國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

程式失控動態分析系統設計與實作

The Design and Implementation of a Dynamic Instrument Tool for Program Crash Analysis

研 究 生: 劉世弘

指導教授: 黃世昆 博士

中華民國九十三年六月

程式失控動態分析系統設計與實作

The Design and Implementation of a Dynamic Instrument Tool for Program Crash Analysis

研 究 生：劉世弘　　Student: Shih-Hung Liu

指導教授：黃世昆 博士　　Advisor: Dr. Shih-Kun Huang

國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

A Thesis

Submitted to

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

In Partial Fulfillment of the Requirements

For the Degree of

Master

In

Computer Science and Information Engineering

June 2004

Hsinchu Taiwan, Republic of China

中華民國九十三年六月

程式失控動態分析系統設計與實作

研究生 : 劉世弘　　　　　　　　指導教授:黃世昆 博士

國立交通大學資訊工程學系（研究所）碩士班

摘要

為了符合市場快速開發的特性，發行後的軟體系統常發生未預期的錯誤。有些錯誤可能導致軟體失控，甚或產生安全弱點。一般現成的商業軟體（Commercial Off-The-Shelf, COTS）都沒有附原始碼，若軟體發生失控，我們能做的就是回報給開發此軟體的廠商，並等待他們的修補（patch）。然而軟體廠商常延誤多時才推出修補程式，有些修補程式甚至與舊有的軟體版本不相容，未能完全修復錯誤。針對現有商業軟體元件，一般仍使用反向工程（Reverse Engineering）工具進行測試與觀察軟體執行行為，以判斷該軟體是否存在可能遭入侵的弱點。本研究的目標在於設計系統、協助判斷程式失控點是否隱藏可被運用的軟體漏洞。我們希望此系統能提供系統化的程式失控分析。

目前已有許多研究著力於偵測程式錯誤並指出錯誤形成的原因，有些是透過靜態程式碼分析或動態觀測程式執行過程來進行分析，而大部分的研究採用的方法是稽核或修改程式原始碼，以達到觀察的目的。然而由於本研究是針對現成的商用軟體，沒有原始碼可供分析，我們因此發展一個實驗與攔截（instrument and interception）的系統，能夠偵測軟體異常執行流程，並判斷是否可能成為安全上的漏洞。本研究發展堆疊錯誤點偵測、逼近(stack corrupt site approximation and identification）與呼叫目標確認（call target validation）兩種機制去偵測程式的執行流程是否發生異常。透過對微軟視窗（Microsoft Windows）平台上商業軟體的實驗，對現有多種弱點都能有效偵測，並經由攔截狀況分析中瞭解產生異常的原因。此實驗也證實錯誤點偵測機制能指出導致堆疊異常的函式。最後我們與相關工具比較，以評估系統的可行性。

The Design and Implementation of a Dynamic Instrument Tool for Program Crash Analysis

Student: Shih-Hung Liu Advisor: Shih-Kun Huang

Department (Institute) of Computer Science and Information Engineering
National Chaio Tung University

Abstract

In order to meet time to market, software often releases with unintended flaws. Some cause software crashes that are highly related to security vulnerabilities. Commercial Off-The-Shelf (COTS) software normally comes without source code. If there happened any program crash, all we can do is to report it to the vendor and wait for the patch. Some software companies, however, develop their patch not in timely manner, or even no longer support the older version. Normally, intended users can use debuggers to observe the running behavior of the software and determine if there exists any vulnerability to exploit. Our objective is to design a tool that helps systematically detect security-related errors from the crash. We want to automate the process to a certain extent for crash analysis.

Much research work focused on detecting program errors and identifying their root causes either by static analysis or observing their running behavior through dynamic program instrument. Much of the work analyzes or instruments the source code of the software. However, with the assumption of lack of the source code, we develop an execution instrument and interception system and add detection mechanism of anomaly control flow inside to automatically judge if a certain crash can be exploited. We develop stack corrupt site identification and call target validation to detect if the control flow of the program is changed abnormally. Case studies of several commercial Windows applications from known exploits have proved the applicability of our system and better understanding of the exploiting path of these vulnerabilities. It manifests that our corrupt site identification mechanism points out the vulnerable function where the stack is polluted. At last, we compare this work with several related work to manifest the evaluation in the recent research.

Keywords: Dynamic Analysis, Software Wrapper, COTS Vulnerability Testing

誌謝

首先，要感謝我的指導教授黃世昆博士，謝謝老師這些日子以來的細心指導，給予許多寶貴的意見與想法，使得這篇論文得以完成；並感謝老師在生活與做事上的觀念分享，讓我明白每個人在一個團隊中要盡心盡力地扮演好自己的角色，才能使這個團隊卓壯與進步；並且要感謝中研院的梁德容教授，他的思路與邏輯常常讓問題更加清楚。再來要感謝中研院軟體方法實驗室的黃舜溫學長以及博士班的蔡昌憲學長，在做研究的這段過程裡給與我技術上與心態上的建議與教導，在我鬆懈的時候給我提醒，在我茫然的時候給我意見，讓我得以完成這篇論文，非常謝謝他們。接著要謝謝中研院軟體方法實驗室的其他學長，有袁勤國學長、蔡和諺學長、洪偉能學長、陳宗裕學長、林宗伯學長與陳寬達學長，給我在研究生這段期間生活上與技術上的幫忙與指教，讓我更加順利。最後要謝謝我的祖母與父母給我的支持與鼓勵，容忍我的任性與脾氣，感謝他們長久以來對我所做的一切；也謝謝宛真的陪伴以及給我的叮嚀與嘮叨，得以完成這篇論文。最後祝福我所有的朋友同學平安快樂。

劉世弘謹誌

民國九十三年六月

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Program running behavior has much to do with software security. Especially, crashed software may be exploited to be a potential vulnerability. It is difficult to reconstruct system failures after a program has crashed and much research effort has been taken on detecting program errors and identifying their root causes either by static analysis or observing their running behavior through dynamic program instrument. In order to meet the time to market, software releases with unintended flaws. Some of them cause software crash, while others may introduce security vulnerabilities. Our goal is to design a tool that helps analyze the program running behavior and determine if it is an exploitable vulnerability. We try to intercept and monitor running behaviors during programs in execution when only COTS (Commercial Off-The-Shelf) executables available for analysis.

1.2 Background

The purpose of this work is to automatically detect security-related errors from the crash. First of all, we must explain what our so-called "exploitable crash" is. Actually, this concept is much similar to exploiting the vulnerabilities in the programs. Buffer overflow vulnerabilities dominate the security attacks in recent years because it provides the attackers with exactly what they need, the space to inject payload and the change of control flow of the program. In this section, two common kinds of buffer overflow crashes will be explained, including how they are produced, how the attackers can exploit them to alter the control flow and how they can be detected.

We cannot generate enough testings to prove inexistence of all bugs. The problem such as `strcpy`, which will be shown in the following, is well known but exists so far. Although we can swap `strcpy` with `strncpy`, it still cannot make our code secure. There still exists many other vulnerable function such as `wsprintf()` or `sprintf()` to manipulate the string operations.

1.2.1 Stack-based Overflow Crash

Stack overflow is a programming error that might cause the program to be controlled by malicious attackers [17]. When the execution control of the program is taken, the attacker can do anything with the privilege of this program. This programming error is a notorious attack problem for years.

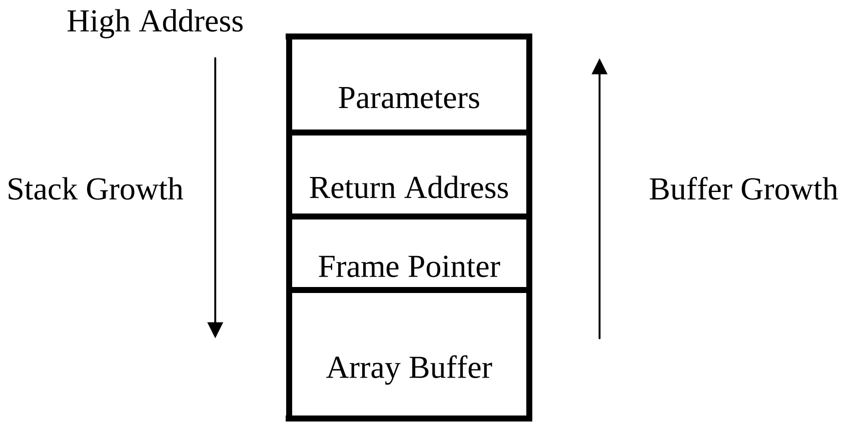


Figure 1 An activation record on the stack

In high-level languages, the parameters passed to the function, the local variables declared in the function, and the return address of the function will be allocated on the stack. Figure 1 shows the arrangement of these elements in an activation record (a logical stack frame corresponding to a function) on the stack. Notice that the stack grows from high memory address to low memory address.

When a function is called, the parameters passed to the function will be pushed on the stack first. Moreover, the order of the parameters to be pushed on the stack depends on the various calling syntax. Then the “CALL” instruction will be executed by pushing the address of the next instruction after “CALL”, i.e. return address, on the stack. Afterwards, the frame pointer stored in the current EBP will be pushed on the stack. The content of this frame pointer is the address of the previous stack frame and will be used to restore the previous stack frame information at the function exit. The last element to be allocated on the stack is the local variable declared in this function. When an array buffer is declared in this function, it will also be placed here. If the function copies an overlong string without bound checking, the stack overflow occurs. Because of the order of the buffer growth, the overlong string could overwrite the values of the frame pointer and the return address, which may result in the control flow corruption. The purpose of this work is to detect crashes due to frame pointer and return address overwriting because this vulnerability will be exploited by careful manipulation.

Figure 2 is a simple example to show the stack overflow crash. We can obviously see that the return address overwritten alters the control flow.

```
/* This is a program to show the stack overflow crash. It is intended to
overwrite the frame pointer and the return address of the function func
with 0x61616161, which is 'aaaa' in ASCII. */

#include <stdio.h>
#include <string.h>

int func(void)
{
    char str[8];      /* declare a character buffer in func */
    strcpy(str, "aaaaaaaaaaaaaaaaaaaa");    /* copy an overlong string
                                              to this buffer */
    return 0;     /* func will not return correctly because of stack corruption */
}
int main(void)
{
    func();      /* call the vulnerable function */
    printf("Should not see this line.\n");   /* execution flow is altered, so this
                                                 message will not be seen */
    return 0;
}
```

Figure 2 A sample program to show the stack overflow crash

1.2.2 Heap-based Overflow Crash

Based on the overwritten of the return address and saved base pointer on the stack, the execution flow can be changed. Heap is the memory region for dynamic data allocated at runtime. Heap-based overflow can also change the execution flow of the program and is more complicated than stack-based overflow. We will introduce how the heap-based overflow overwrites the destination of CALL instruction to change the execution flow and causes programs to crash.

