INSTALL-TIME VACCINATION OF WINDOWS EXECUTABLES TO DEFEND AGAINST STACK SMASHING ATTACKS

Danny Nebenzahl,[1] and Avishai Wool,[2]

[1] *Dept. of Computer Science, Tel Aviv University, Ramat Aviv 69978, Israel.*
nenenzah@post.tau.ac.il

[2] *School of Electrical Engineering, Tel Aviv University, Ramat Aviv 69978, Israel.*
yash@acm.org

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.

Keywords: Computer security, Malware, Buffer-Overflow, Microsoft Windows Operating System

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] used stack smashing. However, this technique is still in

Y. Deswarte et al. (eds.), *Security and Protection in Information Processing Systems*

current use by hackers: For instance, it is the underlying method of attack used by the MSBlast virus [CER03a],[CER03b]. 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].

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 [WK03]:

- 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 [WK03]:

- 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 *vac-*

cinate 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. In contrast, our tool targets the proprietary Microsoft Windows operating systems.

The work closest to ours was recently suggested, independently, by Prasad and Chiueh [PC03]. 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 multi-threading, 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 [PC03] can be found in Section 8.

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, multi-threaded 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% slowdown in the vaccinated program.

Organization: In Section 2 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 multi-threaded 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 describe alternative approaches to stack-smashing protection, and we conclude in Section 9.

2. SOLUTION ARCHITECTURE

2.1 Design Choices

Our first choice was to use a separate stack (as done by LibVerify [BST00] on Linux), rather than insert so-called "canaries" into the stack, as done by StackGuard [CPM$^+$98]. We believe that a separate stack offers better protection than canaries—e.g., [WK03] 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 run-time (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 (cf. [HB99]). In a proprietary operating system like Microsoft Windows, modifying the binary is more robust and less OS-version dependent. Nevertheless, load-time instrumentation is a viable option.

Furthermore, we chose not to use the Detours library [HB99], 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 run-time. 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 [Ric02].

2.3 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], [LD03]. The input to IDA Pro is the binary to be disassembled and the output is a listing file of the disassembled program.

2.4 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.

2.5 Function Analysis and Classification

We 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: (i) Our instrumentation code; (ii) The original entry code instructions that were overwritten by our jump instruction; and (iii) A jump back to the rest of the original function. A similar solution is used to instrument the function's exit-code.

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. 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]. In all of the programs we have instrumented, indirect jumps were used in less than 4% of the functions.

3. 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 2. 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.

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 over 1000 DLLs in a typical Win2000 `C:\WINNT\SYSTEM32` directory. Since DLLs may be used in different combinations, the DLL writer cannot predict the memory address that the DLL will be loaded to. When creating a DLL binary, the compiler creates a *relocation* table. The relocation table enables the OS to load the DLL to any available memory region.

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. When a process writes to a page that is managed according to the Copy-on-Write policy, the OS creates a copy of the page that is private to that 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. 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.

5. INSTRUMENTING A MULTI-THREADED APPLICATION

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

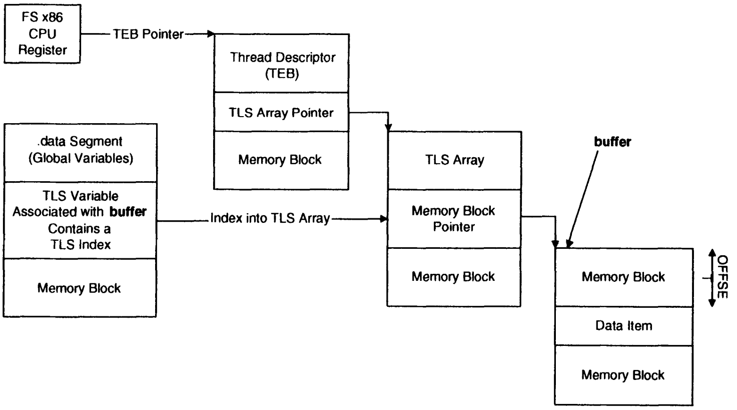

Figure 1. Accessing the TLS variable *void *buffer* at offset *OFFSET*

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 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. 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 and thus has a performance penalty greater 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 run time access sequence is shown in Figure 1.

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. 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.

6. INSTRUMENTING DLLS USED BY MULTITHREADED APPLICATIONS

Another implication of multi-threaded applications is the handling of DLLs used by multi-threaded applications. The technique used to vaccinate multi-threaded applications cannot be used to vaccinate DLLs used by multi-threaded 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 multi-threaded application, a general purpose instrumentation method must address DLLs that are used by multi-threaded applications. In other words, the simpler mechanism we used in section 4 is not sufficient.

We solve the need for private stack memory allocation per-stack by using a Win32 mechanism for allocating dynamically 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.

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	Orig.Size (KB)	Instrumented Size(KB)	Increase (%)	IDA-Pro Listing(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.

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 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 multi-threaded application under controlled race conditions. 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: (i) The added code that needs to be run. (ii) The larger address space that may change the paging performance for the program. (iii) 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 un-instrumented 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 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 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 2.5, 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.

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

8. ALTERNATIVE APPROACHES AND TOOLS

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].

StackShield [Sta00] is another GNU compiler enhancement. 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.

Another tool that inserts instrumentation at compile time is PointGuard [CBJW03] (which encrypts pointers to code). 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].

User tools. A user tool, implemented as a Linux patch, prevents stack smashing exploits by setting the stack memory pages to be non-executable [Sol].

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.

Libverify [BST00] 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).

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.

Comparison with the work of Prasad and Chiueh. The work closest to ours was recently suggested, independently, by [PC03]. 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 [HB99], which slightly limits the class of functions they are able to instrument, and requires load-time activation of the instrumentation. Beyond the work of [PC03], our work has the following features:

- We are able to instrument DLLs.

- We handle multi-threaded applications.

- We handle DLLs called by multi-threaded 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.

9. 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 developed a prototype that successfully instruments simple Win32 applications, DLLs, multi-threaded applications, and DLLs used by multi-threaded applications thus providing a system-wide solution. We have shown that the prototype can defend against real exploit code.

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`.

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. 12th USENIX Security Symposium*. USENIX, 2003.

[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`.

[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.

[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.

[HB99] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Proc. 3rd USENIX NT Symposium*, pages 135–144, 1999.

[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`.

[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.

[LD03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th ACM Conf. Computer and Communications Security (CCS)*, Washington, DC, 2003.

[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`.

[PC03] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX 2003 Annual Technical Conference*, 2003.

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

[Sol] Solar Designer. Nonexecutable user stack. `http://www.false.com/security/linux-stack/`.

[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.