國立交通大學

資訊科學與工程研究所

碩 士 論 文

自動產生攔截控制流程之攻擊程式碼

Automated Exploit Generation for Control-Flow Hijacking Attacks

研　究　生：黃博彥

指導教授：黃世昆　教授

中華民國一百年九月

自動產生攔截控制流程之攻擊程式碼

Automated Exploit Generation for Control-Flow Hijacking Attacks

研　究　生：黃博彥　　Student : Po-Yen Huang

指導教授：黃世昆　　Advisor : Shih-Kun Huang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

September 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年九月

自動產生攔截控制流程之攻擊程式碼

學生：黃博彥　　　　　　　　　　　　　　　　指導教授：黃世昆 教授

國立交通大學資訊科學與工程研究所碩士班

摘要

由於資訊領域的快速發展與應用，各類安全威脅日趨嚴重，而這些威脅都根源於軟體的缺陷，軟體安全性的探討因此成爲重要的議題。這些議題中，最大的威脅來自於軟體缺陷經常性地被揭露、使得駭客的攻擊事件層出不窮，其中零日攻擊(zero-day attacks)更造成系統及經濟上的重大危害。我們以軟體發展過程的角度分析，瞭解到安全漏洞的修補過程，是一場與零日攻擊的時間競賽，若能儘早修補漏洞，將可大幅降低其威脅性。爲了快速掌握漏洞，我們運用在軟體測試領域中，已被廣泛研究運用、自動尋找程式錯誤的方法。然而如何分析眾多的程式錯誤，優先尋找出安全性威脅的漏洞，仍是一個很困難的研究領域。在此論文中，我們將轉換角色，以攻擊者的角度來試圖產生攻擊程式碼、並將過程自動化，以此證明程式中存在安全性漏洞。我們提出基於符號執行的軟體測試方法，實作攻擊程式產生器，可任意攔截控制流程。此概念已實驗在多個眞實的程式，證明此方法之可行性。

Automated Exploit Generation for Control-Flow Hijacking Attacks

Student : Po-Yen Huang Advisor : Dr. Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Due to the rapid deployment of information technology, the threats on information assets are getting more serious. These threats are originated from software vulnerabilities. The vulnerabilities bring about attacks. If attacks launched before the public exposure of the targeted vulnerability, they are called zero-day attacks. These attacks usually damage system and economy seriously. We have analyzed the process of zero-day attacks in the perspective of software process and recognize that it is a race competition between attacks and software patch development and deployment. If developers can fix the vulnerabilities as soon as possible, the threats will be significantly reduced. In order to faster the vulnerability finding process, we use the software testing techniques, focusing on finding bugs automatically. However, it is still hard to locate security vulnerabilities from a large number of bugs. In our paper, we switch to the roles of attackers and aim at generating attacks automatically to prove that a bug is a security vulnerability. Based on symbolic execution, we are able to automatically generate exploit for control-flow hijacking attacks and perform several experiments with real-world programs to prove our method is feasible.

誌謝

考研究所是臨時的決定，重考更是一意孤行，感謝家人的諒解與支持，尤其是父母一路上的加油打氣，以及爺爺和奶奶在我每個星期往返學校前的窩心叮嚀，一直是我求學期間的最大精神支柱，讓我得以開開心心、無憂無慮的渡過研究生生涯。

再者，感謝我的指導老師黃世昆教授，帶領我一起研究與探討軟體安全這個我以前鮮少接觸的領域，並提供豐富的資源讓我得以從中發揮。除了對於生活、課業與研究上的時常關心外，並不時肯定我的表現，對於沒什麼自信的我來說更是莫大的鼓勵。

另外就是研究所這兩年中，每天一起在實驗室相處的各位同學，感謝佑鈞與世欣作為我的榜樣，帶領我了解研究生的生活。感謝孟緯從一開始的暑假至今，一起修課、當助教、作研究...等，很高興我們也一起畢業了。還有翰霖、奕任、俊維、偉明、基傑、韋翔與鈺婷，大家一起相處的每一天，是我最重要的研究所回憶。