The basic method is to pollute a function pointer by the neighboring buffer. Memory objects are often on the heap and through these objects, we can overwrite the pointer of virtual function table [20].

1.3 Objective

We will develop an execution instrument and interception system. This system will instrument the process and perform the detection mechanism to analyze whether the crash is exploitable or not. Usually, the intended attacker does not have source code of the program and uses the debugger to judge whether the program is exploitable or not. Therefore, our instrument process should not need to access the source code. This system will monitor the running behavior of the program when the execution meets the point we instrument. For stack-based overflow crash, we will instrument the prologue and epilogue of the API call and the user function call to record the stack evolvement. By verifying the integrity of the stack-trace made in the prologue and epilogue, we can determine that the stack is corrupted or not and point out the corrupt site of the program. For heap-based overflow crash, we will instrument all the CALL instruction whose destination is determined at runtime. When this type of CALL is executed, we will validate the legality of its target. We want to perform sort of input tracing to find out the path through which the corrupt site is deduced. Through memory-related API call wrapper, we design a data structure to save buffer size and address information. When the stack corrupt site is identified, we can traverse the data structure to find out why the corrupt site occurs due to some specific input data.

With this tool, we don't need to step by step execute the program to observe the crash process of a certain crash and judge its exploitability. It will detect two kinds of control state anomaly: saved base pointer / return address and function pointer corruption. We will evaluate our tool with actual known vulnerabilities.

1.4 Contribution

The contribution of this thesis is as following:

(1) The control flow anomaly detection mechanisms, such as stack corrupt site identification and call target validation, are presented. By using the debugger to analyze a crash, testers have to perform something like single-step executions and observe the running behavior of the program such as stack evolvement with core dump. However, core dump is produced at crash site and the stack has already been corrupted. Therefore, this debugging process is tedious and important actions may be hidden from being observed, such as functions that were called but silently returned before the crash. The mechanisms we proposed serve as measures to automate the process of crash analysis related to the security errors.

(2) Our stack corrupt site identification mechanism is helpful to understand why a certain stack-based crash occurs. If the distance between the corrupt site and the crash site is large, it is very likely that testers may miss the corrupt site and do not know what happened until the program crash. Current research often prevents the occurrence of the crash but does not point out where the corruption point is.

(3) There is an unusual tricky method to exploit the stack-based buffer overflow vulnerability through the saved base pointer [2, 12]. Our stack frame backtracing approach detects the anomaly of the saved base pointer with limitations.

(4) We extensively survey the interception techniques on instrument and intercept programs. Through these techniques our instrument tool monitors their running behaviors in execution when only COTS (Commercial Off-The-Shelf) executables available for analysis on the platform of Microsoft Windows.

1.5 Synopsis

In Section 2, related work will be presented. We investigate the Win32 API hooking techniques, which will be presented in Section 3. It is discussed in a dedicated chapter because our implementation on process rewriting for function call wrapping use a similar method to one of these injection and interception approaches. Moreover, one of API hooking techniques is utilized to provide tainted input analysis to a certain extent. The research method is explained in Section 4. The implementation issues are presented in Section 5. Finally, the experimental results are in Section 6 and the conclusions in Section 7.

2 Related Work

A considerable amount of work has been performed on detecting program errors and identifying their root causes either by static analysis or observing their running behavior through dynamic program instrumentation. In this section we review different works in each category and relate them to our work.

2.1 Static analysis

Static analysis is based on the information provided by the source code. It may validate the call sequence to find the program error or check if the vulnerable function call is used without actually executing the application [29]. The drawbacks of this way to find bugs are: (1) there are too many program states to verify, (2) it cannot know some dynamic information such as pointers and should use other inexact measures to analyze [8, 25, 27]. Livshits and Lam proposed a pointer alias analysis with path and context sensitivity for bug detection in C programs [15]. They made an unsound assumption to reduce spurious aliases and speedup the analysis. Our method combines the static binary disassembly analysis and dynamic instrumentation and monitoring to point out where the corruption occurs.

Chen and Wagner use a formal approach to examine whether the program violates the pre-defined security properties, which are described by Finite State Automata (FSA) [3]. The programs to be tested are modeled as pushdown automata (PDA) and MOPS uses model-checking techniques to determine the reachability of exception states in the PDA. Liblit and Aiken present an algorithm for computing a set of paths given a crash site and a global control flow graph [14]. Furthermore, it uses some post-crash artifacts such as the stack trace and the event trace to reduce the set of possible execution paths. ARCHER (ARray CHeckER) uses path-sensitive, interprocedural symbolic analysis to bound the value of both variables and memory size [31]. Accesses that violate constraints are flagged as errors.

2.2 Runtime Inspection

Some works automatically add codes in the source and observe the behavior of these codes at runtime [5, 28]. The difference from our work is that we instrument the runtime process image, not the source. Therefore even if we don't have the source, we still can detect the program errors, or even add survival patches. Hangal and Lam present DIDUCE for tracking down software bugs using automatic anomaly detection [9]. DIDUCE aids programmers in detecting complex program errors and identifying their root causes. It dynamically formulates hypotheses of invariants obeyed by the program. Our work is also based on runtime inspection, but in different method from DIDUCE. DIDUCE observes the invariants at runtime and check if the program violates them. Binary Rewriting protects the integrity of the return address on the stack by modifying the binary code [19]. The difference from our work is that its detection on stack overflow has false positives when the corruption occurs not in the current stack frame. There is another work presented by Nebenzahl and Wool similar to Binary Rewriting [16]. They developed an anti-stack-smashing defense technique for Microsoft Windows systems. The instrument process, which is called "vaccination" in their paper, is at install-time and does not need access to the source code. But they do not present any results on detecting the known or unknown vulnerabilities of real software. Avijit et al. presented a runtime approach for protection against all known forms of buffer overflow attacks [1]. Their solution consists of two tools: TIED and LibsafePuls. TIED extracts size information of all global and automatic buffers defined in the program from the debugging information produced by the compiler and inserts it back in the program binary as a data structure available at runtime. LibsafePlus performs unsafe C library function wrapping. They combine these two tools to protect any character array operation performed by the unsafe library function. The difference from our work is that they retrieve the array size information from debugging information. However, we observe the runtime behavior of the program. Ruwase and Lam developed a dynamic buffer overflow detector CRED that checks the bounds of memory accesses [22]. It is implemented as an extension of the GNU C compiler version 3.3.1. Their protecting mechanism from buffer overflow is based on the boundary checking other than stack inspection or call target observation. There is another interesting work of runtime inspection called PointGuard [4]. It is a compiler technique to defend buffer overflows by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers.

2.3 System Call Interception Techniques

System call interception is the fundamental technique in our work. We have surveyed a lot of system call interception techniques, which are listed in Table 1. Detours developed by Hunt and Brubacher is a library for instrumenting arbitrary Win32 functions on x86 machines [10]. It replaces the first few instructions of the target function with unconditional jump, which points to the user-provided detour function. Users can do the interception work in the corresponding detour function. The instructions removed from the target function are preserved in a corresponding trampoline function. When the target function is called, the control will jump to the detour function. After finishing the interception work, the detour function can call the trampoline function or return to the caller.

Pietrek develops API-SPY in his book [18]. API-SPY tools lists API's name in the order they are called, and record the parameters as well as the return value. The purpose of this work, the same as Detours, is to get control before the intended target function call is reached. However, the technique used in API-SPY is DLL redirection by modifying the Import Address Table (IAT), much different from the way Detours used, which is to modify the target function's prologue code to transfer control by inserting a JMP instruction at the start of the function.

2.4 Checkpoint Techniques

The best way to find software bugs is to reproduce the fault at any time. However, ideally checkpoint the process has some difficulties. We have surveyed some checkpoint techniques in order to achieve some kind of rollback and replay. Liang et al. present NT-SwiFT for software implemented fault tolerance on Windows

NT [13]. The contribution of this work is that they transparently checkpoint and recover applications on Windows NT for fault tolerance. During checkpointing, NT-SwiFT dumps the user memory space of the application to the physical storage. If the application fails afterwards, users can roll back the state to the last checkpoint in the same process. Srouji et al. also present a system for software fault tolerance on Windows NT platform [26]. The purpose of this work is very similar to NT-SwiFT. However, the main difference between them is that this work tries to reconstruct the system state when rolling back to the checkpoint by re-invoking certain Win32 API calls that change the system state such as CreateThread(), CreateFile(), CreateSemaphore(), etc. Besides, the technique of wrapping Win32 API calls in this work is to modify the address in the IAT, similar to API-SPY.

Zandy and Miller perform checkpoints of GUI-based applications on X-window system [33]. The system enables the GUI of any application to transparently migrate to or replicate on another display. It is based on a small X window server extension that enables an application to retrieve its window session from the window server and a library of GUI migration functionality that is injected in the application process at runtime.

2.5 Fault Triggering and Robustness Testing

We also survey some fault triggering works because we want to produce a lot of crashes and examine whether these crash are exploitable or not. Ghosh and Schmid present an approach to testing COTS software for robustness to operating system exceptions and errors [7]. That is, they bring up an idea to assess the robustness of Win32 applications. It instruments the interface between the software application and the Win32 APIs. By manipulating the APIs to throw exceptions or return error codes, it analyzes the robustness of the application under the stressful conditions. Whittaker and Jorgensen summarize the experiences of breaking software in their lab [30]. By studying how these software failed, they presents four classes of software failures: improperly constrained input, improperly constrained stored data, improperly constrained computation and improperly constrained output. Software testers can use the four classes of failures to break the software.

An empirical study of the robustness of Windows NT applications using random testing was made by Forrester and Miller [6]. This work designs a special random testing methodology to test the capability of the applications on NT to process the messages and events. It uses *SendMessage*, *PostMessage*, or *keybd_event* and *mouse_event* to deliver the random input to the running applications. In their experiment result, almost all the applications on the Windows NT cannot handle the random message/event input well. They, furthermore, analyzes two open source software, emacs and mozilla, to see why they failed. Almost all failures result from the pointer de-referencing error.

Jorgensen adopts different testing methods from other random testing_with hostile data streams [11]. It does not feed non-sense input to the application. Instead, it creates lexically, syntactically, semantically deformed files to feed Adobe Acrobat Reader (AAR) and observes if AAR will fail or not. When AAR fails, it uses debug information to inspect the relationship between the values of the registers and the string inserted in the deformed document intentionally.

2.6 Replay and Debug Parallel Programs

General debugger is not suitable for debugging parallel programs. Ronsse et al. explains the relationship between execution replay and debugging [21]. The main problem is that parallel programs are non-deterministic: each program run (even with the same input) might result in different program execution. It is so-called non-determinism property of the parallel programs. For example, programs do not determine the sequence of using semaphores for processors. It depends on the competition of processes' execution at runtime. As a result, when debugging the parallel programs, the execution details should be recorded for the debugger to reproduce the former execution. The purpose is to remove the non-determinism produced in execution time.

