行政院國家科學委員會專題研究計畫　期中進度報告

程式動態行為安全分析(1/2)

計畫類別：個別型計畫
計畫編號：NSC93-2213-E-009-153-
執行期間：93 年 08 月 01 日至 94 年 07 月 31 日
執行單位：國立交通大學資訊工程學系(所)

計畫主持人：黃世昆

計畫參與人員：蔡昌憲 劉康民 黃有德 宋卓翰 吳孟勳

報告類型：精簡報告
報告附件：出席國際會議研究心得報告及發表論文
處理方式：本計畫可公開查詢

中　華　民　國 94 年 6 月 1 日

Software Security Assurance by Analyzing Program Run Time Behavior

Chang-Hsien Tsai[1] Shih-Hung Liu[1] Shuen-Wen Huang[2]
Shih-Kun Huang[1,2,*]
Deron Liang[2,3]
[1]Department of Computer Science and Information Engineering
National Chiao Tung University, Taiwan
[2]Institute of Information Science, Academia Sinica, Taiwan
[3]Department of Computer Science, National Taiwan Ocean University, Taiwan
[*]skhuang@iis.sinica.edu.tw

Abstract

Software vulnerabilities can be attributed to inherent bugs in the system. Several types of bugs render faults for not conforming to specifications or failures that cause crash of control flow, indefinite hang, or panic resource access. We have developed a progressive method for testing potential vulnerabilities to verify whether such crash-type failures are exploitable. When software bugs are found to be exploitable, then these bugs are very likely to be transformed into software vulnerabilities. To resolve such vulnerabilities, we have developed a tool, BEAGLE, that helps isolate bugs in presence of crash failures and reconstructs the scene of the failure point. The process of reconstruction detects the violation of control state invariants with tainted input analysis and covers security-faults related tests. We analyze bug reports from the most active projects in sorceforge.net and systematically identify exploitable bugs with the precise indication of vulnerability and prove the applicability of our method.

Keywords: COTS Vulnerability Testing, Dynamic Analysis, Software Wrapper, Control State Corruption, Exploitable Bugs

1 Introduction

Abnormal program running behavior severely affects software security. Tainted input and access race conditions can cause system failures such as buffer overflow, privilege violation, or indefinite hang that causes denial of service. We classify cases of anomaly into *crash*(due to stale control flow), *hang*(indefinite wait), and *panic*(crash due to data access violation) and deal with control-crash type failures. Programs may run out of control and crash due to the corruption of branch control state. Branch control state determines the branch flow of the next instruction for execution, corresponding to three types of branch instructions: function call, function return, and jump. If the branch targets of these instructions are dynamic addresses, they may be corrupted with an

invalid address range. For example, dynamic call target can refer to an offset of a virtual table in C++ implementation, or a function pointer in C language. Function returns are usually dynamic. Jump target can also be dynamic. If these targets are corrupted (either unintentionally, or maliciously), the program may fail to meet specifications. Such programs with corrupted control states may also be exploited and thus become vulnerable. It is difficult to reconstruct system failures after a program has crashed due to a corrupted control state and the propagated distance between crash sites and corrupt sites. To cope with this difficulty, we try to intercept and monitor running behavior during programs in execution. We aim to design a tool that analyzes the program running behavior and determine whether it is an exploitable vulnerability. Furthermore, since source codes are usually unavailable to users, in our design we assume that only COTS (Commercial Off The Shelf) executables are available for analysis.

We develop a run-time instrumentation and interception tool called BEAGLE to periodically monitor software running behavior. BEAGLE has the following capabilities. First, when the software crashes, it can approximate the latest checkpoint and determine through tainted input analysis whether the failure point is exploitable. Second, using wrapping system call API techniques BEAGLE observes the internal behavior of running programs, such as API call sequence, call parameters and return values and then determines whether they are anomalous or not. Therefore, BEAGLE is capable of verifying whether the crash site is security exploitable. In the implementation aspect, BEAGLE uses an interactive software wrapper to manipulate the interfaces between the software application and the operating system functions. This wrapper not only intercepts the functions and records the parameters and the return value, but also receives testing directives to replace calling parameters and the return value with any arbitrary value. Thus, we can use BEAGLE to conveniently maneuver the application, change the intended OS function call parameters with testing data, and observe the response of the application to determine the suspicious crash sites.

