implementation, the jump instruction added to each function's entry or exit code takes 5 bytes, and the added instrumentation code takes 29 bytes for each entry point and 41 bytes for each exit point of the function. We use a private return address stack of 768 bytes, which allows a function nesting of depth 192.

To demonstrate the effectiveness of our instrumentation, we instrumented the RegEdt32 application. This application has a known buffer overflow [5] for which exploit code is available. We first verified that the exploit indeed successfully attacks the application. Then, we instrumented the application and checked that the instrumented application still worked correctly. Finally, we verified that the exploit caused the instrumented application to halt, as described in Section 2. We also instrumented other simple Win32 applications (such as Notepad.exe, WinHlp32.exe, and Calc.exe), against which we did not have exploit code. We verified that all these applications worked correctly after vaccination, to demonstrate that our instrumentation does no harm.

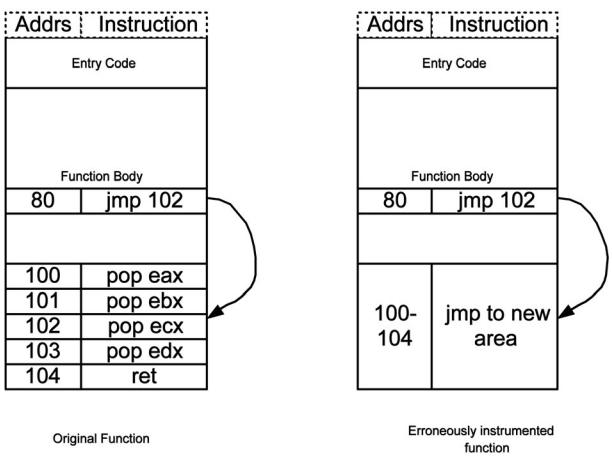

Fig. 3. Erroneous function instrumentation in a CISC architecture. The original 5-byte, 5-instruction exit code was replaced by a single 5 byte jump instruction. The jump from address 80 now lands in the middle of an instruction.

4 Instrumenting a DLL

DLLs are PE files that contain function libraries that can be used by multiple applications simultaneously.

The use of DLLs is extremely common in the Microsoft windows family of operating systems, e.g., there are more than 1,000 DLLs in a typical Win2000 C:\WINNT\SYSTEM32 directory.

A DLL normally specifies a "preferred" virtual address in which it should be loaded. However, to handle situations

in which the address space is already used by an application or another DLL, the Win32 DLL loading mechanism supports relocation—the process of moving or copying a DLL to another address space. The relocation mechanism is supported by both the compiler and the PE file structure: The compiler creates a relocation table, which is part of the PE file. The relocation table is a list of all the addresses in the binary that need to be updated upon relocation.

Instrumenting a DLL imposes two main problems: allocating memory for the private stack and handling DLL relocation. Since a DLL can be used by more than one process, in order to prevent race conditions, memory for our private stack should be allocated per process. Thus, the address of our private stack needs to be determined at runtime, as new processes access the DLL. Handling relocation means that either our instrumentation code must not use direct addressing, or we need to update the relocation table so our instrumentation code will relocate correctly.

We suggest a method to handle both problems simultaneously, based on an operating system paging policy named Copy-on-Write. The paging mechanisms of Win32 lets physical memory be shared by many processes. For example, multiple instances of the same program may use a single copy of code, thus saving memory. In order to correctly handle this sharing, the operating system must handle the situation in which some process updates this shared memory space (for example, a process updates its code). In such a case, only the updating process should see the updated copy. The other processes in memory must remain untouched. Handling this situation is done by using a paging policy of Copy-on-Write. When a process updates a physical page shared with other processes, the page is first copied to a new physical location, and only the copy in the new location is updated. From this moment on, two copies of the original physical page exist, and the modified copy is viewed only by the modifying process.

To use this feature, we made the instrumented DLL into a self-modifying and self-relocating DLL. Upon loading or being attached to a process, the initialization code that we add at the entry point of the DLL determines where it is in memory, and updates the instrumented code so all references to the private stack will be to the correct addresses (thus handling relocation). This action, of rewriting part of the DLL in memory, causes the operating system to duplicate the rewritten pages. The rewritten pages contain the private stack and our instrumentation code (notice that only our instrumentation code accesses our private stack). Thus, by making the instrumentation self-relocating, and hence self-modifying, the operating system gives us for free a separate memory block to store the private stack for each process.

However, our implementation did not completely escape the need to update the relocation table. Since we handle complicated functions by duplicating them, we must update the relocation table so the instructions in the copied functions (e.g., an access to a global variable) relocate correctly.

We implemented and tested a prototype utilizing this method. We first wrote a vulnerable DLL and our own exploit code, and checked that the exploit smashes the stack. Then, we instrumented the DLL, and checked that it works correctly in a multiprocess environment by deliberately causing race conditions between multiple processes that use the DLL. We tested our defense mechanism by causing a buffer overflow in the DLL. Our instrumentation code did indeed catch the stack-smashing and halt. We have not tested our defense successfully against a real exploit of a real DLL because we were unable to find a suitable DLL with available exploit code (despite many days of frustrating attempts). Section 7.3 describes one of the more interesting failures.

5 Instrumenting a Multithreaded Application

Modern operating systems allow for multithreading: Multitasking within a process. This creates a problem similar to the one imposed by DLLs, i.e., the instrumentation code needs to allocate per process memory for the private stack. In order to prevent race conditions between threads in a multithreaded application, there is a need for per *thread* private stack memory allocation.

In a multithreaded environment, the process's memory is shared between threads. In contrast to multiprocess operation, in a multithreaded application, all the threads are allowed to change memory regions owned by the process, and it is the application's responsibility to ensure its correct behavior—with little or no help from the OS. Thus, the Copy-on-Write trick we relied on for solving the per-process memory allocation problem (recall Section 4) is not applicable for the multithreaded scenario: Memory that is shared by multiple threads of the same application is not considered to be "shared" as far as the operating system is concerned.