最後，感謝田筱榮老師、孔崇旭老師與宋定懿老師在百忙之中抽空對此論文的細心指導與建議，讓其更臻完美。時光飛逝，研究所兩年的時光一下就過去了，感謝所有一路上幫助過我的人們，謹以此論文，獻給你們。

Contents

List of Listings

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Software security is an important issue in computer security field. It is due to that the software vulnerability has been the root of a variety of computer security threats. The threats are mostly induced by misuse of syntax or carelessness of logic during software development. With the inherent vulnerabilities, attackers can reason out an “exploit”, which is a crafted program input and can result in arbitrary program execution of malicious code, to intrude into computer systems and cause system damage.

On the other hand, exploits are very good test cases for software developers. In order to fix a large number of bugs, programmers have to set priority for bugs according to the degree of security threats. However, there are currently few effective ways to identify bugs as security vulnerabilities. Exploit generation is a possible way to prove a bug exploitable. Moreover, exploits can help programmers fix vulnerabilities quicker by reproducing the attack behavior.

1.1 Motivation

Manual exploit generation is a difficult and time-consuming process because it requires not only the related low-level knowledge of computer systems, such as operating system internals and assembly language, but also analyzes the control and data flow of program execution by hand. If the program under test is large or uses complex algorithms, the analysis work will be extremely difficult.

In software testing filed, many research and techniques aim to find bugs[1, 2] in a program and generate test cases[3] to trigger those bugs. However, those test cases are meaningless and only to cause the program to crash. On the other hand, the techniques of attack skills and exploit designs are multifarious, but the exploit is still very difficult to be generated manually. For this reason, this thesis tries to bridge the gap between bug finding and exploit techniques and to generate exploit automatically.

1.2 Objective

We have observed that many applications will crash by feeding a certain unexpected data. A program crashes means that control flow of the program execution is changed to invalid execution paths, and the program is very likely exploitable. In a program, many data may influence control flow of program execution, such as return address of functions, and Global Offset Table (GOT). If these sensitive data are corrupted, the intended attackers can hijack the program control. However, in most of the cases, these control sensitive data cannot be touched in normal behavior of program execution. The challenge in this thesis is to generate inputs to corrupt these control sensitive data through vulnerabilities so that the program will execute in invalid control flow.

The purpose of this work is that given a program with potential security bugs and a shell-code, we are able to find an execution path that includes a data flow to taint control sensitive data, and generate a test case that could hijack control flow and jump to the shell-code.

1.3 Overview

The structure of this thesis is shown as follows. Chapter 2 describes the backgrounds of software testing and vulnerability, along with an introduction to related work. Chapter 3 and 4 explain our method and implementation. Chapter 5 shows the experimental results. Finally, Chapter 6 concludes our thesis, with further work.

Chapter 2

Background

The techniques of software testing are very various. Static analysis can handle large-scale programs because it just scans source code without executing the programs, but false positives often happen in static analysis, i.e. less precise. Dynamic analysis executes the programs under test to find the explicit bugs, but it is often slow than static analysis. In this chapter, three popular techniques of software testing are introduced, and compared their differences.

On the other hand, the types of software vulnerabilities are also various, such as buffer overflow, command injection, and race condition. Control-flow hijacking vulnerabilities are main targets in this thesis, and five common vulnerabilities to be experimented are explained in follow related sections. In the final section, we describe the related work about symbolic execution, which is the main used technique of software testing in this thesis, and exploit generation.

