Buffer Overflow Attacks Defending Using A Segment-based Approach

Wei-Zhi YANG
College of Information Science and Engineering, Northeastern University
Shenyang, Liaoning, 110004, China
yweizhi@sina.com

Yu-An Tan
Department of Computer Science and Engineering, Beijing Institute of Technology
Beijing 100081, China
phd200801@126.com

Abstract

A segment-based non-executable stack approach is proposed and evaluated to defend against stack-based buffer overflow attacks under Windows NT/2000 /2003/XP and Intel 32-bit CPUs. A kernel device driver is designed to relocate the application's user-mode stack to the higher address and to modify the effective limit in the code segment descriptor, in order to exclude the relocated stack from the code segment. Once any code that attempts to execute the malicious code residing in the stack, a general-protection exception of exceeding the segment limit is triggered so the malicious code will be terminated. It is highly effective in preventing both known and yet unknown stack smashing attacks and its performance overhead is lower than the page-based non-executable stack approach.

1. Introduction

As the Internet explosive growth, this has brought an increase in remote systems being compromised by malicious attacks. According to CERT/CC [1-3], approximately half of all the security alerts were based on buffer overflow vulnerabilities. In recent years, there have been a large number of increase buffer overflow vulnerabilities being discovered and exploited. Some of the best-known examples were the Morris Worm in 1988, Code Red worm in 2001, Blaster Worm in 2003. More recently, in May of 2004, the Sasser Worm exploited remote buffer overflow vulnerability in the Windows LSA (Local Security Authority) Service. An unauthenticated attacker could exploit this vulnerability to execute arbitrary code with system-level privileges on Windows 2000 and Windows XP machines [4-7].

Buffer Overflows are one of the most common and potentially deadly forms of attacks against computer systems to date [8]. They allow an attacker to locally or remotely inject malicious code into a system and compromise its security. There are several types of buffer overflow, including stack based overflows, heap-based overflows, return-into-libc. The vast majority of buffer overflow attacks are stack based, which overwrite a function's return address on the stack to point to exploit code. Thus, protecting stacks from execution prevents most attacks. Although, various solutions have been proposed to tackle the problem, buffer overflow attacks still dominate. This is mainly due to two reasons: (1) thousands of existing applications (legacy code) are still being used and many of them are vulnerable to buffer overflow attacks; (2) many proposed solutions incur undesirable performance overheads and not ease to use. As a result, users are reluctant to patch their software or harden their operating system.

This paper proposes an architectural solution for run-time protection against buffer overflow attacks: Stack Segment Limitation (SSLimit). As far as we knew, it is the first segment-based solution of non-executable stack under Windows operation system to defense against stack smashing attacks. It is a kernel mode driver for Windows to utilize the processor's feature of checking segment limits, which prevent malicious code stored in stack from seizing the control of the process. It is effective in protecting the application for stack-based overflows, while preserving slight performance penalty, ease of deployment and highly effective in preventing both known and yet unknown stack smashing attacks.

2. The Proposed Approach

Our segment-based approach to prevent execution of malicious code utilizes the segmentation logic of IA-32 to halt the task by generate a general-protection exception. This allows us to ignore the complexities of various vulnerabilities and the difficulties in preventing the stack smashing attacks.

2.1. Utilizing the Segment Limit Check

The memory management facilities of the Intel Architecture are divided into two parts: segmentation and paging. Segmentation provides a mechanism of isolating individual code, data, and stack modules. The limit field of a segment descriptor prevents programs or procedures from addressing memory locations outside the segment. For code segments and other types of segments except expand-down data segments, the effective limit is the last address that is allowed to be accessed in the segment, which is one less than the size, in bytes, of the segment. The processor causes a general-protection exception any time an attempt is made to access the byte at an offset greater than the effective limit.

