Install-time Vaccination of Windows Executables to Defend Against Stack Smashing Attacks

Danny Nebenzahl* Avishai Wool†

November 4, 2003

Abstract

Stack smashing is still one of the most popular techniques for computer system attack. In this paper 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, multi-threaded applications and DLLs used by multi-threaded applications, which present significant additional complications. We present promising performance results measured on SPEC2000 benchmarks: Vaccinated executables were no more than 8% slower than their un-vaccinated originals.

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 [CER02]. Stack smashing is an old technique, dating back to the late 1980's. E.g, the Internet worm [Spa88, ER89] 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 [CER03a, CER03b]. 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 [CER03c]. 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 [BST00, CPM+98].

*Dept. of Computer Science, Tel Aviv University, Ramat Aviv 69978, Israel. nenenzah@post.tau.ac.il
†School of Electrical Engineering, Tel Aviv University, Ramat Aviv 69978, Israel. yash@acm.org

Technical Report EES2003-9,
School of Electrical Engineering, Tel Aviv University, Ramat Aviv 69978 ISRAEL.

1.2 Related Work

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 gaining control over the attacked host: i.e., they 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 counter-measures 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 the operating system, the compilers, and application software. We are not aware of any user tools that address the popular, and much more challenging, Microsoft Windows operating systems.[1]

Following is a brief description of existing approaches to defend against stack smashing. Detailed surveys can be found in [WK03, BAF+03].

1.2.1 Developer tools

Probably the most influential anti-stack-smashing tool is StackGuard. StackGuard [CPM+98] 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 [Imm03] 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 [Mic01, HL02].

[1] Recently Microsoft has deployed a compile-time anti-stack-smashing feature in its Visual C++ .NET compilers.

A different mechanism to detect stack smashing was implemented in StackShield [Sta00]. 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. StackShield is implemented as a compiler enhancement.

Cyclone [JMG+02] 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.

Static source code analysis techniques have also been developed to detect software vulnerabilities that may be exploited by stack smashing attacks [GO98, DRS01, EL02, WFBA00]. 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.

Another compiler enhancement, suggested in [LC02], equips the generated binary code with type and buffer size information, and attempts to use this information in order to detect the event of a buffer overflow.

1.2.2 User tools

Libsafe [BST00] is a run time 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.

The approach closest to ours is Libverify. Libverify [BST00] is an 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.

Recently, a technique based on randomized instruction set emulation, employed at run time, has shown success in detecting various code injection attacks, including stack smashing attacks [KKP03, BAF+03]. This technique has a high performance penalty because of the emulation overhead.

1.2.3 Instrumenting Binaries

Instrumenting binaries and binary translation has become an active area of research over the past few years. In [LB92] 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 it's data, given a binary file. Recent research [Cif96, CE01] shows 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.

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 we target the Microsoft Windows operating systems, running over Intel's x86 architecture. The complexity of the x86 CISC architecture, and the details of the Windows OSes, make this a much more challenging target than Unix over RISC. To the best of our knowledge, ours is the first tool that has both features.

We have built a working prototype implementing our approach, that can instrument Win32 applications running on an x86 Intel Pentium platform. In addition to simple applications, our prototype properly handles the complexities of DLLs and multi-threaded applications. 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% slowdown in the vaccinated program.

Organization: In Section 2 we briefly introduce the structure of Win32 executables. In Section 3 we describe our solution architecture. Section 4 describes the implementation of our technique to vaccinate simple Windows applications. Section 5 describes the techniques used to vaccinate Windows DLLs. Section 6 describes the techniques used to overcome the challenges imposed by multi-threaded applications. In Section 7 we evaluate our solution.

2 Win32 Executables

Before delving into the details of our anti-stack-smashing technique, we first briefly introduce the structure of Win32 executables. The executable file format used in Microsoft operating systems from Win95 up to Windows 2000 and Windows XP is the PE file structure, where PE stands for Portable Executable. The specification of the PE file structure can be found in [Mic99].