2.1 Software Testing

2.1.1 Fuzz Testing

Fuzz testing is a common technique of software testing, which provides random or unexpected data as input to crash a program or to trigger assertions. Fuzz testing often treats the program under test as a black box, and cooperates with a Fuzzer[1], which is a kind of testing tools that will generate data, to repeatedly feed the program with random input. Fuzz testing is fast and precise because programs are concretely executed, but path coverage is probably low because the input are generated randomly. Considering line 12 in Listing 1, the chances of random input

[1]For example, zzuf (`http://caca.zoy.org/wiki/zzuf`) is a transparent application input fuzzer aiming to find bugs in applications.

to take *true* branch at condition statement $if(x == 2011)$ is $\frac{1}{2^{32}}$ if x is 32 bits. Fuzz testing is likely to spend much time to wildly explore the paths. Consequently, fuzz testing is inefficient for covering all paths of programs, but is good at getting some input to crash programs.

2.1.2 Symbolic Execution

Symbolic execution[4, 5, 6] is a popular technique of software testing. In contrast with concrete execution that treats the program under test as a black box and find next new path without any information, symbolic execution attempts to explore all paths in the program more systematically by transforming the path feasibility problem into boolean satisfiability problem.

The main idea of symbolic execution is to replace variables controlled by external environments with symbolic values rather than actual data. The value range of those variables represented by symbolic expressions is unlimited, i.e. any value, when the program runs initially. With program execution, those symbolic variables will taint other non-symbolic variables, and its value will be gradually restricted.

A path condition is a quantifier-free boolean formula, and its satisfiability could be validated by constraint solvers, a kind of solver for Satisfiability Modulo Theories (SMT) problem. The path condition represents the control flow of program execution and its value is *true* initially. Whenever program execution encounters branches that associate with symbolic variables, the symbolic execution forks a new execution with different path conditions, i.e. different restrictions for the symbolic value. On *true* branch, the branch condition is added to the path condition, otherwise the negation of the branch condition is used. Each of two updated path conditions will be passed to a constraint solver to determine whether the path condition is satisfiable or not. If the path condition is not satisfiable, the path will be dropped because it is infeasible. When a program execution terminates, the path condition can be solved by a constraint solver to get a test case that will traverse same path.

Consider the example code in Listing 1, variable x is replaced with a symbol X when the program starts running. At line 5, the execution is forked, one's constraint is $X \geq 0$ and another is $X < 0$. The concrete variable y is tainted by symbolic variable x at line 10, so variable y becomes symbolic and its value is $X + 100$. At line 12, the execution is forked again because variable y is symbolic. An execution is added a constraint $X = 2011$ and another is added $X \neq 2011$. The path whose path constraint is $(X < 0) \wedge (X = 2011)$ is dropped because it is infeasible. Finally, when symbolic execution has explored all three paths, each path condition is passed to a constraint solver to get a solution taking each path respectively. The process of symbolic execution is shown in Figure 1.

Listing 1: An example code for software testing

```
void test(int x)
{
  int y = 0;

  if(x >= 0)
    printf("x is greater than or equal to 0.\n");
  else
    printf("x is less than 0.\n");

  y = x + 100;

  if(y == 2011)
    printf("y is equal to 2011.\n");
  else
    printf("y is not equal to 2011.\n");
}
```