Windows NT/2000/2003/XP works under the memory model called "flat model", by creating 0 based and 4 GB limited segments for both code and data(include stack) accesses. For all applications, CS (code segment selector), SS (stack segment selector), DS (data segment selector) register of CPU are initialized with separate segment descriptors all pointing to same linear address.

A fundamental reason for the stack buffer overflow is that the stack is executable, so the malicious code injected into the stack can be executed just like the valid code. To make the victim program's stack region non-executable, we relocate the stack region and modify the effective limit in code segment descriptor to exclude the stack region from valid code region, as shown in Figure 1. Supposed that U is the highest address of the user-mode applications' code region, the stack is relocated to the location higher than U, then the effective limit of code segment descriptor is set as U. Therefore, once the malicious code residing in the stack seizes the control of the process, a general-protection exception is generated due to exceeding the segment limit; the hijacking attempt will be terminated.

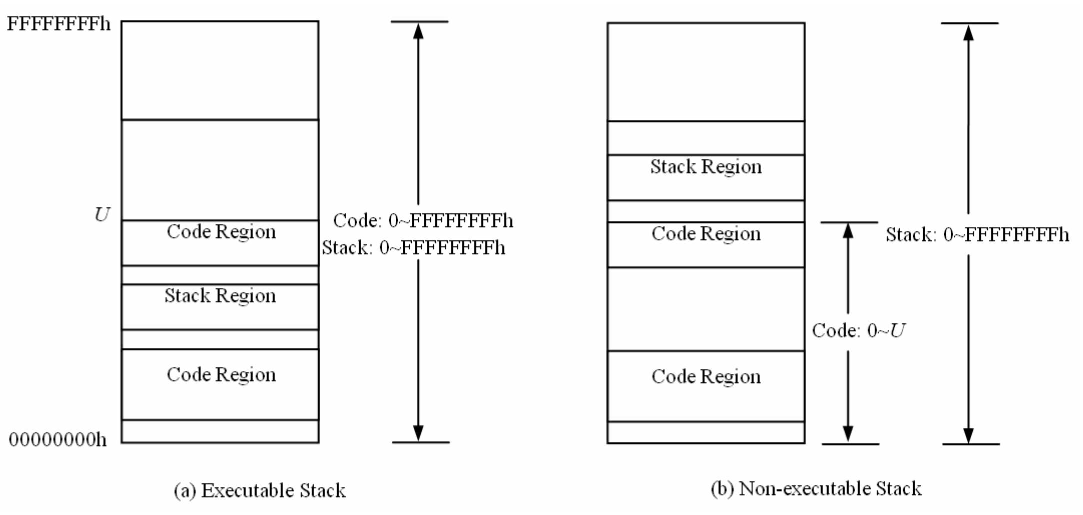

Figure 1. Code/Stack region layout

2.2. Hooking Windows System Services

The kernel component NTOSKRNL.EXE implements hundreds of system services that perform operations specific to the application programming interfaces (API). The system services represent the fundamental interface for any user-mode application to the kernel. This enables the developer to hook the system services and modify the system behavior to suit their needs.

At its startup, SSLimit hooks these two Windows system services: ZwRegisterThreadTerminatePort () and ZwSetInformationThread (). The former one is invoked after the creation of a process while the latter is invoked after the creation of a thread. It locates the system service dispatch table (SSDT) used by the operating system and changes the function pointers to point to some other functions inserted by the developer.

2.3. Relocating Stack Function (RSF)

Whenever a process or thread is created, the system reserves a region as its user-mode stack and also commits some physical memory pages to this reserved region. Our basic idea is to relocate the stack region to the higher address as shown in Fig. 1; also the user-mode stack pointer register must be modified to point to the new stack region. RSF includes the following 5 steps for this purpose.

STEP 1: Get the information of user-mode stack. The thread's user-mode stack information is stored in its thread environment block (TEB) structure, such as the topmost address of the thread's stack, the linear address of the lowest committed page in thread's user-mode stack, and the linear address for the process database (PEB) which can be used to get the reserved stack size and the committed stack size.