3 Win32 API Hooking Techniques

API call interception technique is the groundwork of our system. The ability to control API function calls is extremely helpful and enables developers to track down the internal actions happening during the API call. Actually, this is the reason why the title of this work adopts the word "instrument". The purpose of API call interception is to take control of some execution code. That is the so-called "stub" to force the target application to execute the injected code. Therefore, the injected code can easily monitor the program by parameter logging, return value checking, stack dump, frame pointer tracing, etc. We have investigated some API hooking techniques and the details of these methods will be discussed in the following.

There are two roles in the Win32 API hooking system. One is the Hook Server, which injects the Driver, i.e. spying DLL code, into the address space of the target application at some proper time. The other is the Hook Driver, which is injected in the target process' space to execute the interception work. Usually, the Hook Server should communicate with the Hook Driver. It retrieves the information from the Driver when the Driver performing the interception. In the following, the injection and the interception work we have surveyed will be introduced separately, corresponding to the Server's and the Driver's work.

3.1 Injection

Dynamic-link libraries (DLLs) are the structural elements of Microsoft Windows. They are separate files containing functions that could be called by programs to perform certain jobs. It is the DLLs that we want to write our spying code in. We can consider them as an extension to the application programs. In Win32, each process has its own address space and its own set of loaded DLLs. The DLL's file image must be mapped into the address space of the calling thread's process so that the program could call a function in a DLL. Here comes the problem. How does the application program use our function in the spying DLLs?

Since the target application does not have any information about our spying DLL, we need to use some tricks to force the DLL into the target process to perform the interception. There are three injection techniques we have studied.

3.1.1 Registry

There is a registry key that records the DLL names loaded by the operating system into the address space of each process at the process startup time. We can simply add the DLL name to the value of the following registry hierarchy:

`HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLLs`

The loading of these DLLs in the above registry key is performed when the USER32.DLL initializes. In its `DllMain`, USER32.DLL will use the explicit linking, *`LoadLibrary()`* call, to map these files into the address space. This is a tedious and manual way to inject the spy DLL into our target application process and has some disadvantages:

1. Windows has to be rebooted for the activation of the injection. This will add a lot of overhead for our experiments.
2. All the processes that use the USER32.DLL will be injected the spy DLL. We have to add some check code in the spy DLL to avoid injecting to the processes we are not interested in. Furthermore, if the application we are interested in does not use the USER32.DLL such as most console-based applications, this technique fails to inject the spy DLL.

3.1.2 Windows Hooks

Installing a windows hook by *`SetWindowsHookEx()`* can also force the certain DLLs into the address space of the target processes. The hook is installed as follows:

```
HHOOK SetWindowsHookEx (
    int idHook,
    HOOKPROC lpfn,
```

```
        HINSTANCE hMod,
        DWORD dwThreadId
    );
```

The first parameter indicates the type of hook procedure to be installed. The second parameter identifies the pointer to the hook procedure. The third parameter specifies the handle to the DLL containing the hook procedure. Finally, the last parameter specifies the thread to hook. The operating system will automatically inject the DLL containing the hook procedure into the address spaces of all processes influenced by the hook.

The advantage of this method is that it can use the `UnhookWindowsHookEx()` to unload the DLL when the hook is not needed. However, there are still some shortages:

1. The API call made by the target process before the hook is installed will be missed.
2. The spy DLL will not actually be loaded until the some actions performed by the target process trigger the hook procedure.
3. Windows hooks increase much overhead of extra message processing so as to decrease the performance of the whole system.

3.1.3 Remote Threads

We adopt this technique in this work. This method is more flexible and trying to force the target application process to call the `LoadLibrary()` and load the spy DLL. However, the problem is that we don't have any access to the target process's thread and trick it to load the DLL for us. In order to overcome this difficulty, we need some Win32 functions that could affect other processes. `CreateRemoteThread()` is the one. Its prototype is as follows:

```
    HANDLE CreateRemoteThread (
        HANDLE hProcess,
        LPSECURITY_ATTRIBUTE lpThreadAttributes,
        SIZE_T dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddress,
        LPVOID lpParameter,
```

```
        DWORD dwCreationFlags,
        LPDOWRD lpThreadId
    );
```

CreateRemoteThread() allows one process to create a thread that runs in the virtual space of another process. Compared with the *CreateThread()*, this API just have one more parameter to specify the process that will contain the newly-created thread. The *lpStartAddress* parameter identifies the memory address of the thread function, which will be executed after the remote thread has been created. We can use *GetProcAddress()* to retrieve the address of the *LoadLibrary()* API and consider it as thread function to load our spy DLL. Because KERNEL32.DLL is always mapped to the same address of every process, the address of the *LoadLibrary()* API will be correct for sure. Therefore, we can succeed to ask the target process to execute *LoadLibrary()* on our behalf.

In Matt Pietrek's work, he does not use the *CreateRemoteThread()* API.. Instead, he modifies the target process's memory and registers so that it look like the process is calling *LoadLibrary()* on its own. The advantage is that his method is portable to the platforms that do not support *CreateRemoteThread()*.

3.2 Interception

After injecting the Hook Driver (spy DLL code) into the target process's address space, what we have to do next is to intercept the API call. That is, the injected DLL should be responsible for accomplishing all the preparation for interception. In the following, three interception techniques we surveyed will be introduced.

3.2.1 Modification of the Import Address Table

This technique is based on the fact that Win32 executables files and DLLs are built on the neat structure of Portable Executable (PE) file format, which is an extension of Common Object File Format (COFF). PE file format consists of several logical chunks called sections. Each section stores a specific type of data. For example, the *.text* section contains all general-purpose code produced b the compiler or assembler; the *.edata* section is a list of the functions and data that the PE file exports for other modules.

In order to implement the API interception, we should pay more attention on the *.idata* section, which contains information about functions that the module imports from other DLLs. An important table resided in this section (so-called Import Address Table) contains file-relative offsets to the names of imported functions referenced by the executable's code. When the program is loaded to the memory, the addresses in the IAT will be patched to the real addresses of the imported functions.

Figure 3 shows the process of calling a function in another module. When you call a function in another module (for example, GetMessage in USER32.DLL), the CALL instruction produced by the compiler does not transfer control directly to the function in DLL. Instead, the call instruction transfer control to a *JMP DWROD PTR[00040042]* instruction in the *.text* section. The JMP instruction indirects through a DWORD variable in the *.idata* section. This *.idata* section DWORD contains the real address of API function entry point.

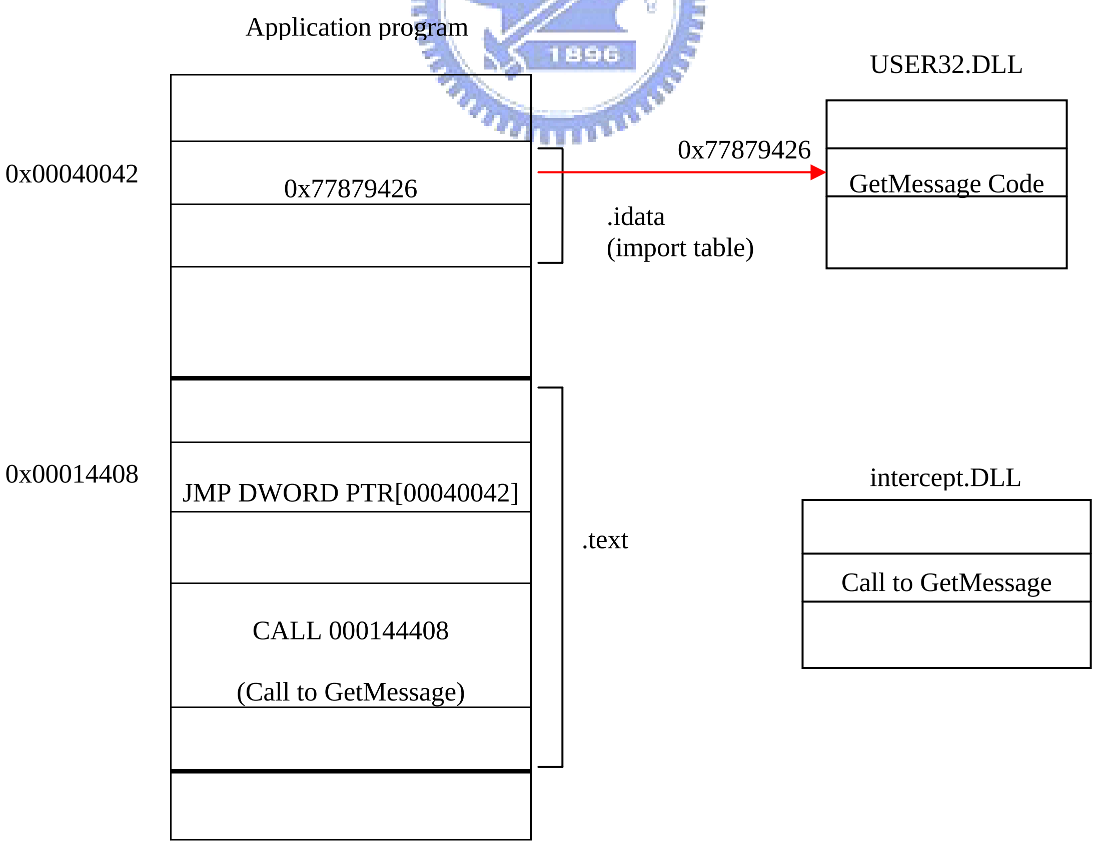


Figure 3 The process of calling a function in another module

3.2.2 API patch

This method directly modifies the API function itself. One approach is to replace the first byte of the target API with a breakpoint interrupt instruction (INT 3). Any call to the target API will generate an breakpoint exception, and the operating system will inform your API interceptor, which serves as a debugger of the target process, to handle it. The shortcoming of this approach is the overhead caused by Windows exception handling mechanism.

