TDDC03 Projects, Spring 2005

Comparative Study of Run-Time Defense Against Buffer Overflows

Andreas Ekbom & Stefan Ottosson

Supervisor: John Wilander

Comparative Study of Run-Time Defense Against Buffer Overflows

Andreas Ekbom, Stefan Ottosson
andek406@student.liu.se, sot@lysator.liu.se

Abstract

Buffer overflows are a common source of security problems in software systems. Various tools and techniques have been devised to prevent attacks on software vulnerable to buffer overflow attacks. In this paper we describe the current state of the art in both prevention and attack techniques. Making certain no area in memory is both executable and writable is shown to be an effective countermeasure against code-injecting attacks, but this too can often be circumvented.

Keywords: buffer overflow, return-into-libc, exec shield, DEP, instrusion prevention.

1. Introduction

As more and more computers are linked together in networks users become more vulnerable than ever before to attacks on their computer systems. Most of these attacks exploit the same type of flaw: *buffer overflows* [15]. Therefore a lot of research has gone into preventing attempts to exploit vulnerabilities of this kind from succeeding. Several tools have been written with this aim in mind.

In 2003 Wilander and Kamkar published a comprehensive comparison of the available prevention tools and known attack variations at that time [1]. Since then new venues of attack have been described, new tools have emerged and old ones updated.

The purpose of this report is to determine how the updated prevention tools, and some entirely new ones, fare against current attack forms.

2. New Developments

Several operating system vendors have developed their own techniques for protection against buffer overflows. Examples are *Exec Shield* incorporated into Red Hat, and DEP included in Microsoft Windows XP. These protections are more ambitious than the individual tools tested by Wilander and Kamkar, in effect making these tools extraneous. Many of the solutions put forth by the authors of those tools, such as the use of *canaries* in *Stack Guard*, have become standard features.

3. Non-Executable Stack and Heap

Many attacks on vulnerable computer systems involve injecting binary code into the target's address space and diverting execution there. These attack methods would fail if no page in memory was simultaneously writable and executable. For instance none of the attack methods used by Wilander and Kamkar work on Fedora Core 2, which has incorporated this feature.

It is often possible to rework an exploit to use a technique known as *return into libc* instead, thereby obliterating the need for any code injection [7].

Therefore it could be argued that classifying attack methods as code-injecting or not would make more sense than distinguishing between heap- and stack-based attacks. Early attempts at countering code-injecting attacks stopped at stack protection[8], but for most programs the heap too can be made non-executable. There are exceptions, but they are rare.

Recent processors from AMD and Intel support marking virtual memory pages as executable or not. AMD calls their technology for *No Execute page protection* (NX) while Intel uses the name *Execute Disable bit* for this feature. Previously the read and execute bits were collapsed into one, making it infeasible to use this bit for non-executable stacks (and heaps) since it must be possible to read stack data. It is however possible to simulate NX even on older hardware using various tricks [9].

3.1 DEP

*Data Execution Prevention (*DEP*)* is a technique developed by Microsoft in cooperation with major CPU vendors. Microsoft's description of DEP is:

"(DEP) is a set of hardware and software technologies that perform additional checks on memory to help protect against malicious code exploits. In Windows

XP SP2, DEP is enforced by both hardware and software."

With hardware-enforced DEP all memory locations in the address space of a process are marked as non-executable unless the location explicitly contains executable code. The hardware implementation varies between vendors but if code is executed from a location that is marked as non-executable, an exception is raised. Since this is a very new technology and the exception that is raised, `STATUS_ACCESS_VIOLATION`, is a new one, most applications must be rewritten to be able to handle this. Not handling an exception of this type leads to program termination, if the exception is raised. [4]

The software-enforced part of DEP is an extension to the hardware part which protects against exploits of the exception-handling mechanism in Microsoft Windows. Unlike the hardware-enforced part, software-enforced DEP works on any hardware able to run Microsoft Windows XP SP2.

If the program being executed was built with *Safe Structured Exception Handling* (SafeSEH), software-enforced DEP ensures that before an exception is dispatched the exception handler is registered in the programs function table. If the program being executed was not built with SafeSEH, software-enforced DEP ensures that before an exception is dispatched the exception handler is located in a part of memory which is marked as executable [4].

3.2 Exec Shield

Exec Shield is a Linux kernel patch written by Red Hat which is used in Red Hat Enterprise Linux starting with version 3 update 3, as well as in Fedora (the community-supported version of Red Hat) Core 1 and later [5]. It works through emulation on older x86 processors but makes use of hardware mechanisms where available. The stack can be set to executable for individual binaries. It also contains some protection against return-into-libc attacks, which are covered in section 5.