STEP 2: Allocate a new stack region. ZwAllocate-VirtualMemory () is invoked to allocate a new stack region by specifying MEM_TOP_DOWN option in its parameter. If the allocation succeeds and the address of allocated memory pages is higher than U, the new stack region is valid, or the relocation aborts and this thread is beyond of protection of SSLimit. The new stack region has the same reserved stack size and committed stack size as the original one.

STEP 3: Modify the user-mode stack pointer. Only two privilege levels are used in Windows, privilege level 0 is the kernel mode (ring 0) and privilege level 3 is the user mode (ring 3). Each thread defines two stacks, when the thread running at ring 3, it uses user-mode stack; while at ring 0, it uses kernel-mode stack. The user-mode stack pointer is saved in the kernel-mode stack when the thread switches from ring 3 to ring 0, and it is restored at switching from ring 0 to ring 3. As shown in Figure 2, SS:ESP is the user-mode stack pointer saved in kernel mode stack, we need to modify ESP to point to the new stack region. The correlative field in TEB is also updated to reflect the stack movement.

STEP 4: Copy the stack's content and adjust the frame pointers. The original stack's content is copied to the new stack region, then the stack frame pointers in the new stack is also be adjusted accordingly.

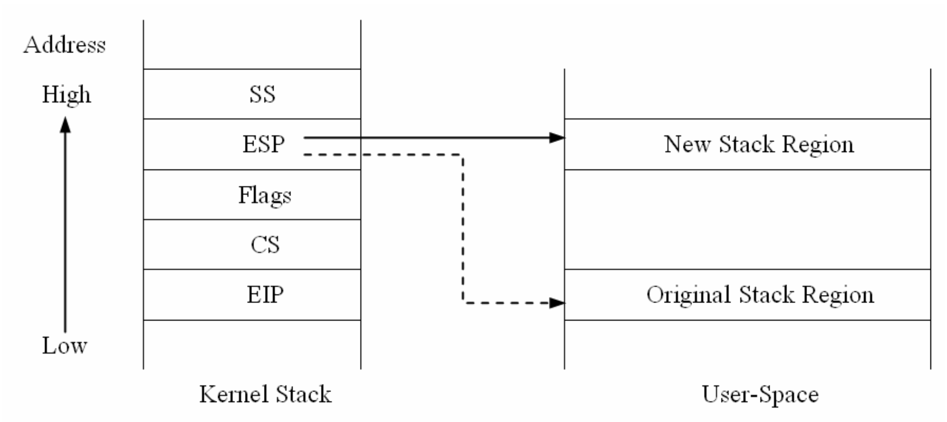

Figure 2. Modify the user-mode stack pointer

STEP 5: Free the original stack region. At last, zwFreeVirtualMemory () is invoked to free the virtual memory pages of the original stack region.

3. Experiments and Results

SSLimit is a kernel driver to relocate the stack and implement a non-executable stack in Windows. It intercepts ZwRegisterThreadTerminatePort () and ZwSetInforma-tionThread () in order to inject RSF into the invocation paths of these two system ser-vices. After the process or thread is created, RSF is invoked to relocate its stack region, so the stack is put in an area of memory that does not allow execution. SSLimit prevents both known and yet unknown stack-based buffer overflow attacks. In case the malicious code in the stack is executed, the general protection will be triggered and the application will abort.

3.1. Effectiveness

The effectiveness of the proposed segment-based approach is evaluated by the several known buffer overflow exploits with recently reported security vulnerabilities. We tested 3 known exploits (SQL Slammer, Blaster and Sasser) against a system guarded by the SSLimit, without the installation of patches for these exploits. These exploits abort the applications but never hijack them successfully.

In Linux, GCC generates trampolines on the stack to pass control to nested functions when calling from outside. This requires the stack being executable, so a special workaround is required for the non-executable stack solution. Fortunately, we found that GCC complier in Windows doesn't utilize the feature of dynamic code in the stack, so we don't need the work around for the GCC trampolines.