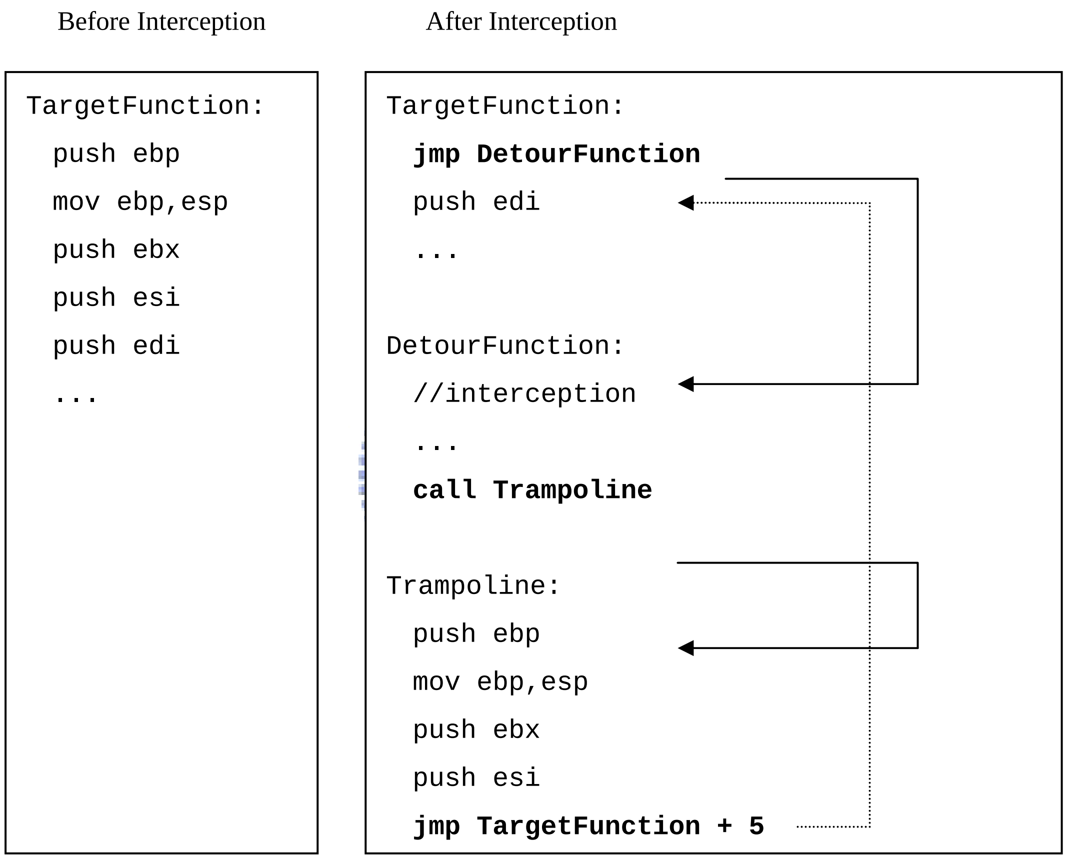


Figure 4 The interception process

Another approach to perform API patch is to modify the first few bytes of the target API with the control-transfer instruction JMP. Actually, our process rewriting mechanism to hook the user function, which will be described in Section 5, adopts this approach. Detours is an implementation of the API patch mechanism, and the following Figure 4 explains the interception process through the terms used in Detours. When execution reaches the target function, control jumps directly to the user-supplied detour function. After the detour function performs the interception work, it calls the trampoline function, which consists of the initial instructions from the target function and a jump to the remainder of the target function.

3.3 The Comparison of API Interception Works

According to the ways of injection of users' DLL into the target process and interception mechanisms, there exists some different kind of works for different purposes. Table 1 compares these surveyed interception work. After the consideration about stack frame evolving due to added monitor function and the completeness of the API interception mechanism, Detours is chosen to be the framework of this work.

Table 1 The comparison of API interception techniques

	Watchd [13]	Detours [10]	API-SPY [18]	Intel [26]
Ways to intercept API	Modify IAT to the wrapper function and when finishing logging then **calls** the real target function.	Modify target function (Replaces the first few instructions of the target function with an unconditional jump to the user-provided detour function)	Modify IAT to the logging routine and when finishing logging then **jumps** back to the real target function.	Modify IAT to the checkpoint wrapper and when finishing logging then **jumps** back to the real target function.
Does other process be influenced?	No (Copy-on-write)	No (Copy-on-write)	No	No
Ways to inject DLL into the process	Use **CreateRemote-Thread** to call LoadLibrary	1.Modify the process' memory and registers such that the primary thread will execute **LoadLibrary** 2.Rewrite the import table of the binary	Modify the process' memory and registers such that primary thread will execute **LoadLibrary**	N/A

Ways to Launch app.exe	**CreateProcess**	1.**CreateProcess** 2.System loader loads app.exe	**CreateProcess**	Use their own loader to load app.exe (Command line loader)
Where to log parameters	In the wrapper function	In the user-provided detour function	In the logging routine of the stub	In the wrapper function
Ways to get the Return value	Get the return of the target function directly in the wrapper function.	Get the return of the target function directly in the detour function.	The target function directly returns to app.exe，so it's not easy to get the return value. (Using the "return address stack")	N/A
Can determine which API to intercept by the configuration file	No (Should write user code in the DLL to be injected)	No (Should write user code in the DLL to be injected)	Yes, after reading the configuration file, it dynamically allocates memory (so-called stub) for each function	N/A
Does save registers before interception	No	N/A (It can be done in the detour function by using "pushad" and "popad".)	Yes (Use "pushad" and "popad" in the stub)	Yes (Save/restore the registers in the stub)

4 Research Method

Our research uses the following approaches to manifest and analyze the crash process as precisely as possible.

4.1 Control Flow Anomaly Detection

If programs crash, programmers and hackers are eager to find the bugs. There are two main causes of a crash. The first is accessing data in an invalid address, for example, null pointer assignment. The second is transferring control to an invalid address, often due to buffer overflow. The latter is the more serious in the two cases. In this situation, we can transfer the control of the program by overwriting the following data:

(1) Return address: The corruption of this data belongs to stack-based control flow anomaly and will be detected by our stack corrupt site identification mechanism. When the current function returns, the program transfers the control to the code designated by the return address. By overwriting the return address, we can jump to any position in the process. After the function returns, the control flow will be intercepted. This is the popular target of buffer overflow exploit.

(2) Saved base pointer: The corruption of this data also belongs to stack-based control flow anomaly and will be detected by our stack corrupt site identification mechanism. Saved base pointer points to the previous stack frame. If the saved base pointer is overwritten, the process will have a fake frame after returning from the current function and will jump to the fake return address.

(3) Function pointer: This data may be in the stack or heap and will be detected by our call target validation mechanism. When overwriting the function pointer, the process will jump to an arbitrary position. Overwriting the virtual function pointer in the heap is also a common vulnerability in C++ program.

The entire control flow anomaly caused by overwriting these data mentioned above would be detected by the following two mechanisms. Some limitations will be described in the Section 6.

4.1.1 Stack Corrupt Site Identification

When the program crashes, by inspecting it using the debugger we know the instruction where the program stops running. The point where the program stops running abnormally is the crash site. When the stack-based overflow occurs, the stack is "corrupt" for the saved base pointer and the return address corresponding to a certain function is overwritten. This is the point where the stack becomes abnormal. At some later time, this program must either crash or be exploited. The goal of the stack corrupt site identification is that right after the control flow of the program has been changed, we identify where the corrupt site is as precisely as possible.

Figure 5 is a sample program to demonstrate the distinction between the crash site and the stack corrupt site. Function *main* passes the pointer of its local buffer *buff* to function *a*, and then function *a* passes it to function *b*. In function *b*, after *strcpy()* finishes copying the overlong string to *main*'s local buffer, the stack is corrupt. However, the program has not yet crashed until the function *main* returns. Obviously, the debugger could not specify the distance between the stack corrupt site and the crash site.

```
#include <stdio.h>


void b(char *buff){
    strcpy(buff, "AAAAAAAAAAAAAAAAAAAAAA"); /* overlong string */
    /* stack corrupt site */
    ......
}
void a(char *buff){
   b(buff);
}
void main(){
   char buff[4];
```

```
    a(buff);
}   /* crash site */
```

Figure 5 The sample program to demonstrate the crash site and the corrupt site

In the following sub-sections, the mechanism to identify the stack corrupt site is described.

4.1.1.1 Pertinent Registers to a Stack

In order to understand the operation on a stack, we should know some specific assembly language knowledge. Normally, there are three registers that are pertinent to the operation on a stack: EIP, EBP and ESP.

EIP is the extended instruction pointer. It stores the address of the current instruction we are executing. When we call a function, this address will be pushed on the stack. We call the saved EIP the return address (RET). When exiting the function, the control flow will go back to RET for later execution. ESP is the extended stack pointer. It points to the current position on the stack. When we use push or pop instruction to add or remove data on the stack, ESP will change as well. Moreover, we could change the ESP by direct stack pointer manipulation. Finally, EBP is the extended base pointer. It is used to access the stack data such as local variables and offsets in a function and should keep the same throughout the lifetime of the function.

4.1.1.2 Stack Frame Backtracing

Stack frame backtracing employs the fact that saved base pointer points to previous saved base pointer in the stack. Typically, the function prologue is used to allocate the space on the stack for local variables. The following short disassembly shows how the compiler decided to implement the allocation of stack variables.

```
// function prologue
PUSH EBP        // save old frame pointer
MOV EBP, ESP    // the current EBP points to the saved EBP
SUB ESP, X      // stack variables allocation with X bytes
```

The old EBP is pushed on the stack, and then the current EBP is overwritten by the address of stack pointer, which points the top of the stack. That is, the current EBP points to the previous saved EBP. If we continuously trace back the saved EBP, the tracing will reach the saved EBP of main function. We utilize stack frame backtracing to verify that the call stack is sound and furthermore identify the stack corrupt site when the stack-based overflow occurs.

We define our term "stacktrace". In Figure 6, function A invokes function B. Therefore, the stack frame of function A is in the higher address and the stack frame of function B is in the lower address. Now assume that the EBP register points to the saved base pointer of function B. If we perform the stack frame backtracing, we will generate a stacktrace, which comprises {$(\text{SavedEBP}, \text{RET})_B$, $(\text{SavedEBP}, \text{RET})_A$, …, $(\text{SavedEBP}, \text{RET})_{Main}$}. Actually, this sequence could be understood easily by realizing that the main function calls some other functions and then some other functions call function A, and then function A calls function B.