The PE file uses a uniform file structure that describes applications, DLLs (Dynamic Link Libraries), drivers and object files. It supports memory allocation, dynamic linkage to other PE files, per-thread memory allocation and other advanced features. A general sketch of the PE file format can be seen in Figure 1. The main components of the PE file structure are as follows:

- MS DOS Header: A header for backward compatibility. Always contains a small program that prints a message in case the program is run in an MS-DOS environment.
- A PE Header: A header common to all PE files. This header contains information about the machine on which the binary is supposed to run, how many sections are in the executable, the time it was linked and whether it is an executable or a DLL.
- An Optional Header: An additional header containing more information about how the binary should be loaded. This is the area holding information such as the starting address, the amount of stack to reserve, and the size of the data segment. The trailer of the Optional Header is a structure called the

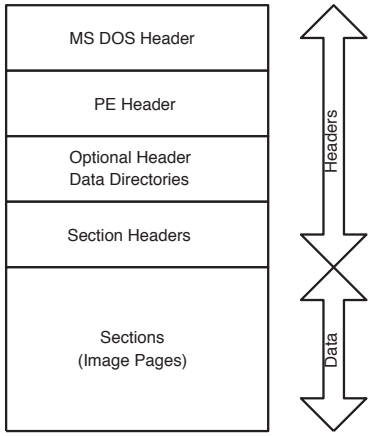

Figure 1: The PE File Format

Data Directories. The Data Directories contain pointers to data in the 'sections'. If, for example, the binary has a function export directories (that is the case in DLLs), there should be a pointer to that directory in the Data Directory structure.

- Section Table: This is an array of entries describing the sections of the file. Each entry describes how the operating system should handle the section by supplying data such as the address the section is expected to be loaded to, what types of data it contains (such as initialized data or uninitialized data), and whether it can be shared.
- Sections: The actual binary data of the file. Sections may contain compiled code, data used by the program, structures used by the operating system to use the file, and other data. Sections may contain a mixture of the above mentioned items, however, this is non-standard. The PE file format specifies several section names that should contain specific data.

A simple program written in C/C++, compiled with the Microsoft Visual C++ compiler, would have the following sections:

- ".text" - containing code.
- ".rdata" - containing read-only initialized data.
- ".idata" - containing initialized data.
- ".reloc" - containing a relocation table, to support the loading of the program to various locations in the memory space.

3 Solution Architecture

3.1 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

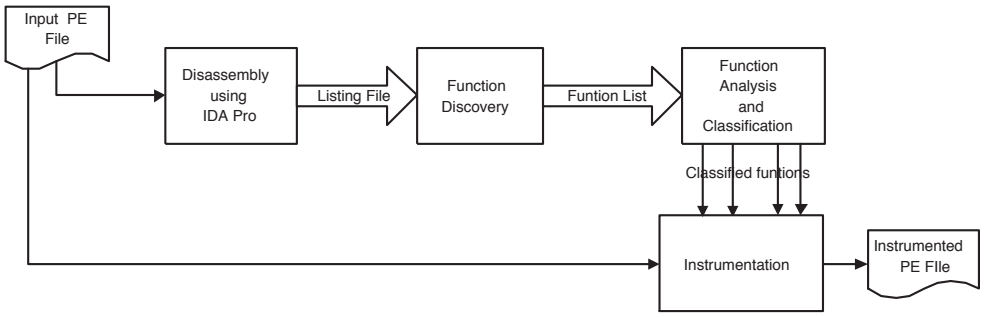

Figure 2: The PE Instrumentation Process

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. This is similar to the approach taken by StackShield. The differences are that we instrument the executable file, once (not on every load), our instrumentation is much more efficient, and we address the Windows operating system.

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 [Ric02].