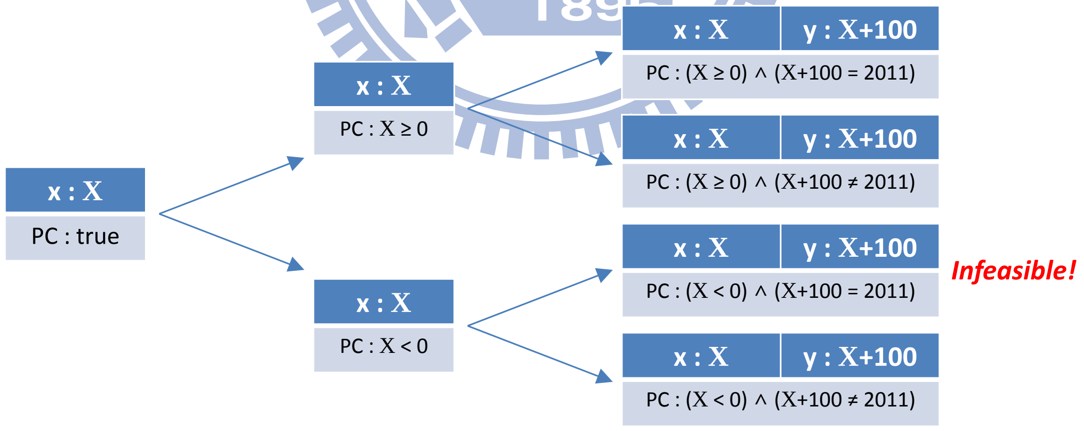

x : X
PC : true
x : X
PC : X ≥ 0
x : X
PC : X < 0
x : X
y : X+100
PC : (X ≥ 0) ∧ (X+100 = 2011)
x : X
y : X+100
PC : (X ≥ 0) ∧ (X+100 ≠ 2011)
x : X
y : X+100
PC : (X < 0) ∧ (X+100 = 2011)
Infeasible!
x : X
y : X+100
PC : (X < 0) ∧ (X+100 ≠ 2011)

Figure 1: The symbolic execution tree of Listing 1

2.1.3 Concolic Testing

In practice, symbolic execution is usually infeasible for large programs because of path explosion problem. The number of paths is growing exponentially in proportion to the number of branches, and if a program contains infinite loops, such as GUI applications, the number of paths will approach infinite. Currently, many research efforts focus on these issues, including using path selection heuristics to quicker find desired paths, using static analysis to prune off the useless parts of search space, etc.

Concolic testing is a strategy combining the accuracy of concrete execution and the systematic capacity of symbolic execution. Concolic testing executes the program under test concretely and symbolically, and explores only one path at a time. Concolic testing first executes the program under test with concrete random input, and symbolic execution is used to collect the branch conditions. Whenever a path terminates and gets a final path condition, concolic testing negates the end condition of the whole path condition to generate the next test case that will explore a new next path followed by depth-first search.

Consider the code in Listing 1, for example, the random concrete value of variable x is 0 initially, and concolic testing explores the path whose path condition is $(X \geq 0) \wedge (X \neq 2011)$. For exploring a new next path, concolic testing inverts the end constraint $X \neq 2011$ and then passes the changed path condition $(X \geq 0) \wedge \neg(X \neq 2011)$ to a constraint solver to get a new test case, e.g. $x = 2011$, and explore other new paths. The process of concolic testing is shown in Figure 2.