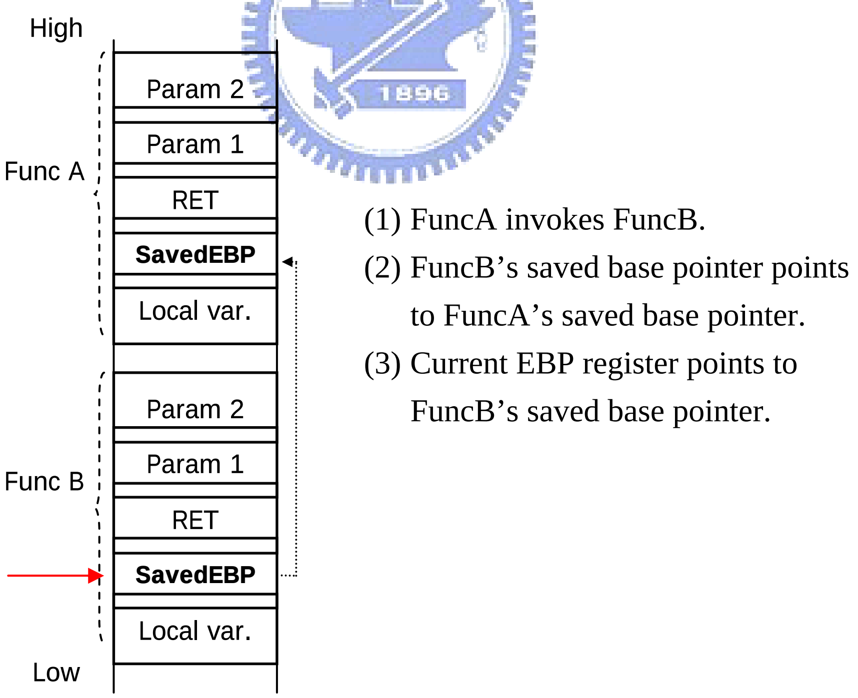


Figure 6 The operation of stack frame backtracing

We first insert a monitor function in the function's prologue and epilogue separately to perform the detection mechanism and we have to ensure that this monitor function will not disturb the original program's normal execution. What this monitor function performs in the function's prologue and epilogue is as following:

(1) In the prologue:

- Reserving all the registers
- Using the current EBP to enforce stack frame backtracing
- Restore all the registers

(2) In the epilogue:

- Reserving all the registers
- Using the current EBP to enforce stack frame backtracing
- Comparing the stacktrace with the prologue's stacktrace and point out the difference
- Restore all the registers

To detect the stack corruption, we compare the stacktraces generated in a certain function's prologue and epilogue. If the stacktraces are different, there must exist some stack buffer in a certain function growing out of bound so that the return address or the saved EBP corresponding to that function is overwritten.

4.1.2 Call Target Validation

This mechanism is designed for the control flow anomaly resulted from the function pointer overwritten. We instrument the application process at the point where each CALL instruction is. With this instruction-grained instrument, we insure that each CALL instruction is transferring control to the normal function entries.

We use the software interrupt to enforce this instrument. We overwrite the first byte of the CALL instruction with breakpoint interrupt instruction (INT 3), and install a corresponding exception handler. When an INT 3 instruction is executed, it generates a Debugger Breakpoint Exception, and the handler gains control to perform call target validation. After finishing the validation, we will restore the original EIP and CALL instruction.

According to the way of the CALL target is determined, we divide the CALL instruction into four types: API call, relative call, memory call and register call. The former two types of CALL instruction does not need to instrument the INT 3 instruction because typically these CALL targets will not be overwritten. The term "static calls" will be used to represent these two calls in the following text. For the memory call and register call, the target of them will be determined at runtime. We only instrument these "dynamic calls". The details of the implementation of this software interrupt will be described in Section 5.

The steps of the call target validation are as following.

(1) Off-line parse the disassembly of the program to get the CALL information.

- Recognize the CALL type as either static or dynamic.
- Retrieve the function entries and the callsites of valid jump instructions.

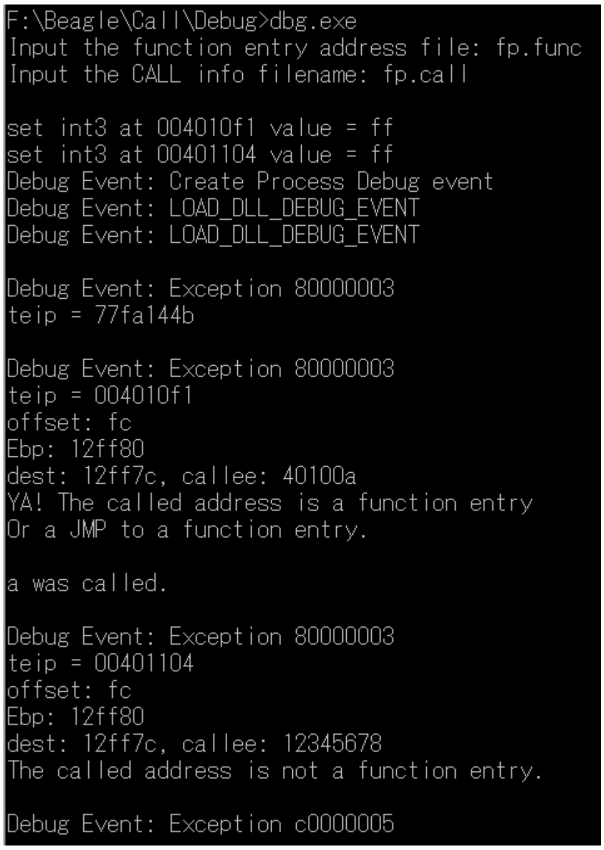
(2) There are some INT3 instructions in the original programs, and we just handle the INT 3 we have inserted.

(3) According to the information parsed at step (1), we could compute the CALL target address at runtime.

(4) When the call target computed at step (3) matches one of the function entries retrieved at step (1), this call target is valid.

```
#include <stdio.h>


void a(void) {
   printf("a was called.\n");
}
void b(void) {
   printf("b was called.\n");
}
int main(void) {
   void (*p)(void);
   p = a;
   (*p)();
   *p = (void (__cdecl *)(void))0x12345678;
   (*p)();
```



```
  return 0;
}
```

Figure 7　A sample program to detect the function pointer anomaly

We could see a sample program in Figure 7. The function pointer ***p*** declared in main is assigned in turn the address of function ***a*** and an invalid value *`0x12345678`*. When the function ***a*** is called through function pointer ***p***, the call target is valid. However, when the next *`0x12345678`* is called, we detect that this call target is invalid because it does not match any function entries in this program.

4.2 Tainted Input Tracing

Establishing the bridge connecting the software robustness and security is a brand-new and fantastic idea in the research area of software testing. Traditional testing techniques are well equipped to find the bugs that violate the specification, but lack of looking for how these bugs relate to the security issues. For example, there are plenty of application crashes during our everyday life and you may wonder whether bugs leading to these crashes are security-related.

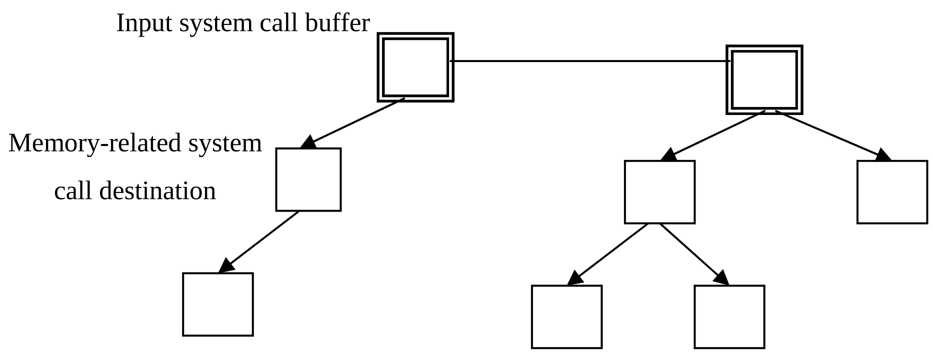


Figure 8　The buffer tree constructed during the program execution

Input tracing mechanism combines the function wrapping techniques and the stack overflow detection with the maintenance of the runtime buffer tree. First of all, I/O related API call such as *ReadFile* would be intercepted to create the root buffer for those parameters related to the input. Second, when memory related API calls, such as *lstrcpyw,* are invoked, its destination memory buffer will be added as a child node of the root buffer. The buffer tree is shown in Figure 8. Finally, when stack corruption occurs, this system will traverse the whole buffer tree and compare the buffer address to the corrupted stack address. If these two addresses matches, the path to the suspect buffer will be printed out. And this path stands for the input pollutant flow causing the stack to be corrupted. That is, malicious users may have capability of putting their payload on the stack.

5 Implementation

The instrument tool mentioned above helps testers to know why the programs crash by observing the stack and input tracing. Furthermore, to manifest the exploit process of the known vulnerable programs is another proof that this tool is useful. Using the log of runtime monitoring on the running applications, this tool can help analyze why this software is exploitable.

5.1 System Architecture

There is an instrument tool to communicate with the API/function wrapper DLL that is injected into the target process. During the execution of the application program, testers may want to modify the parameter or return values of a certain suspicious functions. Figure 9 shows the system architecture.

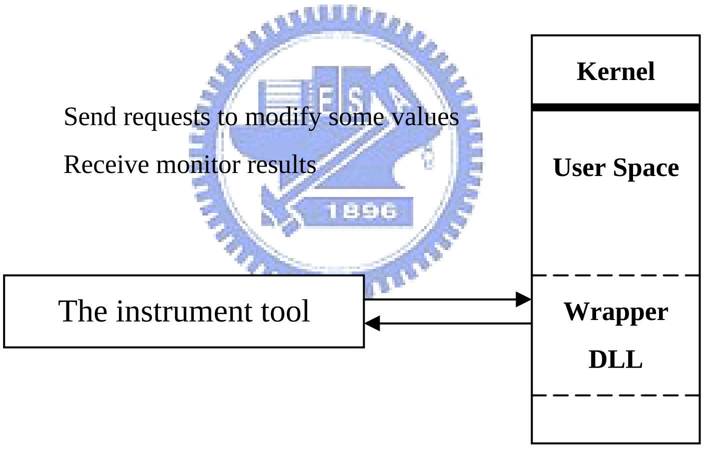


Figure 9　System Architecture

5.2 Process Rewriting for Function call wrapping

For purpose of monitoring the stack frame evolving and tracing EBP/return pair, however, API call interception still seems too coarse to pinpoint the reason why the application programs crash. Actually speaking, the most ideal scenario for crash analysis is to figure out which line of code is the onset of bugs, and it is impossible without source code. What we can do furthermore is to wrap user functions to achieve the finer-grained monitoring.

Function call wrapping is especially helpful to catch the site resulting in crashes happening on the stack. For instance, if a function in a program does some string manipulation without careful bound checking, it may crash when the string in process is out of bound. Such vulnerabilities bring about the classic and simple attack, i.e. stack overflow. By overwriting the return address through stack variables overflowing, the attacker can intercept the programs when this function returns. Therefore, the control jumps to a location where the attacker would have inserted malicious code. To deserve to be mentioned, buffer overflow attack is a kind of injection/interception mechanism. Compared with the API interception techniques mentioned above, buffer overflow cannot successfully return back to the correct site after some destructive activities since the return address and the stack is overwritten. Figure 10 shows the flow of the function call wrapper generation.