3.2 Instrumentation of Binaries

We implemented our approach using static instrumentation: Our vaccination tool instruments the executable file, not its loaded image in memory. 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. The process of instrumenting a binary consists of the following steps:

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

A diagram of the instrumentation process is shown in Figure 2. In the next sections we describe each of these steps in some detail.

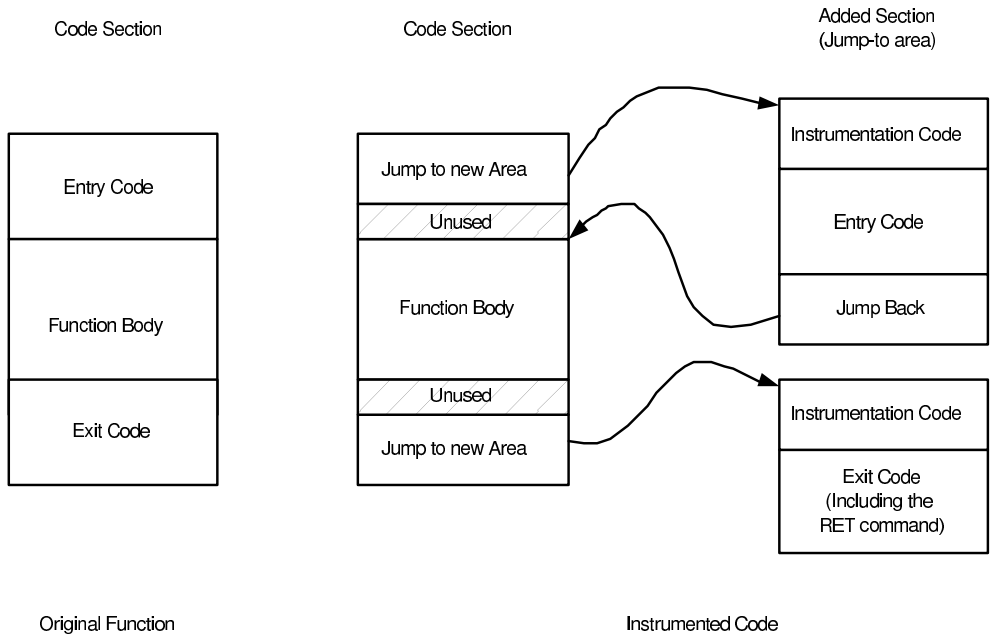

Figure 3: Instrumentation of Simple Functions

3.3 Disassembly

Since the disassembly process is not in the core of our research, we chose to use a commercial disassembly package, the IDA Pro [IDA03]. We chose IDA Pro since it has been recognized as a very accurate disassembler that is capable of distinguishing between code and data [CE01]. Our own experience showed IDA Pro to be more accurate than a number of shareware and free open-source disassemblers (e.g., [PED03, Cho00]). The input to IDA Pro is the binary to be disassembled and the output is a listing file of the disassembled program.

3.4 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 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 "non-standard" 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 generate standard call sequences.