Instead, we use another feature of the Windows operating system. Win32 has a memory allocation mechanism that is capable of allocating memory per thread, called Thread Local Storage (TLS). This mechanism facilitates defining memory structures (variables) that are allocated per thread. TLS allows the programmer to write simple code such as referencing a variable, and each thread will automatically access a different copy of the variable. Allocation of such memory can be defined in the PE file by adding a special section (the .tls section) that describes the per-thread allocation and initialization functions. Accessing a TLS variable is a much more complicated task than accessing a standard variable, since at compile time, the address of the variable is not known. Instead, when a programmer defines a TLS variable (using a special C language extension), another hidden variable is created. This added variable is set by the operating system to a value called the *tls index*. When accessing a TLS variable, the hidden variable is accessed in order to retrieve the tls index. The thread descriptor (a variable maintained by the operating system storing information about each thread) is accessed in order to retrieve the address of a table of pointers. This table is accessed using the tls index to retrieve a pointer to the actual TLS variable. This runtime access sequence is shown in Fig. 4.

We solve the need for per-thread private stack memory allocation by defining the private stack as a TLS variable. The allocation is done by adding a .tls section to the executable file. The .tls section describes the private stack and its initialization function. The instrumentation code accesses the private stack using a sequence defined by the operating system as described in [33]. Note that accessing a TLS variable has a performance penalty greater than that of

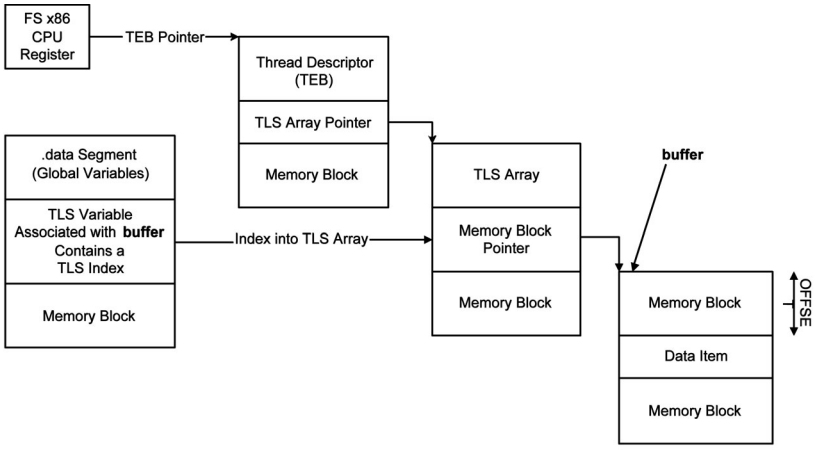

Fig. 4. Accessing the TLS variable *void *buffer* at offset *OFFSET*.

simply accessing a variable because the code needs to reference the operating system structures describing the thread. This change causes our entry code to grow to 39 bytes (instead of 29) and our exit code to grow to 47 bytes (instead of 41). We implemented and tested a prototype utilizing this instrumentation method. We wrote a vulnerable multithreaded program and verified that overflowing a buffer causes stack smashing. Then, we instrumented the program and verified that it still works correctly—including under race conditions between threads. Then, we attacked the program by causing a buffer overflow and confirmed that the instrumented code detected the stack smashing and halted.

Next, we instrumented the multithreaded application Hotfoon, which has a known buffer overflow [3] with a "proof of concept" exploit code. The instrumented Hotfoon worked normally. Unfortunately, the available exploit code causes Hotfoon to crash before it executes the RET instruction in the function whose stack was smashed, and the same behavior was exhibited by the vaccinated program since it halts before reaching our instrumenting code. We did not correct the exploit code to perform a successful stack smashing attack.

To verify that our instrumentation technique can be successfully applied to programs, without prior knowledge if these programs are multithreaded or not, we used our multithreaded instrumentation tool to vaccinate several large and complex applications, including Windows Explorer, Microsoft Internet Explorer, and more. The instrumented programs functioned correctly.

6 Instrumenting DLLs Used by Multithreaded Applications

Another complication of multithreaded applications is the handling of DLLs used by multithreaded applications. The technique used to vaccinate multithreaded applications cannot be used to vaccinate DLLs used by multithreaded applications. The reason is that TLS variables added to a PE file cannot be used in DLLs because a DLL may be dynamically linked to a process that is running. Thus, at the moment of attachment, TLS variables for all existing threads must be allocated and initialized simultaneously. This is not possible for any initialization function, nor is it supported by Win32. Since neither the software end user, nor the software developer knows whether a DLL will be used by a multithreaded application, a general purpose instrumentation method must address DLLs that are used by multithreaded applications. In other words, the simpler mechanism we used in Sections 4 and 5 are not sufficient.

We solve the need for private stack memory allocation per-stack by using a Win32 mechanism for dynamically allocating TLS variables. This mechanism provides the programmer with the following API:

- TlsAlloc—a function for allocating a TLS index. After allocating this index, each thread will be allocated an uninitialized pointer.
- TlsSetValue—a function that allows a thread to set the value of the pointer allocated for a TLS variable. The programmer allocates a memory block for the TLS variable, and sets the pointer to it using this function.
- TlsGetValue—a function that allows the programmer to retrieve the pointer to the TLS data.

The use of this API is as follows: At the initialization of a DLL, one allocates a TLS index, and saves that index in a variable local to the DLL. When a process or a thread attaches the DLL, the DLL allocates the memory needed for the actual TLS variable, and uses the TlsSetValue to store a pointer to the memory allocated. When the DLL is to access the TLS variable, it first calls the TlsGetValue API function to retrieve a pointer to the thread's copy of the variable, and accesses that variable.