3.3 Other Attempts

Another notable implementation is *PaX* [11], which is a set of patches to the Linux Kernel mainly written by an anonymous author who will not comment on his reasons for not divulging his name. Because of the uncertainty regarding its copyright that the mystery surrounding its principal author results in, PaX is not expected to be included into mainstream distributions. PaX also makes some trade-offs that are uncomfortable to vendors[12]. Another competing Linux Kernel patch comes from Solar Designer's *Openwall project*, but it seems more limited in scope and only protects the stack [13].

3.4 Visual C++ Compiler Security Checks

The Microsoft Visual C++ compiler has a built-in compiler flag helping programmers write more secure code. It is the `/RTC1` flag which is an alias for `/RTCsu` where *s* stands for stack checks and *u* stands for uninitialized variable checks. All buffers are tagged at the edges and therefore buffer overruns can be detected. The `/RTC1` flag is an excellent help in writing more secure code, but it only works for debug builds. It was first introduced in Microsoft Visual Studio 6 but since then needs for run-time checks in production code have also emerged. Therefore, starting with Microsoft Visual Studio .NET, a new mechanism was designed making programmers able to build programs in release mode with run-time defense against buffer overflows. This mechanism is also invoked by a compiler flag, namely the `/GC` flag.

The `/GC` flag introduces a *canary*, or *cookie* as Microsoft calls it, between the return address and local variables. This is a well-known technique used by other tools as well, including StackGuard [14] which was tested by Wilander and Kamkar [1]. For the interested reader Microsoft has published an in-depth article [6] about their compiler security checks.

4. New Attack Forms

As vendors adopted effective defenses against traditional buffer overflow exploits attackers refined their methods.

4.1 Return into Libc

By implementing a non-executable stack patch for the Linux kernel Solar Designer showed it was possible even for CPUs which were not designed to support it [8]. He also demonstrated a way, called return into libc, to circumvent his own protection mechanism [10]. The idea is to overwrite the return address with the address of a system function instead of an address to attacker-supplied code. No injection of code is necessary, the attacker only needs to know the address to a suitable system function and to be able to supply it with appropriate arguments. The minimal size of the payload thus shrinks, making it possible to fit into smaller buffers.

4.2 C++ Virtual Methods

With C++ and object orientation comes a new attack target: the table used to implement *virtual* method calls, called the *vtable*. Marking a method as virtual means it should be called by dynamic dispatch, i.e. the decision of which implementation of the method should be called is delayed until run time.

To implement dynamic dispatch a level of indirection is required. Ordinary method calls can be resolved at compile time using static binding, but for calls to virtual methods the address of the method in question must first be fetched from the vtable.

Every class with at least one virtual method has a vtable, which is hidden from the application programmer. Each object in turn has a pointer, called a *vptr*, into a vtable for every virtual method it supports. This pointer too is hidden from the application programmer. Space for these pointers is allocated together with the visible member variables of the object (figure 1). If it is possible to overflow a buffer in the object, and the compiler allocated space for the vptrs *after* the buffer (as in figure 1), then it may also be possible to overwrite the vptrs. Subsequent calls to virtual methods of this object can thus lead to execution of arbitrary code.

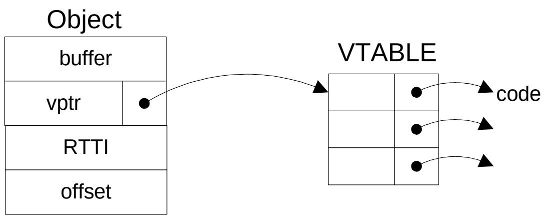

Figure 1 A single object of a virtual class, with a member variable called "buffer"

Rix described a method for exploiting this in Phrack Magazine issue 58[3]. He also noted that the compiler can prevent the vptrs from being overwritten by simply placing them *before* member variables in memory. This indeed mitigates the problem, but in cases where several objects are allocated in contiguous memory the problem persists, since it is then possible to overwrite the vptrs of an adjacent object instead. Figure 2 illustrates this idea.

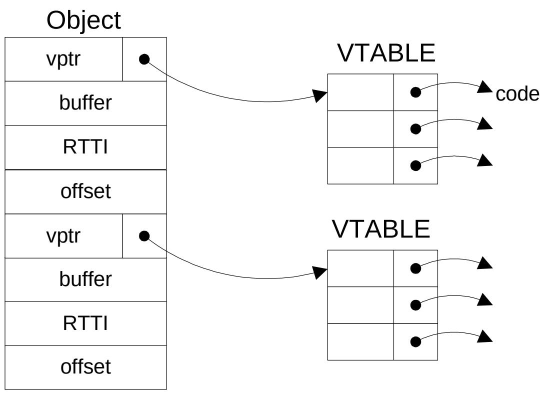

Figure 2 Objects of a virtual class allocated in contiguous memory. Here the location of the vptrs has been changed.