We encounter software crash occasionally and the convention solution is to restart the software. However, there is much valuable information to help developers find the reasons for a crash.

The remainder of this paper is organized as follows. In Section 2 , we will cover the buffer overflow and some popular exploits . In Section 3, we describe the difference between a crash site and a bug site. We propose a scheme to point out the bug site. In Section **??**, we introduce the background of Win32 API hooking techniques. In Section 4, we detail the implementation of the BEAGLE system in a Windows platform. In Section 5, we describe experiments with case studies. In Section 6, related work will be presented. Finally, in Section 7, we present our conclusions and future work.

2 Buffer Overflow and Control State Corruption

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 security attacks in the recent years because they are low-hanging fruits for attackers. Attackers can use the buffer to inject payload and change the control flow of the program. In this section, two common kinds of buffer overflow crashes will be elucidated, including how they are produced, how attackers can exploit them to alter the control flow and

how they can be detected.

If a program writes data in a buffer without boundary checking, other data subsequent to the buffer may be overwritten. This is often due to using unsafe C function calls, such as `strcpy()`, `sprintf()`, `strcat()` and so forth. According to where the buffer is, the overflow is known as stack overflow or heap overflow.

2.1 Stack Overflow

Nowadays, function calls are implemented in the stack because the CPU supports much about this. When a function is called, a new *stack frame* is *pushed* into the stack; when the function exits, the stack frame is *popped* from the stack. A example stack frame is shown in Figure 1. The stack frame consists of the two arguments of the function, the return address of the function, the saved frame pointer and local variables.

For example, if a `buffer` in Figure 1 overflows, local variables and saved base pointer (and even the data in the previous stack frame) will be overwritten. Once the overwritten data is referenced, the program runs into an unexpected state and often crashes. If important data is overwritten with a carefully designed malicious code, the program will execute the injected code.

2.2 Heap Overflow

Heap is the memory region in the memory space for dynamic allocated data . Heap overflow could also change the execution flow of the program and is more complicated than stack 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 provide a way to overwrite the virtual function table pointer

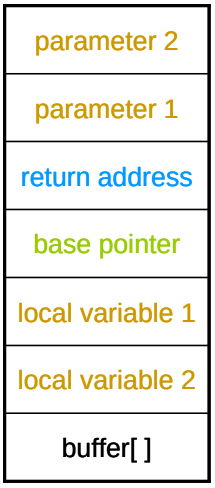

Figure 1. The layout of a stack frame

2.3 Target of Overflow

There are two main causes of a crash. The first is accessing data in an invalid address. The second is transferring control to an invalid address, often due to buffer overflow. The latter is the more serious of the two cases. In this situation, we can intercept the program by overwriting the following data:

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 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 [1].

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. This points to the previous activation record. 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 of the fake frame [17].

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. [6].

These control flow anomaly caused by overwriting these data mentioned above would be detected by the following two mechanisms. Some limitations will be discussed in the Section 4.

2.4 Correct Stack Trace

A correct call stack is very helpful for debugging. Following the stack trace, developers can examine the program source to know what happened. If the crash is caused by buffer overflow,however, the stack is usually corrupted, overwriting important clues to the crash. The main obstacle of debugging stack overflow bug is losing call stack information. Call stack is the first help for debugging crash program. For example, you load your crash program with *gdb* and issue the `bt` command. gdb would print the backtrace of all stack frames. So we get the function call sequences that introduce the crash.

This important clue may disappear when the crash is introduced by the stack buffer overflow. If the buffer overflow overwrites some important data in the stack (saved frame pointer and return address), We confirm that Visual C++ .NET 2003 and GNU gdb can not show the correct call stack after buffer overflow.

2.5 Crash Site and Bug Site

Even with the correct call stack, developers are also not easy to catch where the bug is. Because developers usually trace the bug from the crash site, but there is a situation that the bug site is far from the crash site.

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.