We use this TLS allocation and management scheme in order to allocate and access the private stack. In order to use this scheme, we needed to instrument the DLL in a way that the memory allocation functions would be called when the DLL is loaded and when processes attach the DLL. We found a very simple way to implement this functionality. We wrote a DLL that implements only the TLS and memory allocation functions and the private stack management functions. We add the stack management functions of this DLL to the Import Directory of the instrumented DLL—a Data Directory describing the DLLs used by an application or a DLL. This causes the operating system to call our DLL's initialization and memory allocation functions when the

TABLE 1
Runtime Performance Measurements of Standard and Instrumented SPEC2000 Applications

Benchmark	Original Run Time (sec)	Instrumented Run Time (sec)	Performance Penalty (%)
bzip2	67.6	70.5	4.33
gzip	15.6	16.3	4.36
mcf	435.62	446.5	7.09

TABLE 2
PE File Size Performance Measurements of SPEC2000 and Instrumented SPEC2000 Applications

Benchmark	Original File Size(KB)	Instrumented File Size (KB)	Increase (%)	IDA-Pro Listing File Size (KB)	Instrumentation Time (sec)
bzip2	169	208	20.6	11,784	5.6
gzip	113	154	36.9	39,527	12.8
mcf	77	99	28.6	3795	1.2

instrumented DLL is loaded or when a process or a thread attaches to it.

We enhanced our prototype to use our private stack allocation and management DLL. We developed a vulnerable DLL and a multithreaded application that uses that DLL. We tested that threads may exploit the DLL's vulnerability. Then, we instrumented the DLL and tested the correct operation of the multithreaded application under race conditions verifying the correct operation of the per-thread private stack. Finally, we caused the multi threaded application to exploit the vulnerability of the DLL. Our stack-smashing detection code detected the exploit.

7 Evaluation

7.1 Performance

Instrumenting an application, especially by adding code to all program functions, results in some performance degradation. This performance degradation is caused by several factors:

- the added code that needs to be run,
- the larger address space that may change the paging performance for the program, and
- the change of memory locations in the program that results in a change of the cache performance.

Since the performance penalty is a result of multiple program and system parameters, we decided to evaluate the performance of whole instrumented programs rather than measure microbenchmarks. To do this, we instrumented several SPECINT applications from the SPEC2000 suite [45] using the instrumentation method for handling multi threaded applications (even though all the applications were simple Win32 applications). This evaluation results in a worst-case measurement:

- The performance penalty of the multithreaded instrumentation code is the highest due to the TLS variable accessing sequence.
- The SPEC2000 applications are, in general, number-crunching applications, designed to benchmark compilers and CPUs. These application are much more demanding than the usual, interactive applications used by end-users of the Win32 environment.

We ran the original and instrumented programs on a 2GHz Pentium 4 with 256MB RAM and measured their average runtime. We verified that the instrumented code produced the same output as the standard uninstrumented code. The measured performance penalty was less than 8 percent for the SPEC2000 applications we measured (see Tables 1 and 2 for details), and the increase in program size was 20-37 percent. We consider these results very positive: An 8 percent slowdown most likely will not be noticeable in an interactive program. Table 2 also shows the size of the intermediate IDA-Pro listing file, which can grow quite large, and the instrumentation time, which is roughly proportional to the size of listing file. Note that our focus was on demonstrating a working prototype, so we did not make any effort to reduce the instrumentation time. We believe that the instrumenting tool can be greatly optimized through the use of better data structures.

7.2 Limitations of the Approach

We can identify limitations caused by the fact that we instrument a binary executable file, rather than its loaded image.

1. Known Unhandled Functions: As noted in Section 2.6, we do not instrument functions that use indirect jump instructions. Thus, our tool does not defend against attacks that exploit vulnerabilities in these functions.
2. Unknown Unhandled Functions: The function discovery process may miss functions, for example, functions called by indirect calls such as function call tables. Vulnerabilities in such functions are not defended by our defense mechanism. In some cases, such as when using nonstandard calling sequences, the instrumentation may even prevent correct operation of the executable.
3. Modification of the Original Binary: The fact that we modify the executable file prevents preserving the executable manufacturer's signature and liability, and requires the user's special attention when using executable's integrity tools such as personal firewalls or filesystem checksums.

The first limitation can be addressed by introducing techniques of jump table discovery, which will reduce the number of unhandled functions. The second limitation can be handled at runtime, for example, by checking that the

destination of an indirect call is indeed instrumented. Both limitations can be handled if more information known at compile time will be available; such a suggestion, in the context of security instrumentation of binaries, is given in [18]. These improvements are left for future work. A more detailed discussion of the limitations is given in Section 8.

7.3 Instrumentation of System DLLs

During our research, the Blaster worm [11] attacked thousands of computers worldwide. Blaster was based on a buffer overflow in a Windows system DLL [4]. Therefore, we attempted to vaccinate this DLL to demonstrate that our methods can stop the worm. Unfortunately, we were unsuccessful: Despite our best efforts, the computer would freeze when it loaded the instrumented DLL during boot. Debugging kernel level system services requires specialized tools that were unavailable to us, so we have not been able to determine (and fix) the precise problem. We suspect that such system DLLs may use nonstandard calling sequences, e.g., calling the operating system dispatcher function, which never returns control, or using a jump table or the interrupt control vector to issue system calls. As discussed above, instrumenting a program with nonstandard calling sequenced may result in crashing the instrumented executable.

8 A Correctness Proof Sketch

As discussed in Sections 2.6 and 7.2, we do not instrument all the binary's functions and, thus, cannot promise full protection against all stack-smashing attacks. In this section, we define the limitations of the instrumentation, and provide a proof sketch that our instrumentation does not change the semantics of the original program, while catching all stack-smashing attacks. Specifically, we prove that subject to these limitations, 1) the original and the instrumented program are equivalent and 2) every stack-smashing attack on the original program is detected by the instrumented program.

The equivalence of programs is proved by showing the equivalence of the program's functions. The equivalence of functions is proved by showing the equivalence of blocks of the function's commands. Two command blocks are defined to be observational equivalent (as illustrated in Fig. 5) if running each command block on any state will result in equivalent states. The proof method is an implementation of operational small step semantics.

Our operational semantics model does not take into account timing considerations. In particular, we assume that the program behavior is not sensitive to the performance penalty of the instrumentation. Note that in a real-time system, this assumption might not hold.