Furthermore, he observed the existence of four more hidden bytes allocated for every object, but did not investigate their use. This is understandable given the sparse documentation on the C++ ABI (Application Binary Interface) used by GCC. These bytes however contain run-time type information (RTTI) and an offset. Since these bytes can be overwritten too it would be interesting to see if doing so could lead to a new type of attack.

4.3 Exceptions

Exceptions and exception handling is quite a tricky area to deal with. When thinking about C++ and the try, catch, throw keywords, an exception is a software discovered error that when thrown transfers the control flow of the program to the innermost (matching) catch statement. But when hardware exceptions, such as a division by zero or segmentation fault, occur, it is up to the operating system to handle this.

In the heart of its core Microsoft Windows deals with exceptions according to the *Structured Exception Handling* (SEH) mechanism. There are several good things about SEH but there is also a flip-side to the same coin. SEH can be exploited!

In order to understand how a buffer overflow attack works on SEH one has to understand how SEH works.

Each executing thread in a Windows environment has something called a TIB (Thread Information Block). The first `DWORD` in that structure is a pointer to the thread's `EXCEPTION_REGISTRATION` structure. Located inside this structure is a pointer to the next `EXCEPTION_REGISTRATION` and also a pointer to an _except_handler callback function. Se figure 3 for an overview of this.

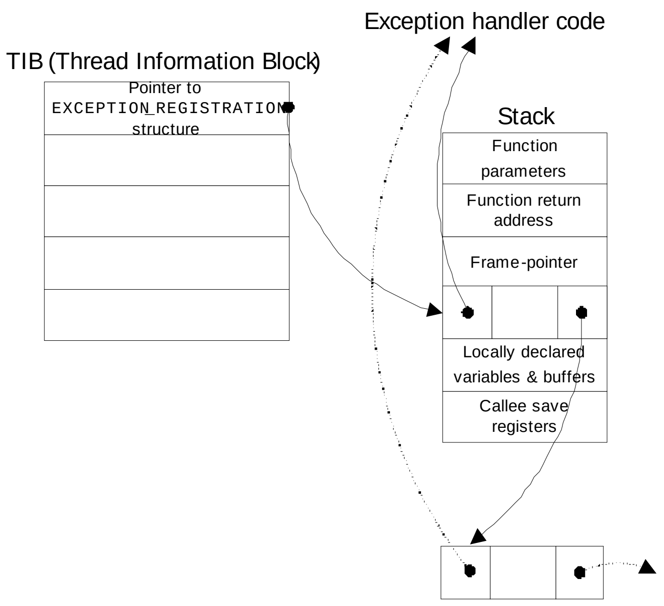
Exception handler code
TIB (Thread Information Block)
Pointer to EXCEPTION_REGISTRATION structure
Stack
Function parameters
Function return address
Frame-pointer
Locally declared variables & buffers
Callee save registers

Figure 3 This figure shows the EXCEPT_REGISTRATION structures' location on the stack and the TIB pointing to the first one in the linked list

This then constitutes a linked list. This is because an application seldom only has one exception handler, it has several, each taking care of one type of exception. When a thread executes something that leads to an exception the control is transferred to the operating system which then looks in the TIB for the pointer to the first EXCEPTION_REGISTRATION. The list is then traversed until the correct handler is found and then the flow of control is passed to the corresponding function.

When a Windows C++ compiler sees the keywords _try and _except in a function it generates code for creating an EXCEPTION_REGISTRATION structure and putting it on the stack. When the function is called during runtime the first thing that happens is that an EXCEPTION_REGISTRATION will be created and put onto the stack. A buffer overflow within this function can possibly overwrite the EXCEPTION_REGISTRATION structure and making it point to arbitrary code instead of the real exception callback. The attacker also must create an exception within this function so control can be passed to the attack code. The last part is easy, just overflow the buffer so much that an access violation exception occurs.

The SEH mechanism has been fixed in response to the above described attack and Windows XP with Service Pack 2 is believed to withstand all current SEH attacks [2]. Similar attacks on Linux might also be possible. The function-pointer clobbering discussed by Pincus and Baker has a very similar approach [2].

5. Further Protection Techniques

When non-executable stacks and heaps were introduced attackers responded with the return-into-libc technique. The response from the security community was to randomize addresses and relocate shared libraries to an area in memory which Red Hat calls the *ASCII Armor.*

5.1 Randomized Addresses

To be able to transfer control to their own code or to a system function attackers need to know the address of the place in memory where they want execution to continue. If the base addresses of the stack, heap and of shared libraries are randomly chosen upon startup of each process, or at some short time interval, the probability of a successful attack using return-into-libc is reduced greatly. The chance that an address that worked on the attacker's computer will also work on another computer, or even at the same machine at a different time, is very slim.