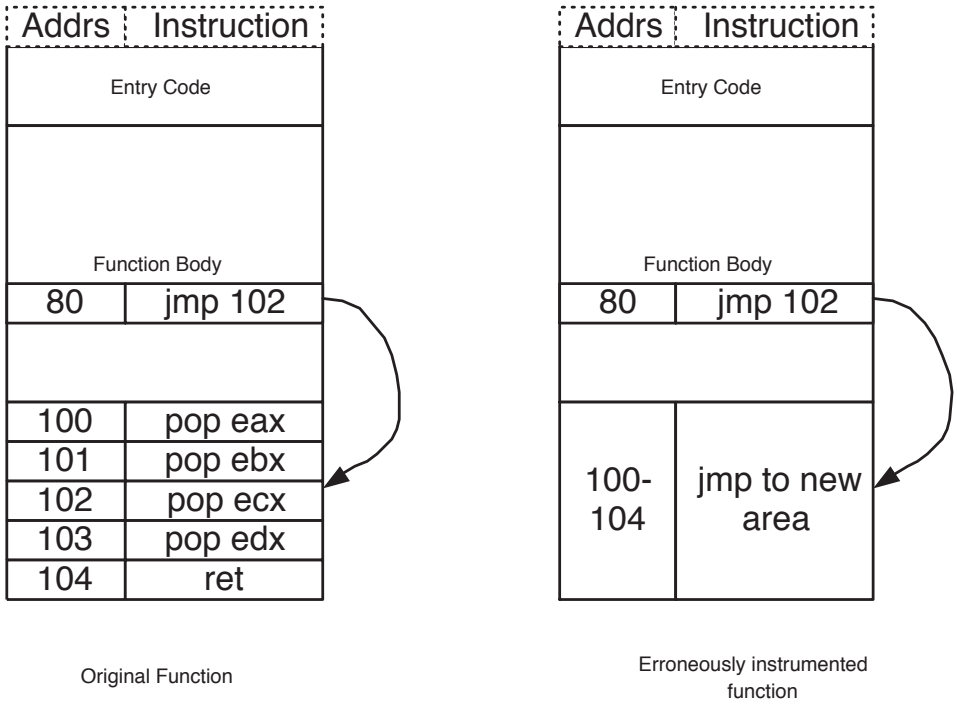

Figure 4: Erroneous Function instrumentation in a CISC Architecture. The original 5-byte, 5-instruction exit code was replaced by a single 5 byte jump instruction. The jump from address 80 now lands in the middle of an instruction.

3.5 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.
- 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 Figure 3.

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 Figure 4. 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.
- Un-handled functions. Functions that include indirect 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 [CE01], 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 [CE01, LB92]. In all of the programs we have instrumented, indirect jumps were used in less than 4% of the functions.

3.6 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 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 [c0v99]. 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.

4 Instrumenting a Simple Win32 Application

In our terminology, a *simple Win32 application* is a windows application that is not multi-threaded. 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 in 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 [Bug03] 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 3. We also instrumented other simple Win32 applications (such as Notepad.exe, WinHlp32.exe, 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.

5 Instrumenting a DLL

DLLs (Dynamic Link Libraries) are PE files that contain function libraries that can be used by multiple applications simultaneously. The benefits of DLLs are:

- Software engineering — the ability to separately design, implement and debug function libraries and programs, without the need to recompile or relink when either of them changes.
- Increase of performance — the same function libraries can serve multiple applications running simultaneously, thus saving memory and paging time. A DLL can be used simultaneously by multiple processes.

The use of DLLs is extremely common in the Microsoft windows family of operating systems: e.g., there are over 1000 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 run time, 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 new copied functions (e.g., an access to a global variable) will 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 multi-process 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 yet tested our defense against a real exploit of a real DLL.

6 Instrumenting A Multi-threaded Application

Modern operating systems allow for multi-threading: 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 multi-threaded application, there is a need for per *thread* private stack memory allocation.

In a multi-threaded environment, the process's memory is shared between threads. In contrast to multiprocess operation, in a multi-threaded 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 5) is not applicable for the multi-threaded 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