8.1 Definitions

In this section, we define our operational semantics model. We begin with defining the program's state, and how it is affected by the program, and proceed to define functions, programs, and the program equivalence.

Definition 8.1. *A state S is a six-tuple*

$$(PC, Reg, \sigma, pstack, RAL, above\_stack),$$

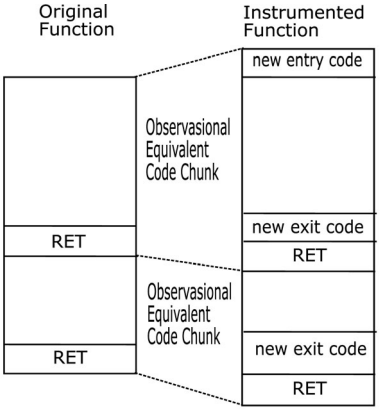

Fig. 5. Functions are proved to be equivalent by proving the equivalence of code chunks of the functions. The term Observational Equivalence denotes equivalence in all but private stack behavior.

where

- *PC describes the current program counter,*
- *$Reg : \{1, \ldots, m\} \rightarrow Val$ describes the content of machine registers,*
- *$\sigma : Loc \rightarrow Val$ maps locations to their values,*
- *$pstack : Loc \rightarrow Val$ maps locations of the private stack to their values,*
- *$RAL : Loc \rightarrow Val$ describes values of locations (on the stack) that are temporarily dedicated to return addresses, and*
- *$above\_stack : Loc \rightarrow Val$ describes locations in the unused area of the stack.*

S_0 denotes the initial state of the program. Note that a location can belong to σ at one state, and to RAL or $above\_stack$ at another.

Definition 8.2. *An instruction i is a machine instruction, including an op-code and specific parameters (e.g. `'INC eax', 'MOVE eax, 0x12345678'`).*

Definition 8.3. *A command c is an instruction in a certain context.*

A command can be viewed as a pair $(context, instruction)$, giving each command a context and making it unique in a program. The common meaning of a context is the address of the command. Our definition of a context is wider. A context of a command is its relative position in a function. This definition allows us to compare functions or function's code chunks of different programs, even if functions were relocated.

Definition 8.4. *A function f is a pair (c, CS), where command c is called (by a CALL instruction somewhere in the program) in order to activate the function (it is the function's entry point), and CS is a set of all the commands of the function.*

Definition 8.5. *A program prg is a pair (f, FS), where f is the function that is called in order to activate the program (the program's entry point) and FS is a set of functions. All initialized data is represented as part of the program's state.*

Definition 8.6. *The derivation* $< c, S > \rightarrow < c', S' >$ *or*

$$< c, (PC, Reg, \sigma, pstack, RAL, above\_stack) > \rightarrow$$
$$< c1, (PC', Reg', \sigma', pstack', RAL', above\_stack')$$

denotes that running the single command c *from state* S *results in a state* S'*, and the next command to be executed is* $c1$*. A CALL instruction is regarded as an atomic command, thus if* c *is a CALL command, then* c' *will be the next command after the return from the function that has been activated by the CALL instruction.*

Note that the instructions c and $c1$ may not be in the same functions. This specifically may happen under a stack smashing attack.

Derivations are deduced using derivation rules. These rules define the semantics of the instructions of the binary. To complete the proof, it is required to list all the derivation rules that are relevant in the 80×86 environment. For example, one rule would be: if $c = (context, \text{'INC eax'})$, and $S = (PC, Reg, \sigma, pstack, RAL, above\_stack)$, and $Reg[eax] = n$ for some integer $n < 2^{32}$, and the next command is some command $c1$, then $< c, S > \rightarrow < c1, S1 >$, where $S1 = (PC + 1, Reg', \sigma, pstack, RAL, above\_stack)$ and $Reg'[eax] = n + 1$ if $n < 2^{32} - 1$ and 0 otherwise.

Definition 8.7. $< c, S > -k \rightarrow < c1, S1 >$ *denotes a k-step derivation, meaning that running* k *commands from command* c *and state* S *results in a state* $S1$ *where the next command to be executed is* $c1$*.*

Definition 8.8. *Two states,*

$$S1 = (PC1, Reg1, \sigma1, pstack1, RAL1, above\_stack1)$$

and $S2 = (PC2, Reg2, \sigma2, pstack2, RAL2, above\_stack2)$*, are called* observationally equivalent*, denoted by* $S1 \equiv S2$*, if* $Reg1 = Reg2$ *and* $\sigma1 = \sigma2$*.*

All the following definitions of equivalence rely on observational equivalence of state.

Definition 8.9. *Equivalence of commands: Two commands* $c1$*,* $c2$ *(of different programs) are* equivalent*, denoted* $c1 \equiv c2$*, if one of the following occurs:*

- *If* $c1.instruction = c2.instruction =$ `RET` *command.*
- *For any states* $S1, S2$ *such that* $S1 \equiv S2$*, if* $(< c1, S1 > \rightarrow < c1', S1' >$ *and* $< c2, S2, > \rightarrow < c2', S2' >)$*, then* $(S1' \equiv S2'$ *and* $c1' \equiv c2')$*.*

Note that the first definition limits the equivalence to the function's boundaries.

Definition 8.10. *Two commands* $c1$*,* $c2$ *(of different programs) are* k-equivalent*, denoted by* $c1 = k = c2$*, if for any states* $S1$*,* $S2$ *such that* $S1 \equiv S2$*, if* $(< c1, S1 > -k \rightarrow < c1', S1' >$ *and* $< c2, S2 > -k \rightarrow < c2', S2' >)$*, then* $S1' \equiv S2'$ *and* $c1' \equiv c2'$*.* k-equivalence *lets us decide equivalence of chunks of code of length* k*.*

Definition 8.11. *Two functions* $f1 = (c1, CS1)$*,* $f2 = (c2, CS2)$ *are equivalent if for any states* $S1$*,* $S2$ *such that* $S1 \equiv S2$ *there exist two constants* $k1$*,* $k2$ *such that*

- $< c1, S1 > -k1 \rightarrow < c1', S1' >$ *and* $< c2, S2 > -k2 \rightarrow < c2', S2' >$*,*
- $S1' \equiv S2'$*, and*
- $c1'.instruction = c2'.instruction = RET$ *command.*

Definition 8.12. *Two programs* $prg1 = (f1, FS1)$*,* $prg2 = (f2, FS2)$ *are equivalent if* $f1 \equiv f2$*.*

8.2 Instrumentation Algorithm Description

Following is a simplified version of the instrumentation process.