The distinction between the crash site and the bug site is essential. The crash site is obviously the point where software crashes, whereas the bug site is the point where buffer (heap or stack) is corrupted. For example, in Figure 2, the function *foo* passes its local buffer *buf* to the function *bar*, which overflows the buffer. After `strcpy()` returns, the stack is corrupted. However, the program does not crash until the function *foo* returns (in line 4). Obviously, the debugger could not specify the distance between the stack bug site and the crash site. Therefore, the bug site needs to be identified in order to find the root causes.

3 Research Method

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

```
void foo(void){
    char buf[8];
    bar(buf);
} /*crash site*/

void bar(char *buf){
    strcpy(buf, "this is a long string");
    /*bug site*/
      ...
}
```

Figure 2. A program with buffer overflow.

3.1 Control Flow Anomaly Detection

There are two values, the address and saved frame pointer, stayed the same during its own function's lifetime. We call these two values *stack invariant* property and check it in prolog and epilog of the function.

The return address of a function is the address where the function returns control. The return address stays the same for the function's lifetime. If the return address changed during function call, there must be something wrong. This is often caused by buffer overflow in stack.

The frame pointer is generally used to address the local variable of the function. The saved frame pointer is the frame pointer of parent function call (the function that call this function). The saved frame pointer is also unchanging during function call.

3.2 Frame Tracing

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.

```
PUSH EBP
MOV EBP, ESP
SUB ESP, X
```

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 $(SavedEBP, RET)_B$, $(SavedEBP, RET)_A$, $\ldots$, $(SavedEBP, RET)_{main}$. 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.

In general, the EBP register points the address of the saved frame pointer. Each saved frame pointer stores the address of the saved frame pointer of parent's function. And each return address is next to the saved frame pointer in high memory address. This property allows us to trace the stack invariant in run-time. We call this *frame tracing*.

4 Implementation

4.1 Interception

System call interception is the fundamental technique in our work. We survey six related

work [14, 15, 20, 26, 18, 21] in Table 1. Detours is a library for instrumentation of arbitrary Win32 functions on x86 machines. It replaces the first few instructions of the target function with an unconditional jump instruction, which points to the user-provided Detour function. Users can do the interception work in the corresponding Detour function. API-SPY tools list API's name in the order they are called, and record the parameters as well as the return value.

As with Detours, the purpose of this work is to take 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). This is very different to Detours, which modifies the target function's prolog code to transfer control by inserting a `JMP` instruction at the start of the function.

4.2 User Function Wrapping

In order to check integrity of the stack, we must wrap the user function at prolog and epilog. As shown in Figure 3, there are two steps to wrap user functions. The first step is to disassemble the program binary via *OllyDbg* [30]. The *function info parser* then parses the assembly code to generate the prolog/epilog info. The second step is to feed the prolog/epilog infomation into the *instrumentation library*. The library then generates user function wrapper on the fly.

4.2.1 Binary Disassembly

Our system relies heavily on the disassembly ability of OllyDBG, which is a assembler level analysing debugger on Microsoft Windows. It does much work on binary code analysis that we could utilize, especially when the source is not available. It can recognize procedures, API calls, and complex code constructs, like the call to jump to a procedure. These analyses help us

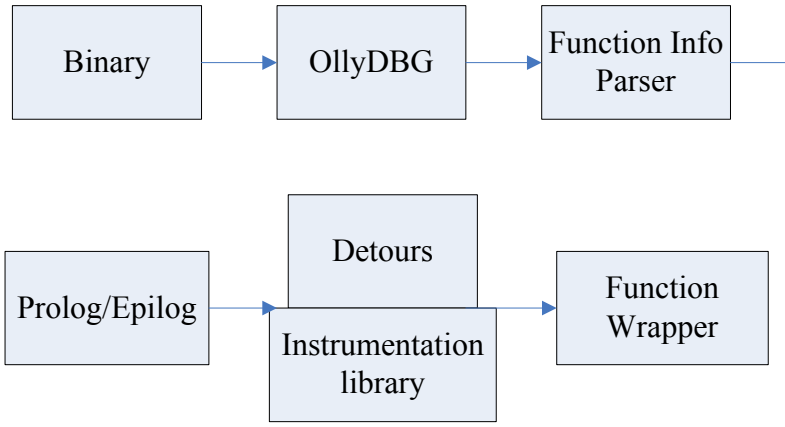

Figure 3. Process of the function wrapper generation

parse the disassembly of the application to retrieve the necessary information such as procedure call sites and entry addresses. In addition, it can disassemble all the executable modules that the application loads.