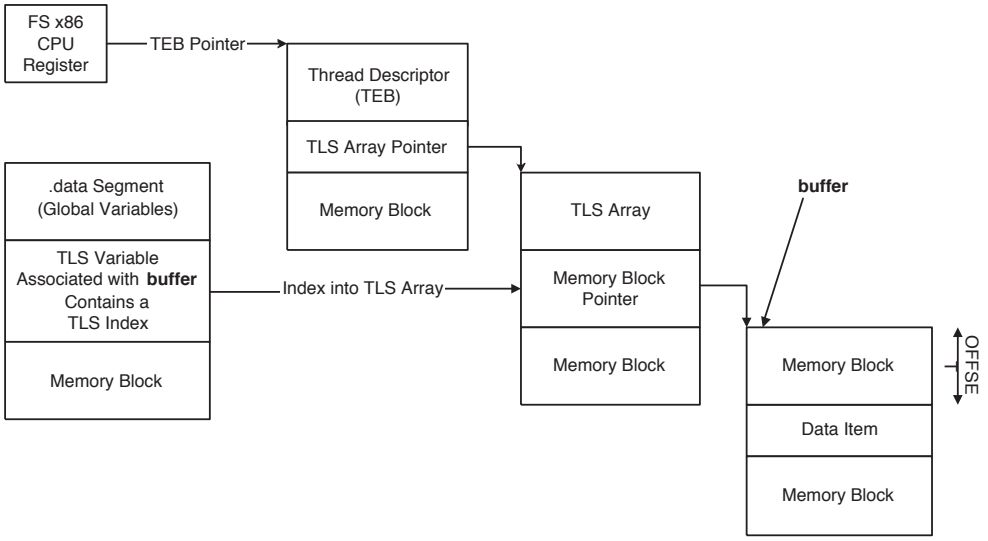

Figure 5: Accessing the TLS variable *void *buffer* at offset *OFFSET*

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 run time access sequence is shown in Figure 5.

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 it's initialization function. The instrumentation code accesses the private stack using a sequence defined by the operating system as described in [Mic99]. Note that accessing a TLS variable has a performance penalty greater than that of 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 multi-threaded 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. We also tested the correct operation of various instrumented applications such as Windows Explorer, Microsoft Internet Explorer and more.[2]

[2]We do not know for sure whether these are multi-threaded or simple applications. However, since these are large and complex programs, we used our multi-threaded instrumentation method, to be on the safe side.

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 1: Run time performance measurements of standard 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

Table 2: PE file size performance measurements of SPEC2000 and instrumented SPEC2000 applications.

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.
- 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 micro-benchmarks. To do this, we instrumented several SPECINT applications from the SPEC2000 suite [SPE00] 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 multi-threaded 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 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 run time. We verified that the instrumented code produced the same output as the standard uninstrumented code. The measured performance penalty was less than 8% for the SPEC2000 applications we measured (see Tables 1 and 2 for details), and the increase in program size was 20-37%. We consider these results very positive: an 8% 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 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 two limitations caused by the fact that we instrument a binary executable file, rather than its loaded image.

1. Known Un-handled Functions: As noted in Section 3.5 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 Un-handled 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.

The first limitation can be addressed by introducing techniques of jump table discovery, which will reduce the number of un-handled functions. The second limitation can be handled at run time, for example by checking that the destination of an indirect call is indeed instrumented. These improvements are left for future work.

7.3 A System-wide Solution

In our research we have developed solutions for three different cases of Windows executables (PE files): simple applications, DLLs, and multi-threaded applications. In this section we would like to examine whether this is enough to form a system-wide solution. In order to answer this question we need to define what a system-wide solution is. We argue that a system-wide solution is an instrumenting application, that can vaccinate any executable file of the user's choice, without forcing him to instrument other executables, and without requiring a-priori knowledge about the executable properties.

At this time our prototype does not meet these requirements. Firstly, our prototype currently cannot instrument a DLL which is used by a multi-threaded application. The reason is that TLS variables added to a PE file cannot be used in DLLs, because a DLL may be linked to an executable dynamically, during the program run. 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. When instrumenting a DLL, one does not know if it will be used by a multi-threaded application, thus the current tool cannot correctly instrument a general purpose DLL. In future work we plan to check the feasibility of dynamically allocated TLS variables as part of our tool in order to solve the multi-threaded DLL case.

A second reason why our prototype is not yet a system-wide solution is that it needs to know whether it is vaccinating a simple or multi-threaded application. Note that simple applications can be treated as multi-threaded application that have only one thread. However, the performance of simple applications' instrumentation is much better than that of multi-threaded applications. In future work we plan to investigate ways to determine whether an application is multi-threaded, in order to select the best instrumentation method.