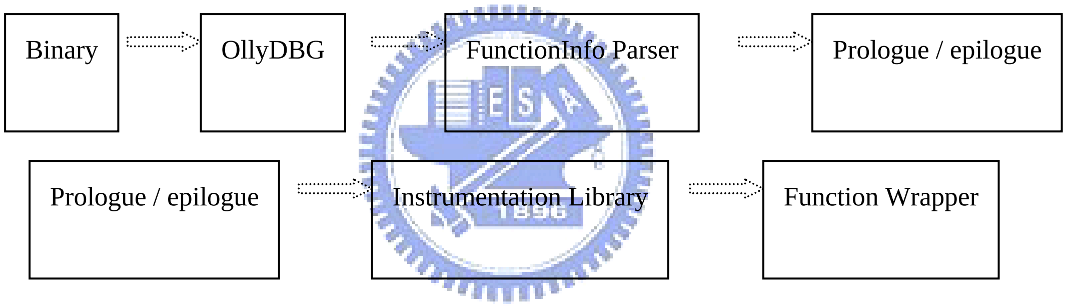


Figure 10 The flow of the function wrapping

The principle of function wrapping is similar to what Detours does in the API call interception. Detours replaces the first few instructions of the target API with unconditional jump to the user-provided monitor function. The primary difference between Detours and this function wrapper is as follows:

(1) Detours acquires the API call entry address from static linking. However, this function wrapper acquires the user function from the disassembly of the binary code of the application program through the FREE tool named “OllyDBG” [32].

(2) Detours only instruments the prologue of the API call. However, this function wrapper instruments both the prologue and epilogue of the user function. Comparing the stack tracing in a function’s entry and exit is extremely helpful to detect the anomaly of the stack.

5.2.1 Binary Disassembly

Our research method relies heavily on the disassembly ability of OllyDBG, which is a 32-bit assembler level analysing debugger for Microsoft® Windows®. It does much work on binary code analysis that we could utilize especially when the source is not available. It could recognize procedures, API calls, and complex code constructs, like call to jump to procedure. These analyses help us parse the disassembly of the application to retrieve the necessary information such as procedure call site, entry address, etc. In addition, it could disassemble all the executable modules the application loads.

5.2.2 Function Info Parser

In order to transfer control from the execution of the application process to our runtime-generated stub, we need to replace instructions at the function prologue and epilogue with a JMP to the stub. The type of the procedures we recognize is the typical function prologue and epilogue, which will do operations on the stack and frame pointer. Our function info parser retrieves prologue/epilogue information that is needed by the instrument library. In typical C/C++ programs, the compiler will generate the prologue as *" PUSH EBP" " MOV EBP, ESP"* and the epilogue as *"POP EBP" "RET (const)"*. The prologue is 3 bytes and the epilogue is at most 3 bytes. Therefore, we need to look the instructions following the prologue and the instructions above the epilogue until the space is enough to put a JMP instruction.

The following example is the result of parser:

>

60F71213 558BEC8B4508 6

60F71243 5DC3 2

60F71248 0FB60A2BC15DC3 7

60F71251 5DC3 2

<

The first line is the needed information of a prologue. The first field is the entry address of this prologue. The second field is the binary code of this prologue that will be overwritten by the instrument library. When instrumenting this prologue, the instrument library will check the third field that is the length of the binary to be replaced. If the length is less than 5 bytes, it means that there is not enough space to substitute the prologue for the JMP instruction and we leave this kind of procedure to breakpoint interruption instruction instrumentation if needed. The second lines to the last line of this function information are the epilogues. There may be multiple return site of this function, but not all of them have enough space to be instrumented.

When looking for more space for instrumenting the JMP instruction, we have to exclude the following situations that might disturb the correct execution of the target program. When the instructions following the prologue or the instructions above the epilogue should not be:

(1) JMP / CALL related instruction

(2) JMP target such as the following example

```
     push ebp
     mov ebp,esp
x:   push edi
     ....
     ....
     jmp x
```

The reason is that if we have to move these instructions to our stub, and the control flow of the original program is disturbed, which will result in the software failure even program crash.

5.2.3 Instrumentation Library

We develop an instrumentation library to replace the certain functions at runtime. According to the information provided by the function info parser, the instrumentation library will allocate the space for the stub and append the intended instructions on the stub. The most important instruction is to CALL the monitor function where we could backtrace the stack for corruption detection. Detours provides some useful library to append the certain instruction on the stub.

5.3 Breakpoint Interrupt

We use the breakpoint interrupt instruction to instrument each CALL instruction in the program we are interested in to enforce the call target validation mechanism. We overwrite the first byte of the CALL instruction with INT 3, and install a corresponding exception handler. When an INT 3 instruction is executed, it generates a Debugger Breakpoint Exception, and the handler gains control to perform call target validation. After finishing the validation, we will restore the original EIP and CALL instruction.

(1) The instrument tool creates/attaches the application process.
(2) Replace the first byte of the CALL instruction with INT 3.
(3) When trapping to OS, the instrument tool uses Win32 Debug API to handle breakpoint exception.
(4) The handler performs the call target validation.

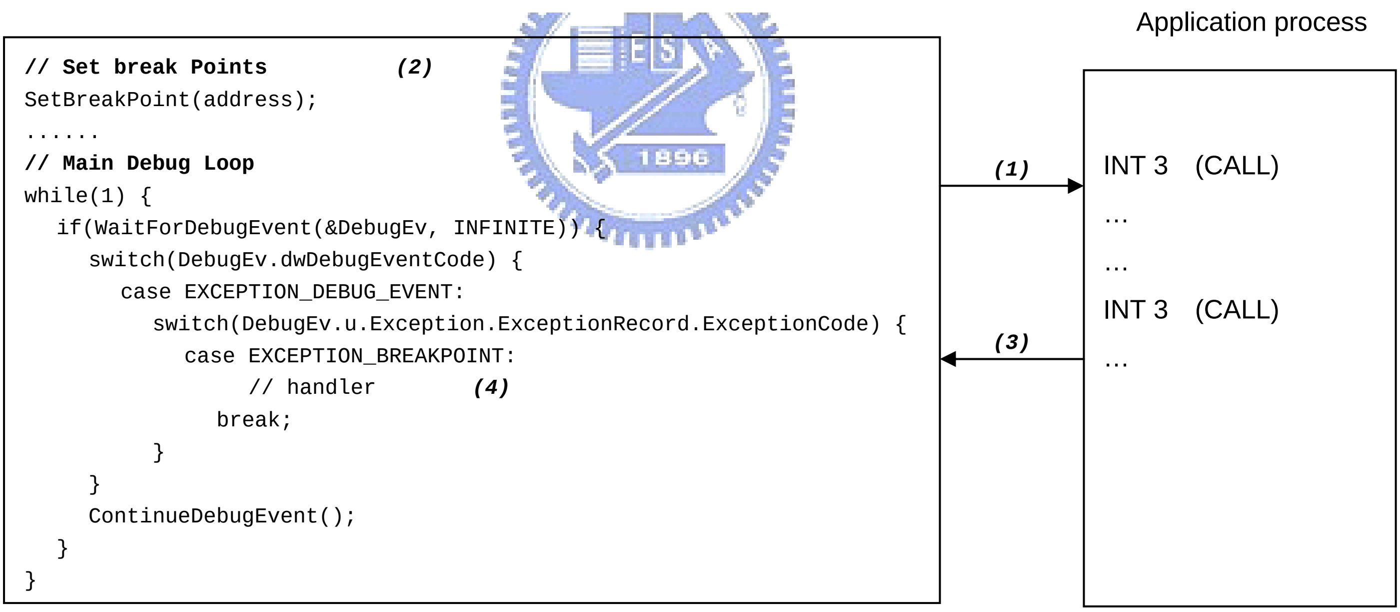


Figure 11 The instrument scenario of INT 3 instruction

The scenario of this instrument process is shown in Figure 11. The whole instrument process consists of four steps. First, the instrument tool will create the application process. Second, it replaces the first byte of the CALL instruction with INT3. Third, when the application process is executing the INT 3 we have inserted in, the control will trap to operating system. Then the instrument tool uses Win32 Debug

API to handle the breakpoint exception. Last, what we have to do is perform the call target validation in the corresponding handler.

When the application process is executing the INT 3 instruction we have inserted in, the *EXCEPTION_DEBUG_EVENT* debug event is generated. A debug event is an object used to communicate with debugger, which is the role we are playing. When a debug event is generated in the target application process, the operating system will inform us to handle this. We will use *WaitForDebugEvent()* to acquire the debug event and information about the event in a *DEBUG_EVENT* structure. This structure is defined as following:

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union {
      EXCEPTION_DEBUG_INFO Exception;
      CREATE_THREAD_DEBUG_INFO CreateThread;
      CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
      EXIT_THREAD_DEBUG_INFO ExitThread;
      EXIT_PROCESS_DEBUG_INFO ExitProcess;
      LOAD_DLL_DEBUG_INFO LoadDll;
      UNLOAD_DLL_DEBUG_INFO UnloadDll;
      OUTPUT_DEBUG_STRING_INFO DebugString;
      RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

The member *dwDebugEventCode* identifies the type of debug event. The *dwProcessId* member is the identifier of the process in which the debugging event occurs. The union *u* provides additional information relating to the debug event. The way to retrieve the additional information is determined by the *dwDebugEventCode* member.

We use *WaitForDebugEvent()* and *ContinueDebugEvent()* to handle the debug event. The *WaitForDebugEvent()* blocks the our instrument tool and waits for a debug event to occur in a process being debugged. When the debug event occurs, the system suspends all threads in the process being debugged. Its prototype is as following:

```
BOOL WaitForDebugEvent(
  LPDEBUG_EVENT lpDebugEvent,  // debug event information
  DWORD dwMilliseconds         // time-out value
);
```

The second parameter describes the number of milliseconds to wait for a debug event. If a debug event does not occur in this time, the function times out and returns FALSE. If a debug event occurs then the function returns TRUE and puts the information about event type into the *DEBUG_EVENT* structure. Then we check the event type. If it is the event corresponding to INT 3, we perform the call target validation measure as described in Section 5. After our code for validating the call target, we have to use the *ContinueDebugEvent()* to resume the thread execution and wait for next event to occur.

5.4 Experience and Further Discussion

When implementing this instrument tool, we encounter some issues that are not intuitively simple to overcome. We address these issues in this sub-section and describe our solutions and experience.

5.4.1 Stack Region

When performing stack frame backtracing, we need to figure out when to stop tracing the frame pointer. The straightforward idea is that the frame pointer should not point to the address that is out of stack region.

At first, we try to use *VirtualQueryEx()* API to retrieve the meta-data of a stack region. It provides information about a region of consecutive pages beginning at a specified address that share the same attributes. `VirtualQueryEx()` determines the attributes of the first page in the region and then scans subsequent pages until it scans the entire range of pages, or until it encounters a page with a non-matching set of attributes. Because of our wrong assumption that the whole stack region shares the same attributes, we make a serious mistake on determining the stack upper boundary. Therefore, in this wrong implementation we did not traverse the whole stack and missed many stack frames to check.