4.2.2 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 prolog and epilog, which sets up and destroys the stack frame respectively. Our function info parser retrieves prolog/epilog information that is needed by the instrumentation library.

When the prolog is instrumented, the targeted library will check 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 prolog for the `JMP` instruction and we leave this kind of procedure to the breakpoint mechanism, if needed. There may be multiple return sites in this function, but not all of them have enough space to be in-

	Dynamic hook	Dynamic unhook	Replace	Overhead	Impact
Watchd	OK.	N/A	N/A	N/A	crash together
Detours	OK.	N/A	N/A	$2\% \sim 3\%$	crash together
API-SPY	OK.	N/A	N/A	N/A	crash together
Intel	OK.	N/A	N/A	N/A	crash together
GSWTK	OK.	OK.	OK.	N/A	N/A
Binary rewriting	NO.	NO.	NO.	$1\% \sim 3\%$	crash together
Beagle	OK.	OK.	OK.	N/A	crash together

Table 1. Comparison of interception techniques

strumented.

In most C/C++ programs, the typical function prolog is

`"PUSH EBP; MOV EBP,ESP"`

and the typical function epilog is

`"RETN"` or `"RETN const"`

The two instructions in the prolog occupy only 3 bytes. We need to check the following instructions for another two bytes. Similarly, we need to find more space before the epilog. However, the instructions should not be:

1. JMP related intructions
2. Instructions which jump from others. (e.g. `PUSH EDI` in the following code.)

```
        PUSH EBP
        MOV EBP, ESP
LABEL:  PUSH EDI
        ...
        JMP LABEL
```

4.2.3 Instrumentation Library

We develop an instrumentation library to replace certain functions at runtime. According to the information provided by the function info parser, the instrument library would 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 a useful library to append instructions to the stub.

The instrumentation library takes prolog/epilog info from function info parser. It inserts a `JMP` instruction in prolog and epilog on the fly, if space is sufficient.

4.3 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.

4.3.1 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.

4.3.2 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.

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.

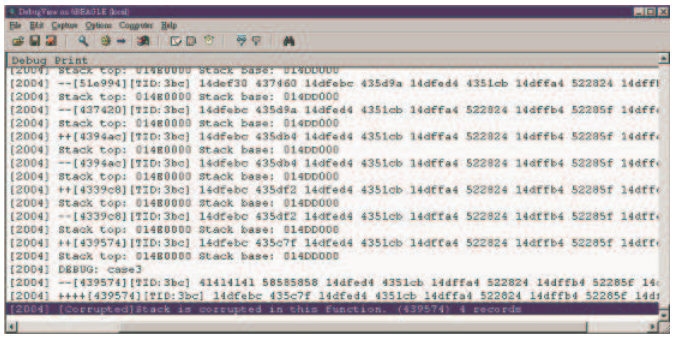

Figure 4. The stack backtrace of the RobotFTP Server 1.0 with overlong input

5 Experiments

5.1 Stack Tracing and Tainted Input Analysis

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 can detect the abnormal stack at runtime when running the exploit and terminate the program.

RobotFTP Server is an FTP server for the Microsoft Windows platform. It has a nontrivial 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 of 1,994 character 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 transferring control to the attacker's payload. The result is shown as Figure 4.

We can see the first frame pointer and return address pair in the third line from bottom, (41414141, 58585858). This is the second of the overlong input usernames mentioned above. Before the program returns from this vulnerable function, our epilog monitor function backtraces the stack. BEAGLE then determines if stack trace is abnormal by comparing it with the stack trace in the prolog monitor function.