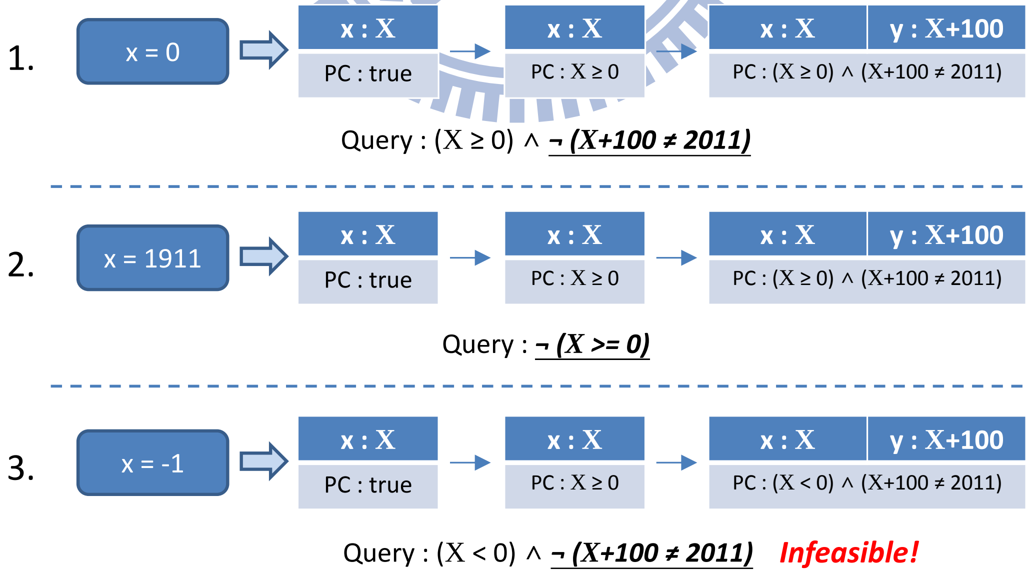


Figure 2: The process of concolic testing on Listing 1

In addition to handling path explosion problem, concolic testing addresses what symbolic execution gets stuck in some constraints, e.g. $(X = Y * Y * Y)$, because constraint solvers have trouble with non-linear constraints. Concolic testing can replace variable Y with concrete value, e.g. Y is 2, and the constraint is simplified to $X = 8$.

2.2 Software Security

2.2.1 Vulnerability and Exploit

Some bugs may not be threats for system security consideration, and those only cause the program crash with wrong output purely when unexpected input data passed to the programs. In other words, those are not exploitable. If a bug is exploitable, which is called *vulnerability*, attackers can make a program with the vulnerability perform malicious behavior and damage computer security. Unexpected test cases may trigger the vulnerability and lead a program to crash, but a crafted test case can avoid a program to crash and hijack control of programs to perform malicious tasks. The crafted test case is called *Exploit*, which is a well-designated data and reasoned out carefully by attackers. The attackers will take advantage of vulnerabilities to redirect program flow to malicious actions. The relation between exploits and input space is shown in Figure 3. Exploits often contain a piece of binary code as the payload called *Shellcode* performing malicious tasks. The behavior of shellcode is usually to open a new command shell, and it could be manually generated or by some tools, such as Metasploit[2].

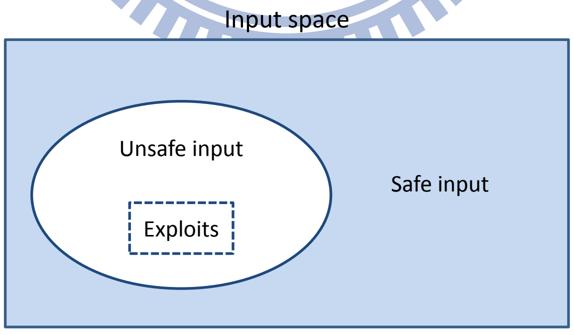


Figure 3: The relationship between exploits and input space

2.2.2 CPU Architecture and Operating System

Because exploits depend on CPU architectures and operating systems, our work is aimed at 32-bit x86 architecture and Linux system. Intel x86 architecture is the most popular CPU for

[2]Metasploit (`http://www.metasploit.com/`) is a framework for developing and executing exploit code.

Personal Computer. The design of x86 is little-endian and CISC (Complex Instruction Set Computer). The main related registers may hijack control of programs are shown as follows:

- EIP register – Points to the next instruction to be executed.
- ESP register – Points to the top of stack. Because the return address is stored in stack when calling a function, ESP may influence EIP register indirectly.
- EBP register – Points to the location of current stack frame. Because EBP is used to update ESP when functions return, EBP may influence ESP and EIP register indirectly.

In Linux system, the widely used executable format is ELF (Executable and Linking Format). The memory layout of ELF executable at rum-time is shown in Figure 4. The binary is loaded at memory address 0x0804800, and stack starts at 0xbfffffff and grow downward.

In addition, the compiler is also an important role for exploit generation. Different compilers with different versions may have different policies to generate binary code, such as variable alignment, the order of variable allocation, and the order of parameter pushing. In our work, GCC (GNU Compiler Collection) is the default compiler.

