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

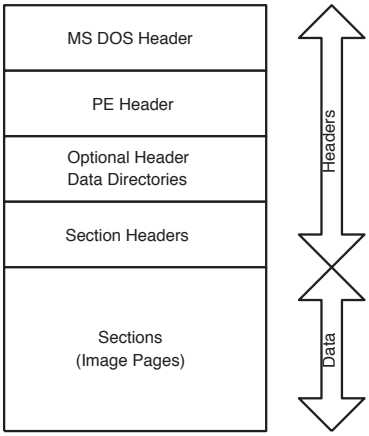

Figure 1: The PE File Format

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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.