Given a program $prg1 = (f1entry, FS1)$, the instrumentation process generates a program $prg2 = (f2entry, FS2)$ such that:

1. $f2entry$ is a new function that initializes the private stack, and then CALLs $f1entry$.
2. For each function $f1$ in $FS1$ generate a function $f2$ in $FS2$ as follows:
   - The commands at the beginning of $f1$ are replaced by a JMP to a new location $f2ep$. The rest of $f1$ is left unchanged.
   - Code for pushing the return address to the private stack is inserted at location $f2ep$.
   - All the instructions of $f1$ are copied to new addresses after the above mentioned entry code. While copying, special rules are applied to RET and branch commands:
     - RET commands are converted to a sequence of exit commands that pop the private stack and compare the value on its top to the return address on top of the stack. The original RET command is concatenated to the new exit code.
     - Branch commands' parameters (destination addresses, whether relative or absolute) are updated to take into account the added code.
     - If an indirect JMP is found, the instrumentation process fails.

8.3 Limitations

In this section, we list the limitations on a binary to be instrumented. Our proof sketch will show that as long as these limitations are met, the correctness of the instrumentation can be proved.

1. Binary's Functions: The binary to be instrumented consists of functions with the following limitations:
   a. The program consists of a set of functions. All these functions are discovered by the disassembly process. It is clear that if this limitation is not met, we cannot promise full instrumentation of the program, nor can we prove the instrumentation's correctness (since our semantics needs to know how each CALL instruction affects the state).
   b. All the function's instructions are found (there is no "hidden" function code that was not detected by the disassembly process). If we cannot

promise discovering a whole function, our instrumentation code might have false alarms; we may add an instrumentation entry code to a function, but miss its exit code and, thus, our private stack will be unbalanced. For this reason, our instrumentation process does not handle programs that use indirect JMPs.

c. Functions are invoked only by CALL commands, and are left only by RET commands (no alternative implementation of CALL or RET, or "manual" manipulation of the RALs are allowed). Our current implementation discovers each function's entry point by the fact that it is CALLed, and discovers its exit points by RET commands. Using an alternative mechanism of calling a function (such as pushing a return address by a PUSH command and JMPing to a function's entry address) might cause the function to be left undetected.

d. The first 5 bytes (the length of a long JMP command) of each function are code and not data. If this limitation is not met, overwriting of the first commands of a function by a JMP to a new area will overrun data and, thus, change the function's semantics.

2. Non-self-Modifying or Self-Referencing Program: The binary to be instrumented should not be self-modifying since we statically instrument it. The instrumentation changes the binary and, thus, self-referencing program's semantics will not be preserved by the instrumented version.
3. Limited Nesting Level: The program has a fixed, limited nesting level (limited by the size of the private stack).
4. The program to be instrumented does not manipulate memory areas of the private stack. If the program manipulates the private stack area—then we cannot rely on values of the private stack. This would affect the semantics of the instrumented program (the instrumented program might halt, reporting stack smashing because of a change of the data in the private stack).
5. The program to be instrumented does no reference data that is above the stack (stack areas that have been freed). The instrumentation code uses the stack, thus values of locations above stack in the instrumented program may differ from the same locations in the uninstrumented version.

8.4 A Correctness Proof Sketch

In this section, we present our proof sketch. We show that the instrumented program and the original program are observational equivalent, and that the instrumented version does catch stack smashing attacks.

Theorem 1. *Under the above limitations, the generated prg2 has the following properties:*

1. *Our nonstandard semantic model is equivalent to the standard semantics.*
2. *prg1 and prg2 are equivalent.*
3. *If starting from state S_0 prg1 stack is smashed, the stack smashing will be detected by prg2 before using the malicious return address on the stack.*

Proof of part 1. Omitted. □

Proof of part 2. Proof sketch: We shall show that $prg1 \equiv prg2$, by showing that all their functions are equivalent.

Suppose $f1 = (c1, CS1)$ is converted to $f2 = (c2, CS2)$. Let ecl be the instrumentation's entry code length, and xcl be the instrumentation's exit code length.

The instrumentation's entry code does not affect the function's behavior. If $f2 = (c2, CS2)$, then for any state S, $< c2, S > -ecl \rightarrow < c2e, S'' >$, where $S \equiv S'$, and RAL remains unchanged. This is true under the limitations set above. The new JMP that replaced the beginning of the original function can be placed, since by limitations (1), (1d), (2) the program does not reference the area taken by this JMP instruction. We can show that the instrumentation entry code itself does not affect the observational equivalence of states by analyzing the few commands comprising it. We rely on limitation 5 and let the instrumentation code use the stack.

We would like now to prove that $c2e \equiv c1$, but this is not true, since before each RET command in $f1$ new instrumentation exit code is added in $f2$. What we can prove is that running until the command before an RET instruction in $f1$ the two functions are equal.

Let CR be a set of commands in CS1 such that for each command $cr \in CR$ $cr.instruction = \text{RET}$. For each state S, there exists a constant k such that $< c1, S > -k \rightarrow < cr, S' >$ and $cr \in CR$ (since prg1 halts, each function will return from one of it's return commands. If prg1 does not halt, then $c2e \equiv c1$, since the new commands that we added at the epilogue of the function are not reachable and thus not relevant). We claim that $c2e = k = c1$. This is true since the construction of $f2$ is by copying commands from $f1$, without changing data references, while only updating the jump destinations. Under the limitations mentioned above, we can copy $f1$'s commands: The only code that directly references the moved code is branches within the function $f1$. These branches are updated during the copying process (recall limitation 2 that ensures that commands other than branches should not reference the code).