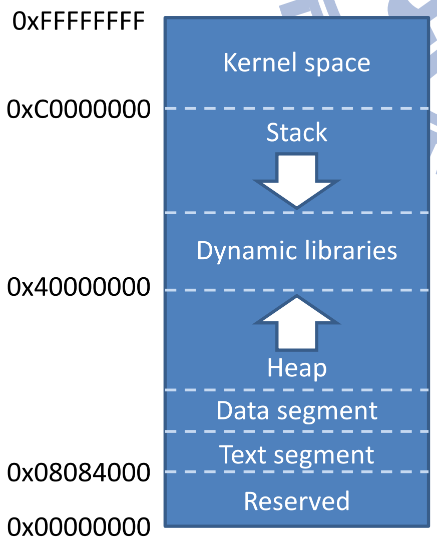


Figure 4: Memory layout in Linux

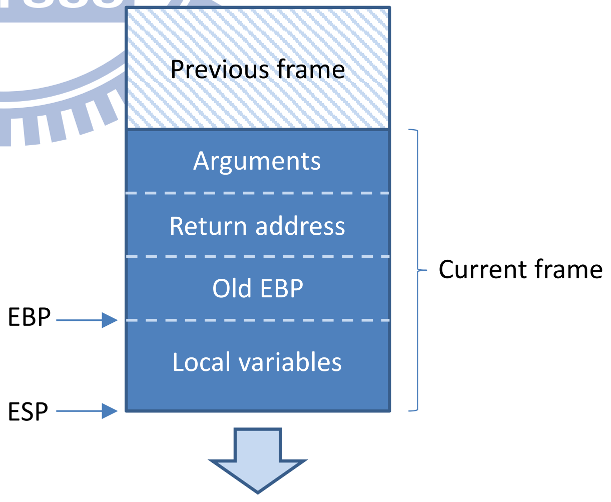


Figure 5: The layout of stack frame

2.2.2.1 Stack and Heap

Stack is an important memory region and structure for function calls. The structure of stack is LIFO (Last In, First Out) and the layout of stack frame is shown in Figure 5. On calling a function, the caller pushes its parameters and return address into stack, and then jumps to the target function. Next, the callee pushes the value of EBP register into stack and allocates the space for local variables. Because the return address is pushed into stack, control flow will be hijacked by restoring the corrupted return address to EIP register when function returns. Listing 2 and 3 show the prologue and epilogue in x86 assembly.

Listing 2: Function prologue

```
push %ebp
mov  %esp , %ebp
sub  $0x8 , %esp
```

Listing 3: Function epilogue

```
leave    ; mov  %ebp , %esp
         ; pop  %ebp
ret      ; pop  %eip
```

Heap is another memory region and allocated dynamically. For example, malloc() function in C is used to get a chunk in heap. In Linux, Glibc implements *dlmalloc* allocator to manage heap. A chunk allocated in heap is an 8-byte aligned data structure, and contains a header structure and free space. The header stores the information about the size of the previous and current chunk, and the least significant bit of size element, *PREV_INUSE* flag, specifies whether the previous chunk is in use. Figure 6 shows the two adjacent allocated chunks.