5.2 Buffer Overflow in *Serv-U 4.1* [25]

While executing `SITE CHMOD` on a nonexistent file, Serv-U constructs the error message. 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 overflows 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. BEAGLE successfully detects stack corruption in the epilog of the function `0059BBF8` and infers the correct calling sequence. Other approaches, such as StackGuard and StackShield, do not detect the buffer overflow until the crash site.

5.3 Buffer Overflow in *Palace 3.x client* [24]

The Palace is a graphical chat program. 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:

```
palace://('a'x118)('BBBB')('XXXX')
```

When this situation occurs, BEAGLE detects that the ebp/ret pair is abnormal.

6 Related Work

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

6.1 Runtime Inspection

Using conventional debuggers, we must set breakpoints carefully, or we will miss the bug. *Bidirectional debuggers* allow developers to trace programs forward and backward. Biswas and Mall use inverse statements and execution traces to roll programs back to a previous state [4].

6.1.1 Automatic Debugger

Memory checker is another useful debugging tool. *CRED* [23] is an extension of *GNU C compiler* to track the referent object of each pointer. *Purify* [12] is a famous commercial software to detect memory errors. *Valgrind* [19] emulates the x86 CPU and runs the program binary directly. *Memcheck* in Valgrind can discover various bugs, or suspicious things, while a program is running, such as reading or writing to memory where it shouldn't read or write, using uninitialized variables. A lot of research automatically adds codes in the source and observe the behavior of these codes at runtime. 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 can still detect program errors and add survival patches.

DIDUCE [11] tracks down software bugs using automatic anomaly detection. It aids programmers in detecting complex program errors and identifying their root causes. It dynamically formulates hypotheses of invariants obeyed by the program. DIDUCE observes the invariants at runtime and check if the program violates it.

6.1.2 Abnormality Detector

As described in Section 2, buffer overflow exploit must transfer the control of program by overwriting important variables. Observing these abnormal changes, one can detect buffer overflow exploit and terminate the vulnerable program. Several works design compiler extension to add this checking. *StackGuard* [7] adds *canary* between return address and saved base pointer. *SSP* [8], originally named propolice, and Microsoft Visual Studio /GS option [5] adds canary between the saved base pointer and local variables. Before any user function returns, its canary is checked to detect buffer overflow. *StackShield* [28] uses *Ret Range Check* to protect the return address. It saves return the return address of the current function in another global variable. When the function returns, it matches the return address with the stored return address. *Binary Rewriting* [**?**] protects the integrity of the return address on the stack by modifying the binary code. It adds the same detection mechanism similar to StackGuard without source code. However, all these approaches only detect the crash site.

Feng et al. [9] extract return addresses from the call stack and use them to detect exploit. But this approach requires a training phase to learn all valid return address.

Software wrapper is another approach to monitor dangerous library call. *libsafe* [2] wraps dangerous functions (such as `strcpy()`, `strcat()` and etc.) to enforce boundary checking. Wrapping functions compute the size between the buffer starting address and saved frame pointer. If the input data is larger than the size, libsafe halts the program to avoid overwriting the saved base pointer and the re-

turn address. [22] wraps heap-related functions to detects heap overflow. By wrapping `malloc()`, it inserts canary and padding in front of each memory chunk. By wrapping `free()`, it checksums the chunk to ensure the canary unchanged. *STOBO* [13] wraps user functions to detect buffer overflow.

6.1.3 Exploit Avoidance

Most of buffer overflow exploits depend on injecting malicious code in the stack. RISE [3] `XOR` trusted binary code with random number in loading time and `XOR` back in instruction fetch time. Malicious code injected without XOR becomes garbage and soon crash.

Some researches work on *non-executable stack* to render the kind of attack useless [27]. By the way, Intel and AMD are working on their next generation CPU to include this ability in hardware [16]. Although these techniques are good for security, they do not solve all the problems. Non-executable stack just protects from some of the buffer overflow attacks and makes them be *Denial of Service*.

6.2 Fault Triggering and Robustness Testing

Fault triggering systems aide in producing system crashes. We can 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 [10]. They instrument the interface between the software application and the Win32 APIs. By manipulating the APIs to throw exceptions or return error codes, they analyzes the robustness of the application under the stressful conditions. Whittaker and Jorgensen summarize the experience of breaking software [29]. By studying how these software failed, they present 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.