The inserted exit commands do not affect the state observational equivalence. If while running $f1$ a command $c1$ from a state $S1$ such that $< c1, S1 > \rightarrow < c1', S1' >$, $c1'.instruction = \text{RET}$ is reached, in the copy of $c1$, running the command $c2 \equiv c1$ in $f2$ from a state $S2$ such that $S1 \equiv S2$ will reach $c2' = \text{RET}$ after a finite known number of commands—xcl, with a state $S2'$ such that $S1' \equiv S2'$. This can be shown by analyzing the new exit code commands—it is easy to show that they only manipulate the private stack, using the stack regularly to temporarily save registers (recall limitation 5).

Thus, by the definition of equivalence of functions we can show that every function $f2$ created in $FS2$ in equivalent to a function $f1$ in $FS1$: Assume that for given states $S1$, $S2$ such that $S1 \equiv S2$, $< c1, S1 > -k1 \rightarrow < c1', S1' >$, where

$$c1'.instruction = \text{RET}.$$

Then, as previously shown,

$$< c2, S2 > -k2 \rightarrow < c2', S2',$$

$S1' \equiv S2'$, where $c2'.instruction = \text{RET}$, and $k2 = k1 + ecl + xcl$. This is true for all functions in $FS1$ and $FS2$, thus $prg1 \equiv prg2$. □

Proof of part 3. Proof sketch: We shall show that for a state S_0 that stack-smashes a program prg1, the instrumented program prg2 will detect the stack smashing.

Suppose prg1 is subject to a stack smashing attack. This means that there is some function $f1$ (which in $prg2$ is instrumented to $f2$), where when reaching the RET command, the relevant return address in RAL points to some wrong location. We have seen in part 2 of the proof that up to the new exit code inserted in $f2$, $f2$ behaves like $f1$. Because of limitations 3 and 4, the new exit code will now detect the stack smashing when comparing the return address on the stack to the one on the private stack. □

9 Alternative Approaches and Tools

Following is a brief description of existing approaches to defend against stack smashing. Detailed surveys can be found in [50], [1]. A comparison of various approaches can be found in [7].

9.1 Developer Tools

Probably, the most influential anti-stack-smashing tool is StackGuard. StackGuard [15] is a compiler enhancement, that equips the generated binary code with facilities that can detect a stack smashing attack. StackGuard works by having each function's entry code place a per-run constant, so called a *canary*, on the stack. The function's exit code verifies the canary's existence. The assumption is that a buffer overflow which overwrites the return address would also overwrite the canary. StackGuard has been commercialized by Immunix [25] and has been used to produce a full hardened Unix system. A similar compiler option is now supplied as a standard feature in Microsoft Visual C++ .NET compilers [34], [23].

A different mechanism to detect stack smashing was implemented in StackShield [47]. In StackShield, the attack detection is based on tracking changes of the actual return address on the stack. Each function's return address is recorded in a private stack upon function entry, and the function's exit code verifies that the return address has not changed. This mechanism can detect attacks that try to modify the return address without touching the canary and, thus, is more secure than StackGuard. StackShield is implemented as a GNU compiler enhancement.

Small and Seltzer [46] also inserted Stackshield-like instrumentation. Their system transforms C++ code compiled by the GNU C++ compiler into safe binary code by rewriting the assembly code output by the compiler.

Another mechanism implemented as a compiler enhancement is PointGuard [7]. PointGuard encrypts pointers that point to code. The encryption ensures that the attacker cannot predictably modify code pointers, thus preventing the attacker from causing his code to run. Other types of a compilers enhancements, such as those suggested in [30], [41], attempt to equip the generated binary code with code and data that will enable the detection of the event of overflowing of a buffer as it occurs. The enhanced binary code generates data regarding buffer limits when buffers are allocated, and tests the validity of buffer accessing operations to detect buffer overflowing.

Cyclone [26] is a dialect of the C programming language. It prevents buffer overflows by restricting the C language to a subset of the original language, that is less error prone, but also less powerful. CCured [35] is a tool that combines a safer dialect of the C programming language with runtime tests to ensure the programs are type safe.

Static source code analysis techniques have also been developed to detect software vulnerabilities that may be exploited by stack smashing attacks [21], [16], [17] [19], [49]. The techniques exhibit a clear tradeoff between accuracy of detection and scalability: The more accurate techniques can handle functions comprised of only a few tens of lines, and the more efficient techniques tend to be less accurate heuristics.

Hardware solutions were also developed to thwart stack smashing attacks. The Intel IA32 architecture [28] offers a mechanism to prevent code from running from various memory address ranges, which places an obstacle on code injection attacks. The Intel Itanium architecture offers a mechanism to protect return addressing, by adding special CALL and RET instructions where the return address can be explicitly declared—thus, allowing the compiler or the developer to use secure return addresses. Another hardware solution that performs validity checks on return addresses is [48].

9.2 User Tools

Many stack smashing exploits inject code into the stack, and then overrun the return address, residing on the stack to the injected code. A user tool, implemented as a Linux patch prevents such exploits by setting the stack memory pages to be nonexecutable [43]. SecureStack, is an implementation of a nonexecutable stack for the Windows OS developed by SecureWave [42].

Libsafe [2] is a runtime attack detection mechanism that can discover stack smashing attacks against standard library functions. It is implemented as a dynamically loaded library that intercepts calls to known vulnerable library functions, and redirects them to a safe implementation of these functions.

Libverify [2] is a Unix-based attack detection technique similar to StackShield, but in which the attack detection code is embodied into a copy of the executable image, which is created on the heap at load time. However, the authors only handled the simplest case (single threaded programs, no DLLs, on a Unix-based system). Libverify needs to hook into the program loader—a difficult requirement to meet on a proprietary operating system—and also doubles the memory needed to run the program.

The hardware-based user tool [32] relies on a processor-managed private stack, where only the processor can manage the private stack.

Recently, a technique based on randomized instruction set emulation, employed at runtime, has shown success in

detecting various code injection attacks, including stack smashing attacks [27], [1]. This technique has a high-performance penalty because of the emulation overhead.