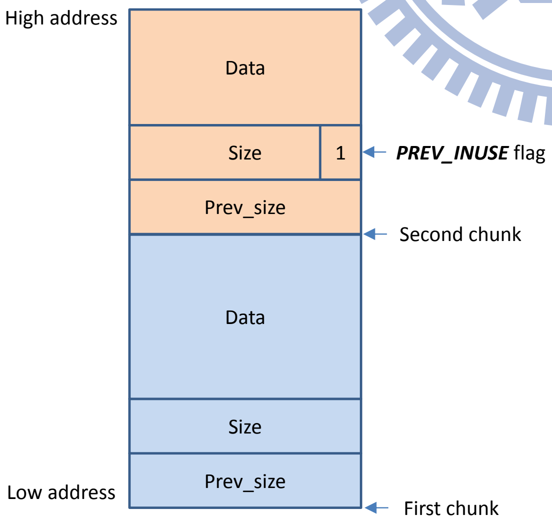


Figure 6: Heap layout in Linux

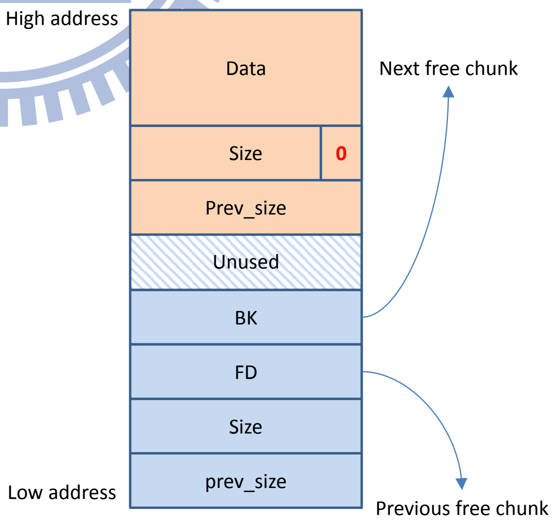


Figure 7: Heap layout after a chunk is free

When a chunk is deallocated, allocator checks whether the adjacent chunks are free. If the adjacent chunk is free, it is merged into a new big free chunk. All free chunks are stored in a doubly-linked list, each free chunk contains a forward pointer pointing to the next free chunk and a backward pointer pointing to the previous free chunk. Both pointers are stored in the unused chunk itself. Figure 7 shows the layout after freeing the first chunk.

2.2.3 Software Vulnerabilities

2.2.3.1 Stack Buffer Overflow

Stack buffer overflow is a common vulnerability and usually caused by unsafe functions, such as strcpy() and gets(). As mentioned in Section 2.2.2.1, the return address of functions is stored in the stack frame, and restored to EIP register when functions return. If the length of source input is not checked, stack smashing will happen by feeding a long input over the boundary of the destination local buffer. The corrupted return address will cause control flow hijacked when the function returns.

2.2.3.2 Off-by-one Overflow

Off-by-one error is also a common bug and arises from an error boundary condition. If EBP register or a pointer variable is overwritten, it will be exploitable. Because EBP register will update ESP register when the function returns, and the corrupted pointer may write data to arbitrary locations, both cases could influence control flow indirectly.

2.2.3.3 Heap Buffer Overflow

As noted in Section 2.2.2.1, a chunk will try to merge adjacent free chunks when it is deallocated. Listing 4 defines the behavior for updating the fd and bk pointer to merge chunks. If a heap buffer overflow vulnerability occurs, attackers can overwrite the header of next chunk to fake the size and the value of both pointers. When the current chunk is deallocated, the allocator will try to merge with next chunk and the unlink operation will cause arbitrary write of 4-byte data by attackers to arbitrary memory addresses.