Our solution to overcome this problem is to use Thread Information Block (TIB) to identify when to stop backtracing the frame pointer. TIB is a key system data structure in Microsoft Windows and there are many data related to threads inside it, including a pointer to the thread's structured exception handler list, the location of the thread's stack and the location of the thread local storage. Furthermore, each thread in the system has its corresponding TIB.

In all Intel-based Win32 implementations, the FS register points to the TIB. As a result, we have to look at what the FS register points to for getting the information hidden in the TIB. For example, FS:[0] points to the structured exception handling chain, while FS:[2C] points to the thread's local storage array. The information we needed to judge the stack region is `pvStackUserTop` and `pvStackUserBase` field in the TIB. The 04h DWORD `pvStackUserTop` filed contains the linear address of the topmost address of the thread's stack. This thread should not have a stack pointer value that is greater than or equal to the value of this field. The 08h DWORD `pvStackUserBase` field contains the linear address of the lowest committed page in the thread's user mode stack. As the thread uses successively lower addresses in the stack, those pages will be committed, and this field will be updated accordingly. The 18h DWORD `ptibSelf` field holds the linear address of the TIB. We use this data to access the `pvStackUserTop` and `pvStackUserBase` structure. The following code is to demonstrate how to access these system data structure.

```
PTIB pTIB;
__asm {
    mov EAX, FS:[18h]
```

```
        mov [pTIB], EAX
    }
```

Therefore, we could use *pTIB->pvStackUserTop* and *pTIB->pvStackUserBase* to set the boundary when performing stack frame backtracing.

5.4.2 Stack Evolvement After Instrument

We need to explain more about the stack evolvement after our code is instrumented. The instrument library replaces certain functions at runtime. It will allocate the space for the stub and append the instructions used to perform stack frame backtracing on the stub. The instruction to call the monitor function will add a stack frame on the stack, and this stack frame is not our concern.

The instrument library inserts a JMP instruction in the prologue and epilogue and the inserted JMP instruction in prologue jumps to the following stub code:

```
PUSH addr
CALL Monitor_Function
ADD ESP, 4
```

// Instructions which is recognized by parser and moved from the original prologue

```
PUSH EBP
MOV EBP
```

……

// Jump back to the next instruction after prologue recognized by the parser

```
JMP Next_inst_after_prologue
```

The *PUSH addr* instruction is intended to pass a parameter `addr`, which is the address of the prologue, to the `Monitor_Function` but it adds 4 bytes on the stack. Afterward, the *CALL Monitor_Function* instruction pushes the return address of monitor function on the stack. After calling the monitor function, its saved base pointer will also pushed on the stack. Therefore, in the monitor function we should access the return address of the wrapped function by adding 12 bytes as following:

```
unsigned long ret = *(unsigned long *)(EBP+12);
```

Similarly, the inserted JMP instruction in the epilogue jumps to the following stub code:

```
......
PUSH addr
CALL Monitor_Function
ADD ESP, 4
// Instructions which is recognized by parser and moved from the original epilogue
......
POP EBP
RETN
```

The stack evolvement in the epilogue is similar to that in prologue. Therefore, access to the return address and saved base pointer of the wrapped function is the same as that in prologue and is not trivial as well.

5.4.3 Corrupt Site Approximation

Because of insufficient space to instrument a JMP instruction to prologue and epilogue, we do not wrap all the typical functions in the target program. Therefore, some corrupt site approximation could be discussed to increase the precision of the corrupt site identification.

For a certain wrapped function, its stacktraces performed in prologue and epilogue will fall in one of situations below under an assumption: a "normal" stacktrace is defined.

(1) If the stacktrace in the prologue is normal but the stacktrace in the epilogue is abnormal, it means that the stack is corrupted in this wrapped function.

(2) If the stacktraces in the prologue and epilogue are normal, it means that the stack is not yet corrupted.

(3) If the stacktrace in the prologue is abnormal, it means that no matter the stacktrace in the epilogue is normal or not, the stack is corrupted in one of the previous functions.

Case 3 can be divided into two situations.

(i) If the stacktrace in current wrapped function's prologue and the stacktrace in the previous wrapped function's prologue differ in one saved base pointer / return address pair as following, it means that the corruption occurred in the previous wrapped function.

Stacktrace in previous wrapped function's prologue:

(EBP$_1$,RET$_1$),(EBP$_2$,RET$_2$),…,(EBP$_n$,RET$_n$)

Stacktrace in current wrapped function's prologue:

(EBP$_1$,RET$_1$),(EBP$_2$,RET$_2$), …,(EBP$_n$,RET$_n$),**(EBP$_{n+1}$,RET$_{n+1}$)**

(ii) If the stacktrace in current wrapped function's prologue and the stacktrace in the previous wrapped function's epilogue differ in one more saved base pointer / return address pairs as following, it means that the corruption occurred in one of the previous unwrapped functions.

Stacktrace in previous wrapped function's prologue:

(EBP$_1$,RET$_1$), …,(EBP$_n$,RET$_n$)

Stacktrace in current wrapped function's prologue:

(EBP$_1$,RET$_1$),…,(EBP$_n$,RET$_n$),**(EBP$_{n+1}$,RET$_{n+1}$)**,**(EBP$_{n+2}$,RET$_{n+2}$)**,**(EBP$_{n+3}$,RET$_{n+3}$)**

If we could retrieve the function entries corresponding to these different stack frames, we could use another method such as software interrupt to wrap these functions to identity the exact corrupt site. Therefore, we could increase the precision of corrupt site identification.

6 Experiments and Assessment

This instrument tool is used to detect some known buffer overflow vulnerabilities through the proof-of-concept exploit code that will lead the program to crash.

6.1 Buffer Overflow in RobotFTP Server 1.0

To validate the correctness of the BEAGLE prototype, we need to verify that our stack corrupt site detection does point out the vulnerable function where the stack is polluted. We instrument RobotFTP Server 1.0, which has a known stack overflow vulnerability, to demonstrate that BEAGLE could detect the abnormal stack at runtime when running the exploit and terminate the program. The description of the vulnerable program follows.

RobotFTP Server is an FTP server for the Microsoft Windows platform. It has a non-trivial buffer overrun bug in the function that processes the login information that an FTP client sends. An attacker can first login with a username longer than 48 characters and login again with a username 1994 character long to overflow the return address of this function. When this program is running under the BEAGLE instrumentation, this buffer overflow will be detected and terminate the program to prevent from transferring control to the attacker's payload. The result is shown in Figure 12.

We can see first frame pointer and return address pair in the third line from bottom, (41414141, 58585858), and this is the second overlong input username. Before program returns from this vulnerable function, our epilogue monitor function backtraces the stack and discovers that this stack trace is abnormal by comparing the stack trace in the prologue monitor function.

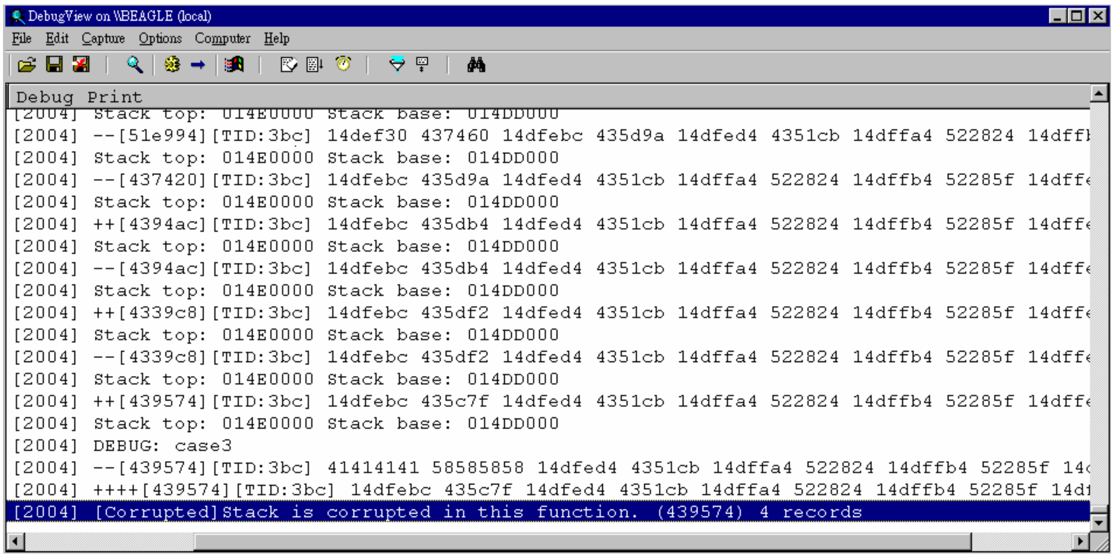


Figure 12 The stack backtrace of the RobotFTP Server 1.0 when overlong input

6.2 Buffer Overflow in Serv-U 4.1

While executing `SITE CHMOD` on a nonexistent file, Serv-U constructs the error message [24]. The code resembles the following:

```
sprintf(dst, "%s: No such file or directory.", filename);
```

The length of the `dst` buffer is limited. If a long filename was received, Serv-U will crash.

The function 00419080, which handles the `CHMOD` command, passes its local variable as an error message buffer to function 0059F9B0. The function 0059F9B0 calls function 005A01C4, which calls function 005A015C, which calls function 005A0114, which calls function 0059F988, which calls function 0059BBF8. The last function 0059BBF8 then overflow the local variable in the first function 00419080.

By our definition, the function 00419080 is the crash site of this bug; while the function 0059BBF8 is the corrupt site of this bug. The instrument tool successfully detects stack corruption in the epilogue of the function 0059BBF8 and infers the correct calling sequence. Other approaches, such as StackGuard or Binary Rewriting, would not detect the buffer overflow until the crash site.

6.3 Buffer Overflow in Palace 3.x client

The Palace is a graphical chat. Its client has a stack-based buffer overflow due to a dangerous call to `wsprintf` when a user visits an overlong link similar to the following [23]:

```
palace://('a' x 118)('BBBB')('XXXX')
```

When this situation occurs, this instrument tool detects that saved EBP / return address pair is abnormal.