9.3 Instrumenting Binaries

Instrumenting binaries and binary translation has become an active area of research over the past few years. In [29], instrumentation has been implemented to add profiling measurements to a given binary file. The main issue is whether and how a binary can be instrumented without changing the original program's semantics. One of the basic questions in this field is whether a program's code can be distinguished from its data, given a binary file. Recent research [14], [8] show that in most cases, this can be done: Assuming that the code was generated by a compiler, data can be separated from code. Furthermore, assuming that the code is not self-modifying and does not reference code as data, the binary's logic can be discovered. Thus, instrumentation, without changing the program's semantics, is possible.

9.4 Comparison with the Work of Prasad and Chiueh

The work closest to ours was independently suggested by Prasad and Chiueh [38], in parallel with early versions of our work [36], [37]. They too add anti-stack-smashing instrumentation into Windows binary files. However, their work leaves several key issues unresolved. Most notably, they only handle simple Win32 executables (single thread programs, no DLLs). Furthermore, they demonstrated that their approach successfully vaccinates a test program with a deliberate buffer overflow, rather than a live exploit. Finally, they use the Detours library [22], which slightly limits the class of functions they are able to instrument, and requires load-time activation of the instrumentation. Beyond the work of [38], our work has the following features:

- We are able to instrument DLLs.
- We handle multithreaded applications.
- We handle DLLs called by multithreaded applications.
- We can instrument a wider set of functions, including functions with jumps into the middle of entry/exit code, and very short functions. This is because we wrote our own instrumentation code rather than using Detours.
- We demonstrated that our approach protects against a real buffer overflow vulnerability found in the wild.

10 Conclusions

We presented an install-time vaccination technique as a countermeasure against stack-smashing-attacks on the proprietary Microsoft Windows OS. Our technique enables software users to be protected without access to the source-code.

We chose to implement a somewhat complicated defense technique, that uses memory outside of the stack. We have successfully applied our technique to binary executables, including those using shared memory and concurrency. The fact that we can do so demonstrates the feasibility of developing other security instrumentation techniques that may require memory, such as encryption or digital signing.

We developed a prototype that successfully instruments simple Win32 applications, DLLs, multithreaded applications, and DLLs used by multithreaded applications. Our approach has a very low performance penalty. We have shown that the prototype can vaccinate standard Windows executables, and can defend against real exploit code. Therefore, our vaccination technique can be considered a real-world system-wide solution.

Acknowledgments

An extended abstract of this work appears in the *Proceedings of the 19th IFIP International Information Security Conference, 2004.*

References

[1] E.G. Barrantes, D.H. Ackley, S. Forrest, T.S. Palmer, D. Stefanovic, and D.D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS),* 2003.

[2] A. Baratloo, N. Singh, and T. Tsai, "Transparent Runtime Defense against Stack Smashing Attacks," *Proc. USENIX Ann. Technical Conf.,* 2000.

[3] "Hotfoon Dialer Buffer Overflow Vulnerability,"Bugtraq id 6156, Nov. 2002, http://www.securityfocus.com/bid/6156.

[4] "Microsoft Windows DCOM RPC Interface Buffer Overrun Vulnerability,"Bugtraq id 8205, July 2003, http://www. securityfocus.com/bid/8205.

[5] "Microsoft Windows RegEdit.exe Registry Key Value Buffer Overflow Vulnerability,"Bugtraq id 7411, Apr. 2003, http://www.securityfocus.com/bid/7411.

[6] "Adding Sections to PE Files: Enhancing Functionality of Programs by Adding Extra Code," 1999, http://bib.universitas-virtualis.org/go. php?id=bibuv-gdl-grey-2004-c0v3rt-119.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," *Proc. 12th USENIX Security Symp.,* 2003.

[8] C. Cifuentes and M. Van Emmerik, "Recovery of Jump Table Case Statements from Binary Code," *Science of Computer Programming,* vol. 40, nos. 2-3, pp. 171-188, 2001.

[9] CERT/cc Statistics 1988-2001, 2002, http://www.cert.org/stats/.

[10] "CERT Advisory CA-2003-16: Buffer Overflow in Microsoft RPC," July 2003, http://www.cert.org/advisories/CA-2003-16.html.

[11] "CERT Advisory CA-2003-20: W32/Blaster Worm," Aug. 2003, http://www.cert.org/advisories/CA-2003-20.html.

[12] "CERT Vulnerability Note VU#579324: Cisco IOS HTTP Server Vulnerable to Buffer Overflow When Processing Overly Large Malformed HTTP GET Request," 31 July 2003, http://www.kb.cert.org/vuls/id/579324.

[13] S. Cho, "Windows Disassembler, v0.22," 2000, http://cyber.chongju.ac.kr/~sangcho/index.html.

[14] C. Cifuentes, "Partial Automation of an Integrated Reverse Engineering Environment of Binary Code," *Proc. Working Conf. Reverse Eng.,* pp. 50-56, 1996.

[15] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. Seventh USENIX Security Symp.,* pp. 63-78, Jan. 1998.

[16] N. Dor, M. Rodeh, and M. Sagiv, "Cleanness Checking of String Manipulations in C Programs via Integer Analysis," *Proc. Eighth Int'l Static Analysis Symp. (SAS),* 2001.

[17] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation,* pp. 155-167, 2003.

[18] D.C. DuVarney, V.N. Venkatakrishnan, and S. Bhatkar, "SELF: A Transparent Security Extension for ELF Binaries," *Proc. 2003 Workshop New Security Paradigms,* pp. 29-38, 2003.

