Computer Standards & Interfaces 38 (2015) 17–24

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

An enhancement of return address stack for security

Chien-Ming Chen [a], Shaui-Min Chen [b], Wei-Chih Ting [c], Chi-Yi Kao [b], Hung-Min Sun [b,*]

[a] *School of Computer Science and Technology, Shenzhen Graduate School, Harbin Institute of Technology, Shenzhen, China*
[b] *Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC*
[c] *Industrial Technology Research Institute, Hsinchu, Taiwan, ROC*

ARTICLE INFO

Article history:
Received 20 August 2012
Accepted 16 August 2014
Available online 27 August 2014

Keywords:
Stack smashing attack
Buffer overflow
Memory pointer corruption attack
Return address stack
Binary rewriting

ABSTRACT

Stack smashing is one of the most popular techniques for hijacking program controls. Various techniques have been proposed, but most techniques need to alter compilers or require hardware support, and only few of them are developed for Windows. In this paper, we design a *Secure Return Address Stack* to defeat stack smashing attacks on Windows. Our approach does not need source code and hardware support. We also extend our approach to instrument a DLL, a multi-thread application, and DLLs used by multi-thread applications. Benchmark *GnuWin32* shows that the relative performance overhead of our approach is only between 3.47% and 8.59%.

1. Introduction

With the rapid growth of computer system, more and more issues have been concerned. One of the most concerned issues is security. Stack smashing attacks, which exploit buffer overflow vulnerabilities [18,12,31,7,30] to hijack the program control from attacked applications, are the most widely used type of attacks. Due to careless programmers, vulnerabilities [28,29] exist all the time. Diverse techniques have been proposed to thwart stack smashing attacks, such as static analysis and dynamic detection. However, they are not as useful as we thought because most of them must alter compilers [34,33,10,8,11,22,32,1,20] and recompile source code, or require hardware support [35,34,13,9] to execute specialized instructions. Another reason is that most of them are built only for Linux. However, Windows is still the most popular operating system today, and there are more applications that contain buffer overflow vulnerabilities. Therefore, we need to protect those applications from stack smashing attacks on Windows.

There is a class of techniques [26,34,23,8,25,2,32] which creates a safe area to backup return addresses to prevent stack smashing attacks. The safe area is called private stack, canary stack, or return address repository, etc. For consistency, we call it return address stack throughout this paper. These techniques revise the prologue and epilogue of each protected function. The revised function prologue will store copies of the return address into return address stack, and the revised function epilogue will restore the return address on stack with copies of them. Obviously, these techniques should guarantee that the return address stack is absolutely secure because attackers may attempt to modify the contents of the return address stack to hijack the program control. In general, these techniques set return address stack as read-only mode most of the time to protect it. The only situation that the return address stack becomes writable is in the revised function prologue when it is pushed into return address stack. In this paper, all we considered is outside the box by showing that only setting return address stack as read-only is not secure enough if the return address stack is dynamically allocated.

Our approach is based on *Binary Rewriting*, so we can protect applications from stack smashing attacks without source code and hardware support. Hence, we only focus on those techniques that use *Binary Rewriting*. We classify them into two groups according to the way they allocate return address stack: (1) static allocation [26,23] and (2) dynamic allocation [25,2]. The first group statically allocates a return address stack like adding a new section called return address stack into Portable Executable (PE) or Executable and Linking Format (ELF) file. Therefore, the return address stack is already created before running the protected application. The second group dynamically allocates a return address stack at the beginning of the protected application. In this way, the return address stack is certainly located in the heap. We discovered that there is a potential security risk if the return address stack is dynamically allocated. Because the second group dynamically allocates an area to be the return address stack, it must have an *Entry Pointer* of the return address stack in order to pass the address to the revised function prologue and epilogue of each protected function. However, the second group only keeps eyes on protecting the return address stack but fail to protect *Entry Pointer* of the return address stack. For this reason, we can launch a *Memory Pointer Corruption Attack* to hijack the program control from protected applications by modifying *Entry Pointer* of the return address stack even if they are under

* Corresponding author at: No. 101, Section 2, Kuang-Fu Road, Hsinchu 30013, Taiwan, ROC. Tel.: +886 3 5715131x42968; fax: +886 3 5723694.

http://dx.doi.org/10.1016/j.csi.2014.08.008
0920-5489/© 2014 Elsevier B.V. All rights reserved.

protection. The detail of *Memory Pointer Corruption Attack* will be explained in the next section.

In this paper, we design a *Secure Return Address Stack* to protect the return address stack and entry pointer of the return address attack from stack smashing attacks on Windows. Our approach does not need source code and hardware support because we combine *DLL Injection* with *Dynamic Binary Rewriting* to implement it. Moreover, we also extend our approach to instrument a DLL, a multi-thread application, and DLLs used by multi-thread applications. Benchmark *GnuWin32* shows that the relative performance overhead of our approach is only between 3.47% and 8.59%.

The rest of this paper is organized as follows. In Section 3, we describe our approach and implementation. We evaluate our approach in Section 4. Section 5 describes a drawback, a limitation, and two potential security issues for instrumenting a multi-threading application. In Section 6, we survey related works of stack smashing attacks, and a conclusion will be given in Section 7.

2. Background

In this section, we review the stack smashing attacks and the memory pointer corruption attacks.