In order for an executable to be loaded at a random address it has to be compiled to *PIC* (Position Independent Code), which means it is relocatable, just like a shared library. Unfortunately compiling to PIC results in a small performance penalty, but even though this is true for shared libraries as well it still has not stopped programmers from doing most of their heavy lifting in libraries. Hence this trade-off can be considered reasonable.

5.2 ASCII Armor

The ASCII Armor is Red Hat's name for the lower 16 megabytes of virtual memory, to which all system libraries are relocated [5]. On Fedora Core 2 all system libraries are prelinked every 14 days, each time with random addresses containing zeros. The rationale for this is that since most overflows are due to insecure use of string-handling functions, exploiting typical vulnerabilities is made more difficult if all shared libraries have addresses with zeros in them. This is because strings are usually zero-terminated, meaning string functions such as strcpy() will stop when they encounter a zero. This feature is not unique to Red Hat, they just serve as an example because they gave it a descriptive name.

6. Results

We used Wilander and Kamkar's testbed with two additional test cases: the attack on Windows' exception handling mechanism, and the vptr attack. Naturally the exception attack was not tried on Linux, and vice versa with the vptr attack and Windows, as they are incompatible.

The results have been compiled in a table, however there is one caveat. The reader might be misled into believing Exec Shield is a panacea due to its impressive number of attacks prevented. This is an effect of limiting the tests to code-injecting attacks. The numbers would have been different had return-into-libc attacks been included. Unfortunately no computer equipped with hardware support for DEP was in our possession so therefore no test were committed using hardware enforced DEP.

	prevented	***missed***
Windows SP1	0 (0%)	19 (100%)
Windows SP2	1 (5%)	18 (95%)
Windows SP2 /GC	3 (16%)	16 (84%)
Windows DEP	N/A	N/A
Exec Shield	19 (100%)	0 (0%)
Basic Linux	0 (0%)	19 (100%)

7. Conclusions

Up until now exploiting buffer overflows has been easy. With the new tools presented in this report it is the belief of the authors that these kinds of attacks will be more difficult in the future. There are still many years left until all users have an effective protection system in their computer since Microsoft's DEP technology relies on hardware support. And until then most users will still be vulnerable to conventional buffer overflow attacks.

Buffer overflow exploits that do not use code injection will probably become more common since current prevention tools do not fully protect against this class of attacks.

References

[1] John Wilander and Mariam Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention", In *10th Network and Distributed System Security Symposium (NDSS'03)*, pp. 149-162, February 5-7, 2003, San Diego, California

[2] Jonathan Pincus and Brandon Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", *IEEE Security & Privacy*, IEEE Computer Society, July/August 2004, pp. 20-27.

[3] rix, "Smashing C++ VPTRS", Phrack Magazine 56 article 0x08 (May 2000), http://www.phrack.org/phrack/56/p56-0x08

[4] Starr Andersen, "Changes to Functionality in Microsoft Windows XP Service Pack 2 – Part 3: Memory Protection Technologies", Microsoft TechNet (August 2004), http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

[5] Arjan van de Ven, "Security Enhancements in Red Hat Enterprise Linux", http://people.redhat.com/drepper/nonselsec.pdf

[6]Brandon Bray, "Compiler Security Checks In Depth", Visual Studio Team, Microsoft Corporation (February 2002), http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchCompilerSecurityChecksInDepth.asp

[7] Nergal, "The advanced return-into-lib(c) exploits", Phrack Magazine 58 article 4, http://www.phrack.org/show.php?p=58&a=4

[8] Solar Designer, "Non-executable stack -- final Linux kernel patch", Linux Kernel mailing list 14 May 1997, http://www.ussg.iu.edu/hypermail/linux/kernel/9705.1/0493.html

[9] Arjan van de Ven, "New security Enhancements in Red Hat Enterprise Linux v.3, update 3", http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

[10] SolarDesigner, "Getting around non-executable stack (and fix)", *Bugtraq mailing list,* http://www.securityfocus.com/archive/1/7480 , Aug. 1997

[11] The PaX team, PaX, web site

[12] Ingo Molnar, "Thoughts on kernel security issues", http://lkml.org/lkml/2005/1/20/42

[13] README file for "Patch for Linux 2.4.30, version 1", http://www.openwall.com/linux/

[14] Immunix, "StackGuard: Protecting Systems from Stack Smashing Attacks", http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/

[15] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. "A first step towards automated detection of buffer overrun vulner-abilities", In *Proceedings of Network and Distributed Sys-tem Security Symposiu*m, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.