3.2. Performance

SSLimit's performance penalty is the execution time of RSF (Relocating Stack Function) invoked in case of the process or thread creation, this incurs a slight performance overhead. To quantify the performance overhead of SSLimit, we measure the execution times of five real-world applications and compare the results with SSLimit and OverflowGuard. OverflowGuard is a non-executable stack solution in Windows but used the page-based techniques instead of segment-based ones. Table 1 shows the mean execution times for each of these applications in 3 cases: (1) the original application (i.e., without SSLimit and OverflowGuard), (2) SSLimit installed, and (3) OverflowGuard installed. The execution times are based on 100 runs and are given seconds. All experiments were conducted on a 233 MHz Pentium machine with 256 MB of memory running Windows 2000 professional version.

The average performance degradation of 5 applications with SSLimit and OverflowGuard is approximately 0.445% and 2.135%. The test result is encouraging and it proves that the overhead of SSLimit is negligible. SSLimit is a cost-effective method to defense stack smashing attacks and it has only minimal overhead for overall system performance. OverflowGuard and the other page-based solutions have much higher performance penalty, because they need to monitor the allocation and free of all user-mode memory pages, which leads to a large amount of CPU cycles.

4. Conclusions

This paper proposes and evaluates a segment-based approach for preventing stack smashing attacks for Windows. It implements the non-executable stack while preserving slight performance penalty, without re-compilation and highly effective in preventing both known and yet unknown stack smashing attacks.

Table 1. Comparison of execution time and overhead between SSLimit and OverflowGuard

Applications	Normal	SSLimit		OverflowGuard	
	Time(s)	Time(s)	Overhead	Time(s)	Overhead
UlraEdit	2.553	2.563	0.392%	2.627	2.896%
Visual C++	3.104	3.131	0.867%	3.190	2.771%
WinAmp	2.243	2.248	0.223%	2.293	2.229%
WinWord	3.617	3.639	0.608%	3.659	1.162%
WinRar	0.680	0.681	0.147%	0.691	1.618%
(Average)	-	-	0.445%	-	2.135%

References

[1] S.Q. CHEN, B. SHEN, S. WEE, et al., “Segment-based streaming media proxy: Modeling and optimization,” *IEEE Transactions on Multimedia*, vol. 8, pp. 243-256, 2006.

[2] L.T. NOVAK, C.C. CHEN, Y.H. SONG, “Segment-based eyring-NRTL viscosity model for mixtures containing polymers,” *Industrial and Engineering Chemistry Research*, vol. 43, pp. 6231-6237, 2004.

[3] R. SADEGHI, “Segment-based Eyring-Wilson viscosity model for polymer solutions,” *Journal of Chemical Thermodynamics*, vol. 37, pp. 445-448, 2005.

[4] S.Q. CHEN, H.N. WANG, X.D. ZHANG, et al., “Segment-based proxy caching for Internet streaming media delivery,” *IEEE Multimedia*, vol. 12, pp. 59-67, 2005.

[5] R. SADEGHI, “Extension of the segment-based Wilson and NRTL models for correlation of excess molar enthalpies of polymer solutions,” *Journal of Chemical Thermodynamics*, vol. 37, pp. 1013-1018, 2005.

[6] R. SADEGHI, M.T. ZAFARANI-MOATTAR, A. SALABAT, “Density modeling of polymer solutions with extended segment-based local composition nonrandom two-liquid (NRTL), Wilson, and nonrandom factor (NRF) models,” *Industrial and Engineering Chemistry Research*, vol. 45, pp. 2156-2162, 2006.

[7] M.L. SHANG, H. HUANG, “Buffer overflows attacks analysis and real-time detection,” *Computer Engineering*, vol. 31, pp. 36-38, 2006.

[8] L.D. PAULSON, “New chips stop buffer overflow attacks,” *Computer*, vol. 37, pp. 28, 2004.