[19] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysi," *IEEE Software,* vol. 19, no. 1, pp. 42-51, 2002.
[20] M.W. Eichin and J.A.A. Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," *Proc. IEEE Symp. Security and Privacy,* 1989.
[21] A.K. Ghosh and T. O'Connor, "Analyzing Programs for Vulnerability to Buffer Overrun Attacks," *Proc. 21st NIST-NCSC Nat'l Information Systems Security Conf.,* pp. 274-382, 1998.
[22] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," *Proc. Third USENIX NT Symp.,* pp. 135-144, 1999.
[23] M. Howard and D. LeBlanc, *Writing Secure Code,* second ed. Microsoft Press, 2002.
[24] "The IDA Pro Disassembler and Debugger,"v4.51, 2003, http://www.datarescue.com/idabase/.
[25] Immunix Secured Solutions, 2003, http://www.immunix.com.
[26] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," *Proc. USENIX Ann. Technical Conf.,* June 2002.
[27] G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS),* 2003.
[28] S. Kuo, "Execute Disable Bit Functionality Blocks Malware Code Execution," White paper, Intel, 2005, http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/pentium4/optimization/149308.htm.
[29] J.R. Larus and T. Ball, "Rewriting Executable Files to Measure Program Behavior," Technical Report CS-TR-92-1083, Univ. of Wisconsin, Madison, 25 Mar. 1992.
[30] K.-s. Lhee and S.J. Chapin, "Type-Assisted Dynamic Buffer Overflow Detection," *Proc. 11th USENIX Security Symp.,* 2002.
[31] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS),* 2003.
[32] R.B. Lee, D.K. Karig, J.P. McGregor, and Z. Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," *Proc. Int'l Conf. Security in Pervasive Computing (SPC-2003),* Mar. 2003.
[33] *Microsoft Portable Executable and Common Object File Format Specification,* rev. 6.0, 1999, http://www.microsoft. com/whdc/hwdev/hardware/pecoff.mspx.
[34] "Microsoft Visual C++ Compiler Options: /gs (Control Stack Checking Calls),"Online documentation, 2001, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_.2f.gs.asp.
[35] G.C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," *Proc. Symp. Principles of Programming Languages,* pp. 128-139, 2002.
[36] D. Nebenzahl and A. Wool, "Install-Time Vaccination of Windows Executables to Defend against Stack Smashing Attacks," Technical Report EES2003-9, School of Electrical Eng., Tel Aviv Univ., 2003.
[37] D. Nebenzahl and A. Wool, "Install-Time Vaccination of Windows Executables to Defend against Stack Smashing Attacks," *Proc. 19th IFIP Int'l Information Security Conf.,* pp. 225-240, Aug. 2004.
[38] M. Prasad and T.-c. Chiueh, "A Binary Rewriting Defense against Stack Based Overflow Attacks," *Proc. USENIX 2003 Ann. Technical Conf.,* 2003.
[39] "PEDasm: A Symbolic Disassembler for Win32," 2003, http://www.geocities.com/SiliconValley/Lab/6307/.
[40] G. Richarte, *Four Different Tricks to Bypass StackShield and StackGuard Protection,* Core Security Technologies, 2002, http://downloads.securityfocus.com/library/StackGuard. pdf.
[41] O. Ruwase and M. Lam, "A Practical Dynamic Buffer Overflow Detector," *Proc. Network and Distributed System Security (NDSS) Symp.,* pp. 159-169, Feb. 2004.
[42] "SecureStack v1.0: Buffer Overflow Protection for Windows NT/2000," 2001, no longer available, a similar design can be found at http://datasecuritysoftware.com/index.html.
[43] Solar Designer, "Nonexecutable User Stack," http://www.false.com/security/linux-stack/, 2006.
[44] E.H. Spafford, "The Internet Worm Program: An Analysis," Technical Report CSD-TR-823, Purdue Univ., West Lafayette, IN 47907-2004, 1988.
[45] SPEC CPU2000 V1.2. Standard Performance Evaluation Corporation, 2000, http://www.specbench.org/osg/cpu2000/.
[46] C. Small and M. Seltzer, "MiSFIT: A Tool for Constructing Safe Extensible Systems," *IEEE Concurrency,* pp. pp. 33-41, 1998.
[47] Stackshield, 2000, http://www.angelfire.com/sk/stackshield.
[48] Z. Shao, Q. Zhuge, Y. He, and E.H.-M. Sha, "Defending Embedded Systems against Buffer Overflow via Hardware/Software," *Proc. Ann. Computer Security Applications Conf.,* 2003.
[49] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Network and Distributed System Security Symp. (NDSS),* pp. 3-17, Feb. 2000.
[50] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *Proc. 10th Network and Distributed System Security Symp. (NDSS),* pp. 149-162, Feb. 2003.

Daniel Nebenzahl received the BSc degree from the Jerusalem College of Technology and the MSc degree from the Tel-Aviv University. He is interetsed in the many facets of system, computer, and communication security.

Mooly Sagiv received the PhD degree in computer science from Technion Israel Institute of Technology, Haifa. He joined Tel Aviv University's School of Computer Science in 1997 where in 2000 he was a senior lecturer and since 2004, he has been an associate professor. He has been a visiting professor at the University of Chicago and Datalogisk Institute at the University of Copenhagen. In addition, he has been a researcher at the University of Wisconsin-Madison and IBM's Israel Scientific Center. His research interests include programming languages, compilers, abstract interpretation, profiling, pointer analysis, shape analysis, interprocedural dataflow analysis, program slicing, and language-based programming environments.

Avishai Wool received the BSc (cum laude) degree in mathematics and computer science from Tel Aviv University, Israel, in 1989, and he received the MSc and PhD degrees in computer science from the Weizmann Institute of Science, Israel, in 1993 and 1997, respectively. Dr. Wool then spent four years as a member of technical staff at Bell Laboratories, Murray Hill, New Jersey. In 2000, Dr. Wool cofounded Lumeta corporation, a startup company specializing in network security, and its successor, Algorithmic Security Inc. Since 2002, Dr. Wool has been an assistant professor in the School of Electrical Engineering, Tel Aviv University, Israel. Dr. Wool is the creator of the firewall analyzer. He is an associate editor of the *ACM Transactions on Information and System Security*. He has served on the program committee of the leading IEEE and ACM conferences on computer and network security. He is a senior member of IEEE, and a member the ACM and USENIX. His research interests include firewall technology, network and wireless security, smartcard-based systems, and the structure of the Internet topology.