7 Conclusion

We have tried to build up the relationship between system robustness and software security. Unreliable software with inherent bugs may be exploited to violate security specifications, meant to be security faults. Types of bugs are either faults not conforming to system specifications or failures such as crash, hang, and panic. We design and implement the BEAGLE system to back-track crash type failures and analyze tainted input by an input pollutant tracing algorithm to determine if such failures are security exploitable. Crash-type failures are potentially vulnerable to be exploited by tainted input due to corruption of control state. We try to approximate sites of control state corruption by stack checkpoints and monitoring run-time status. Known exploits have been tested to show the applicability of the system. We hope to discover more failures that will introduce security exploits with finer-grained stack checkpoints and further improve the precision of approximation process for corruption detection.

References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):File 14, 1996.

[2] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting critical elements of stacks. White paper, Bell Labs, Lucent Technologies, December 1999.

[3] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction

set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 281–289. ACM Press, 2003.

[4] Bitan Biswas and Rajib Mall. Reverse execution of programs. *ACM SIGPLAN Notices*, 34(4):61–69, April 1999.

[5] Brandon Bray. Compiler security checks in depth. Technical report, Microsoft Corporation, 2002.

[6] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack Magazine*, 10(56):File 5, 2000.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[8] Hiroaki Etoh. Gcc extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/.

[9] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 62–77, Los Alamitos, CA, May 11–14 2003. IEEE Computer Society.

[10] Anup K. Ghosh and Matthew Schmid. An approach to testing cots software for robustness to operating system exceptions and errors. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, November 1–4 1999.

[11] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 291–301, New York, May 19–25 2002. ACM Press.

[12] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136, 1992.

[13] Eric Haugh and Matt Bishop. Testing c programs for buffer overflow vulnerabilities. In *Proceedings of the 2003 Symposium on Networked and Distributed System Security*, February 2003.

[14] Yennun Huang, P. Emerald Chung, Chandra Kintala, Chung-Yih Wang, and DeRon Liang. NT-SwiFT: Software implemented fault tolerance on Windows NT. In USENIX, editor, *Proceedings of the 2nd USENIX Windows NT Symposium: August 3–5, 1998, Seattle, Washington*, Berkeley, CA, USA, 1998. USENIX.

[15] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium (WIN-NT-99)*, pages 135–144, Berkeley, CA, July 12–15 1999. USENIX Association.

[16] Michael Kanellos. Amd, intel put antivirus tech into chips. http://news.com.com/2100-7355-5137832.html, January 2004.

[17] klog. The frame pointer overwrite. *Phrack Magazine*, 9(55):File 8, 1999.

[18] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and

countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, August 2000. USENIX.

[19] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[20] Matt Pietrek. *Windows 95 System Programming Secrets*. IDG Books, 1995.

[21] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, June 2003.

[22] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Runtime detection of heap-based overflows. In *proceedings of 17th USENIX Large Installation Systems Administration (LISA) Conference*, October 2003.

[23] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[24] SecurityFocus. The palace graphical chat client remote buffer overflow vulnerability. http://www.securityfocus.com/bid/9602.

[25] SecurityFocus. Rhinosoft serv-u ftp server site chmod buffer overflow vulnerability. http://www.securityfocus.com/bid/9675.

[26] Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A transparent checkpoint facility on NT. In *Proceedings of the 2nd USENIX Windows NT Symposium: August 3–5, 1998, Seattle, Washington*, pages 77–86, Berkeley, CA, USA, 1998. USENIX.

[27] PaX Team. Documentation for the pax project. http://pax.grsecurity.net/docs/index.html.

[28] Vendicator. Stack shield:a "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/, January 2000.

[29] James A. Whittaker and Alan Jorgensen. Why softwarre fails. *SIGSOFT Software Engineering Notes*, 24(4):81–83, 1999.

[30] Oleh Yuschuk. Ollydbg. http://home.t-online.de/home/Ollydbg/.