2.1. Stack smashing attacks

To understand stack smashing attacks, Fig. 1 shows a typical example. The C programming language is a popular language due to its high execution efficiency, but it has a lot of unsafe functions. For example, *strcpy*( ) is one of them because it does not check boundary automatically. Lack of boundary checking during a buffer copy operation may cause areas adjacent to the buffer be overwritten. As shown in Fig. 1, *msg* points to a sequence of attack data that are inputted by attackers, and these attack data have more than eighty bytes, including shellcode. After executing *strcpy*( ), all attack data will be copied into stack because *strcpy*( ) does not check the boundary of buf and *msg*. At this time, *buf* and *return address* are already overwritten by attack data. Therefore, the program control of this attacked program is transferred to the shellcode when *ret* instruction in the victim function is executed. This is a generic stack smashing attack that involves exploiting such an unsafe copy to overwrite the return address on stack with the address of shellcode.

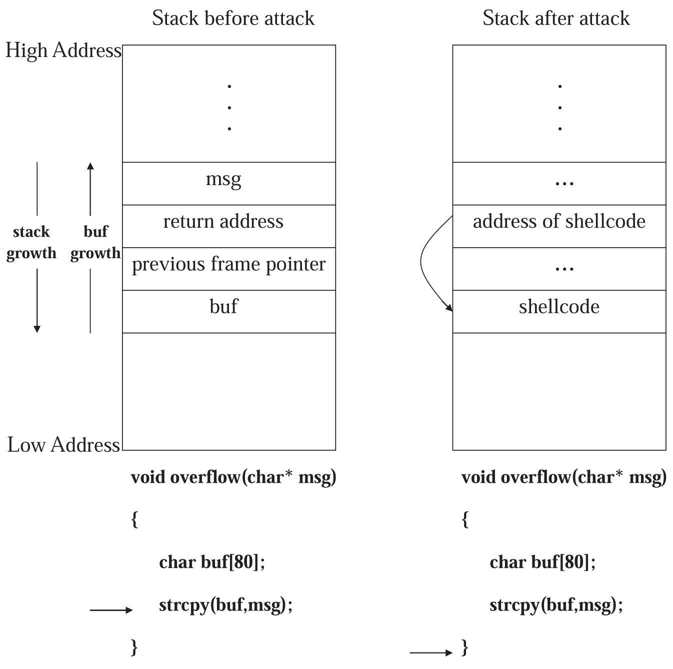

Fig. 1. Stack smashing attack.

2.2. Memory Pointer Corruption Attack

Memory Pointer Corruption Attack [4,27], we call it MPC attack for short in this section, is a variety of stack smashing attacks. Fig. 2 is a pattern of vulnerable function. If there is an application which has a function like the one in Fig. 2, attackers will have a chance to launch MPC attack. In real world, there is a ftp server called *wu-ftp*, version 2.5, which has a function called *mapped_path*( ) like Fig. 2. Thus, MPC attack is not an imaginary problem. Hence, we will use Fig. 2 as an example to demonstrate how to manipulate a MPC attack.

msg is an input of vulnerable function, and it points to a sequence of data which are inputted by attackers. The sequences of data include three main parts as follows.

1. Attack data.
   Attack data will be used to rewrite the content of attack target.
2. Attack target.
   Attack target is an address which is assigned by attackers.
3. Shellcode and other data.

After executing the first *strcpy*( ), the contents of *buf* are rewritten by attack data, and the content of *msg* is overwritten by attack target. Besides, shellcode and other data are also copied into stack. At this moment, titmsg is already changed to point to the attack target. Then, attack data will directly rewrite the content of attack target after executing the second *strcpy*( ). Therefore, we are able to rewrite anything on anywhere of memory via controlling the input *msg*. There is a very good sentence of Phrack Magazine [4] to appropriately describe MPC attack: "When a buffer overwrites a pointer… The story of a restless mind".

Previous approaches [25,2] always considered that return address stack is absolutely secure if it is set to be read-only. Then, attackers are hard to modify the contents of return address stack even if they use MPC attack. We also believe that return address stack is secure enough, but modifying the contents of return address stack is not the only way to hijack the program control from protected applications. Previous approaches dynamically allocate an area to be the return address stack when the protected application starts to run, so they must have an *Entry Pointer* of the return address stack, we call it *Entry Pointer* for short in this section, to be referenced by all protected functions. *Entry Pointer* is as important as a key of encryption, but it is not protected by previous approaches in any way because they only care of protecting the return address stack. Therefore, attackers are able to rewrite *Entry Pointer* to hijack the program control from protected applications that are protected by previous approaches via manipulating MPC attack.

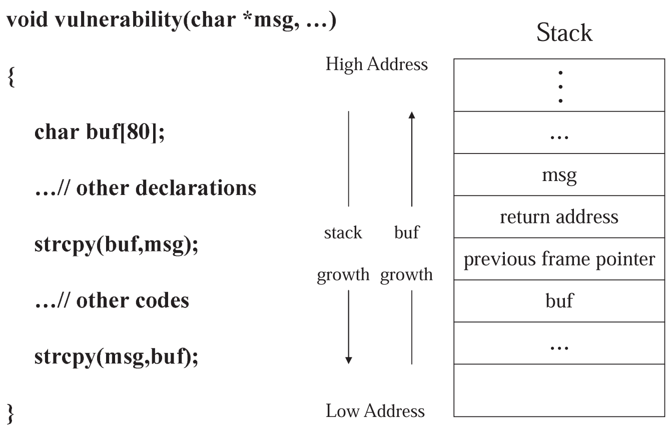

Fig. 2. Vulnerable function.