Finally, we need to test our tool on more Win32 applications and DLLs, and validate its success in defending against more real exploits.

8 Conclusions

We presented an install-time vaccination technique as a counter-measure 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, and multi-threaded 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, we believe that our vaccination technique can be extended into a real-world system-wide solution.

References

[BAF+03] 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. In *Proc. 10th ACM Conf. Computer and Communications Security (CCS)*, Washington, DC, 2003.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proc. USENIX Annual Technical Conference*, 2000.

[Bug03] Microsoft Windows RegEdit.exe registry key value buffer overflow vulnerability. Bugtraq id 7411, 16 April 2003. `http://www.securityfocus.com/bid/7411`.

[c0v99] c0v3rt+. Adding sections to PE files: Enhancing functionality of programs by adding extra code, 1999. `http://mup.anticrack.de/c0v3rt+%20-%20Sections.htm`.

[CE01] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2–3):171–188, 2001.

[CER02] CERT/cc statistics 1988-2001, 2002. http://www.cert.org/stats/.

[CER03a] CERT advisory CA-2003-16: Buffer overflow in Microsoft RPC, 17 July 2003. `http://www.cert.org/advisories/CA-2003-16.html`.

[CER03b] CERT advisory CA-2003-20: W32/Blaster worm, 11 August 2003. `http://www.cert.org/advisories/CA-2003-20.html`.

[CER03c] 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`.

[Cho00] Sang Cho. Windows disassembler, v0.22, 2000. `http://cyber.chongju.ac.kr/~sangcho/index.html`.

[Cif96] Cristina Cifuentes. Partial automation of an integrated reverse engineering environment of binary code. In *Working Conference on Reverse Engineering*, pages 50–56, 1996.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[DRS01] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Proc. 8th International Static Analysis Symposium (SAS), LNCS 2126*, Paris, France, 2001. Springer-Verlag.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[ER89] Mark W. Eichin and Jon A. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, 1989.

[GO98] A. K. Ghosh and T. O'Connor. Analyzing programs for vulnerability to buffer overrun attacks. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 274–382, 1998.

[HL02] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2nd edition, 2002.

[IDA03] The IDA Pro disassembler and debugger, v4.51, 2003. `http://www.datarescue.com/idabase/`.

[Imm03] Immunix secured solutions, 2003. `http://www.immunix.com`.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conf. Computer and Communications Security (CCS)*, Washington, DC, 2003.

[LB92] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report CS-TR-92-1083, U. of Wisconsin, Madison, WI, USA, 25 March 1992.

[LC02] Kyung-suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proc. 11th USENIX Security Symposium*. USENIX, 2002.

[Mic99] *Microsoft Portable Executable and Common Object File Format Specification, rev. 6.0*, 1999. `http://www.microsoft.com/whdc/hwdev/hardware/pecoff.mspx`.

[Mic01] 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`.

[PED03] PEDasm: A symbolic disassembler for Win32, 2003. `http://www.geocities.com/SiliconValley/Lab/6307/`.

[Ric02] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection. Core Security Technologies, 2002. `http://downloads.securityfocus.com/library/StackGuard.pdf`.

[Spa88] Eugene H. Spafford. The Internet worm program: An analysis. Technical Report CSD-TR-823, Purdue University, West Lafayette, IN 47907-2004, 1988.

[SPE00] SPEC CPU2000 V1.2. Standard Performance Evaluation Corporation, 2000. `http://www.specbench.org/osg/cpu2000/`.

[Sta00] Stackshield, 2000. `http://www.angelfire.com/sk/stackshield`.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Distributed System Security Symposium (NDSS)*, pages 3–17, San Diego, CA, February 2000.

[WK03] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, pages 149–162, San Diego, California, February 2003.