6.4 Smashing C++ VPTR

We demonstrate the call target validation mechanism through a smashing C++ VPTR example. Figure 13 shows that object `f1`'s buffer is overwriting object `f2`'s virtual function pointer and cause the control of the program abnormal.

```
#include <stdio.h>
#include <string.h>
class foo
{
    public:
        char buf[20];
        virtual void bar(void){
            printf("calling bar!\n");
        }
};
int main(int argc, char* argv[])
{
    class foo *f2 = new foo();
    class foo *f1 = new foo();
    gets(f1->buf);  /* overflow point*/
    f2->bar();
    return 0;
}
```

Figure 13 The sample program of smashing C++ VPTR

In Figure 14, our instrument tool sets breakpoint interrupt instruction to perform the call target validation. Because the overlong input overwrites the virtual function pointer and causes the object `f2` could not find its correct virtual function table. We find out that address `61616161` is not a legal function entry in the program and give the alarm.

```
F:\Beagle\Call\Debug>dbg.exe
Input the function entry address file: vptr.func
Input the CALL info filename: vptr.call

set int3 at 004010f8 value = ff
Debug Event: Create Process Debug event
Debug Event: LOAD_DLL_DEBUG_EVENT
Debug Event: LOAD_DLL_DEBUG_EVENT

Debug Event: Exception 80000003
teip = 77fa144b
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Debug Event: Exception 80000003
teip = 004010f8
ReadProcessMemory failed!
Eax: 61616161
dest: 61616161, callee: 0
The called address is not a function entry.

Debug Event: Exception c0000005
```

Figure 14 The result of detecting the function pointer anomaly

6.5 Wrapping Coverage

We show our function wrapping coverage by parsing the disassembly of five common programs in Microsoft Windows. Our function info parser is responsible for distinguishing whether a function has enough space to instrument a JMP instruction in its prologue and epilogue. The result is shown in Table 2 and will be discussed following the Table.

The meaning of each column is detailed as follows:

(1) **Typical**: This is the number of functions that have typical prologue and epilogue. The ideal situation in our implementation is that we wrap the whole typical functions in the program and perform stack frame backtracing mechanism to detect the stack anomaly.

(2) **Prologue-epilogue**: This is the number of functions that have sufficient space in prologue and epilogue to insert a JMP instruction. It is these functions that we wrap, and we can compare the stacktraces produced in prologue and epilogue.

(3) **Prologue:** This is the number of functions that have sufficient space in prologue.

(4) **Epilogue:** This is the number of functions that have sufficient space in epilogue.

Table 2 Wrapping coverage

Program	Typical	Pro/epilogue	Prologue	Epilogue	Wrapping %
Word	9863	7283	9862	7284	73.84%
Excel	9046	6701	9045	6701	74.08%
Access	3034	1902	3033	1902	62.69%
PowerPoint	3149	2115	3148	2115	67.16%
Notepad	36	26	36	26	72.22%

Our wrapping coverage depends on whether there are sufficient spaces in the prologue and epilogue. Because we have to ensure that the JMP instruction we insert in function's prologue and epilogue will not disturb the program, the limitation of our wrapping technique occurs. When the instructions following the prologue or the instructions above the epilogue are JMP / CALL related instruction or JMP target, we should not wrap this function. According to our experiment, if we ignore these conditions, the program will even crash. Another reason for not enough space is that the function is less than 5 bytes.

We observe that the number of functions that have sufficient space in epilogue is always less than that in prologue. This is because that an epilogue has higher probability to become a JMP target. We found that it is also the main reason why epilogues have insufficient space to insert a JMP instruction.

6.6 Comparisons

We try to compare our work with the related work that also adopts dynamic method to provide buffer overflow protection. The comparison is shown in Table 3.

The comparison can manifest the evaluation of this work in the recent research.

Table 3　Comparison of our work with other dynamic approaches

	StackGuard [5]	RAD [19]	TIED&Libsafe [1]	CRED [22]	Our Work
Protection principle	Protect return address	Protect return address	Bound checking	Bound checking	Protect return address / saved base pointer / function pointer
Method	Insert canary	Backup return address	Wrap unsafe C function for range checking	Validate pointer access	1. Backtrace stack frame 2. Validate call target
Implementation platform	Linux	Windows	Linux	Linux	Windows
Source code needed?	Yes	No	No if compiled with **–g** option	Yes	No
When to instrument	Compilation time	Off-line binary rewriting	Runtime	Compilation time	Runtime
RET protection	Yes	Yes	Yes	Yes	Yes
Function pointer protection	No	No	Yes	Yes	Yes
BOF due to pointer arithmetic	Yes	Yes	No	Yes	Yes

Corrupt site identification	No	No	Yes	Yes	Yes
Granularity of corrupt site	N/A	N/A	Function level	Instruction level	1. Function level for stack-based corruption 2. Instruction level for heap-based corruption
Runtime overhead	Low	Low	Low	High (memory consumption)	Low

The rows in Table 3 are as following:

(1) **Protection principle**: There are two primary principles to prevent buffer overflow attack. One is to protect some important data related to control flow, and the other is to check boundary of each memory access.

(2) **Method**: This row presents the method to achieve the corresponding protection principle.

(3) **Implementation platform**: This row shows the platform on which the work implements.

(4) **Source code needed**: This row shows whether the source code of the target program is needed or not. A certain technique without the source code results in the wider-spread adoption in practice.

(5) **When to instrument**: This row indicates the timing at which the guard code is instrumented in the target program.

(6) **RET protection**: This row indicates whether the work provides return address protection mechanism.

(7) **Function pointer protection**: This row indicates whether the work provides function pointer protection mechanism.

(8) **BOF due to pointer arithmetic**: This row indicates that whether buffer overflow on some critical location due to pointer arithmetic can be detected or not. Some mechanisms can only guard against buffer overflows due to improper use of C library functions.

(9) **Corrupt site identification**: This row shows whether a certain technique can identify where the corruption occurs.

(10) **Granularity of corrupt site**: This row shows the level that a certain technique

reaches to identify the corrupt site.

(11) **Runtime overhead**: When a certain mechanism is instrumented in the target program, we evaluate the runtime overhead coarsely into two categories: low and high. The former means that the instrument will not affect the program's normal execution and the latter means that the program's performance could decrease a lot.

Although this research is intended to analyze a crash related to the security problems, this tool can also be treated as a protection tool. The protection provided by other buffer overflow protector is just to terminate the program and give the alarm when buffer overflow occurs.

7 Conclusion and Future Work

We presented the control flow anomaly detection mechanism such as stack corrupt site identification and call target validation, as a measure to automate the process of crash analysis related to the security errors. We study and employ the interception techniques to instrument and intercept programs. By these techniques we monitor their running behaviors in execution when only COTS (Commercial Off-The-Shelf) executables available for analysis on the platform of proprietary Microsoft Windows. Our contribution lies, not in inventing new approaches to detect buffer overflow attacks, but in trying to add some sort of automation in crash analysis to build up a relationship between software robustness and system security. Moreover, our stack corrupt site identification is helpful to understand why a certain stack-based crash occurs. When the program crashes, its inherent bug may have correlation to the vulnerability to be exploited. We design a tool that helps analyze the program running behavior and determine if it is an exploitable vulnerability. By process rewriting and breakpoint interruption to get control over a particular piece of code execution, we intercept the running process and checkpoint their execution status to judge if this crash is exploitable or not.

A limitation of current implementation is the lack of data flow analysis. Under the assumption of source code unavailable, it is not easy to understand how the tainted input flows to the corrupt site. If the information of data flow path is available, it will be helpful to determine the exploitability of the software. However, the primary problem is how to combine the runtime observation with the data flow analysis to deduce the exploitability.

References

[1] K. Avijit, P. Gupta and D. Gupta. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proceedings of 13th USENIX Security Symposium*, August 2004.

[2] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack Magazine*, 10(56): File 5, 2000.

[3] H. Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communication Security*, November 2002.

[4] C. Cowan, S. Beattie, J. Johansen and P.Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[5] C. Cowan, C.Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of 7th USENIX Security Conference*, pages 63-78, January 1998.

[6] Justin E. Forrester and Barton P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. *4th Usenix Windows System Symposium*, August 2000.

[7] A. K. Ghosh and M. Schmid. An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors. *10th International Symposium on Software Reliability Engineering*, November 1999.

[8] Samuel Z. Guyer and Calvin Lin. Client-Driven Pointer Analysis, *The 10th International Static Analysis Symposium*, 2003.

[9] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of International Conference on Software Engineering*, May 2002.

[10] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[11] Alan A. Jorgensen. Testing with Hostile Data Streams. *Software Engineering Notes* vol 28 no 2, March 2003.

[12] klog. The Frame Pointer Overwrite. *Phrack Magazine*, 9(55): File 8, 1999.

[13] D. Liang, P. E. Chung, Y. Huang, C. Kintala, W. J. Lee, T. K. Tsai and C. Y. Wang. NT-SwiFT: software implemented fault tolerance on Windows NT. In *Journal of Systems and Software*, November 2002

[14] B. Liblit and A. Aiken. Building a Better Backtrace: Techniques for Postmortem Program

Analysis. *UCB Technical Report*, No. UCB//CSD-02-1203, October 2002

[15] V. B. Livshits and M. S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, September 2003.

[16] D. Nebenzahl and A. Wool. Install-time Vaccination of Windows Executables to Defend Against Stack Smashing Attacks. In *Proceedings of 19th IFIP International Information Security Conference*, August 2004.

[17] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49): File 14, 1996.

[18] Matt Pietrek. Windows 95 System Programming SECRETS. *IDG Books*, 1995

[19] M. Prasad and T. Chiueh. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211-224, June 2003.

[20] rix. Smashing C++ VPTRs. *Phrack Magazine*, 10(56): File 8, 2000

[21] Michiel Ronsse, Koen De Bosschere, Jacques Chassin de Kergommeaux. Execution Replay and Debugging. In *Proceedings of the Fourth International Workshop on Automated Debugging*, August 2000.

[22] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[23] SecurityFocus. The Palace Graphical Chat Client Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/9602.

[24] SecurityFocus. Rhinosoft Serv-U FTP Server SITE CHMOD Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/9675.

[25] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis, In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 1997.

[26] J. Srouji, P. Schuster, M. Bach and Y. Kuzmin. A Transparent Checkpoint Facility on NT. In *Proceeding of the 2nd USENIX Windows NT Symposium*, August 1998.

[27] B. Steensgaard. Points-to Analysis in Almost Linear Time, In *ACM Symposium on Principles of Programming Language*, pages 32-41, January 1996.

[28] Vendicator. Stackshield: a "Stack Smashing" Technique Protection Tool for Linux. http://www.angelfire.com/sk/stackshield/, January 2000.

[29] J. Viega, J. T. Bloch, T. Kohno and G. McGraw. Token-based Scanning of Source Code for Security Problems. *ACM Transactions on Information and System Security*, 5(3): 238-261, August 2002.

[30] James A. Whittaker and Alan A. Jorgensen. Why Software Fails. *ACM SIGSOFT Software Engineering Notes*, 1999.

[31] Y. Xie, A. Chou and D. Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, September 2003.

[32] O. Yuschuk. OllyDBG. http://home.t-online.de/home/Ollydbg/

[33] V. C Zandy and B. P. Miller. Checkpoints of GUI-based Applications, In *Proceedings of USENIX 2003 Annual Technical Conference*, June 2003.