2.2.3.4 Uninitialized Variable

As shown in Listing 2 and 3, the epilogue of function just moves ESP register to deallocate local variables when functions return, and the prologue increases the size of the local variables to ESP register when calling a function. If a local variable without initialization is used, its value will reuse the value left by the previous function invocation. Figure 8 shows an example, Listing 4: The macro of *unlink* operation

```
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

when invoking two function a() and b() sequentially, the stack frame of the last function will reuse the overlapping space with the previous functions. Therefore, if the previous variable can be controlled, the current variable can also be controlled via this vulnerability.

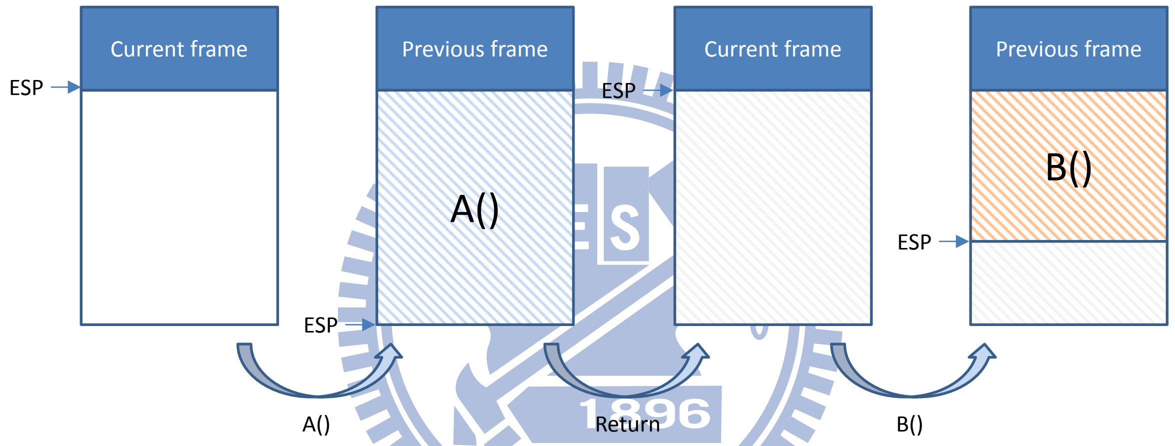


Figure 8: The example process of uninitialized variable vulnerability

2.2.3.5 Format String

If some C functions that perform formatting output, such as printf(), use unchecked input as the format string parameter, attackers can use some format tokens, such as %x, %s, and %n, to print the information in stack or write data to arbitrary memory addresses. When format tokens are encountered in a format string, the program expects that the data are retrieved from the stack. But if the input is not provided in function arguments, the program will read from or write to wrong addresses in the stack. The %n format token writes the number of characters output in front of itself to the location provided in arguments, and attackers can use %n format token to write arbitrary value to arbitrary address.

2.2.4 Protection Mechanisms

To prevent from attacks, many protection mechanisms have been proposed and applied in some compilers or operating systems. Those methods aim to increase the difficulties of successful attacks. The main mechanisms are listed below:

- ASLR (Address Space Layout Randomisation)
  - ASLR randomizes the locations of memory regions, so that attacks won't work because the address of shellcode is random and unexpected. In Linux, ASLR randomizes Stack, Heap, and share library, but not for the program image.
- W$\oplus$X (Writable $\oplus$ eXecutable)
  - W$\oplus$X sets a memory region either writable or executable. Exploits usually contain shellcode as payload, but W$\oplus$X stops the shellcode from being "executed" because the exploit must be "written" to memory.
- Stack Hardening
  - In order to protect stack against buffer overflow attacks, compilers have implemented some techniques to defend return addresses from corruption. In GCC compiler, *StackGuard* inserts a "canary" in front of return addresses and checks whether the value is changed when functions return. In addition, *Stackshield* copies away return addresses to a non-overflowable area, and restores it when functions return.
- Heap Hardening
  - To protect heap from smashing, heap consistency checking has implemented to check whether the metedata, which records the information about neighbouring chunks, is corrupted or not, whenever a heap block is freed. For example, some security checks are implemented to exclude infeasible sizes in metadata since version 2.3.4 of Glibc.

In addition, many safe functions are provided for programmers to avoid using unsafe functions. For example, strcpy() function is very dangerous because of buffer smashing, and strncpy() function is the safe version of strcpy() function that copies a string with bound checking.

2.3 Related Work

2.3.1 Symbolic Execution and Constraint Solving

Symbolic execution is a popular software testing technique, and has been applied on many dynamic symbolic execution tools. DART[7] combines symbolic execution with concrete execution, and uses the concrete value to simplify constraints whenever symbolic execution is stuck on them. CUTE[8] deals with multi-thread and pointer programs. SAGE[9] and Pex are developed by Microsoft and SAGE is the first tool that implements Whitebox fuzzing. Catchconv[10] is built on Valgrind[11], which is an instrumentation framework for building dynamic analysis tools, and also implements Whitebox fuzzing. KLEE[12] is a redesign of EXE[13], and is built on LLVM compiler infrastructure[14]. KLEE deals with the interactions with outside environments, and uses many search heuristics and constraint optimizations to get high code coverage.

In addition, many research efforts improve the limitation of symbolic execution, such as path explosion. SPD[15] uses control and data dependencies to avoid analyzing unnecessary paths, and RWset[16] analyzes tracks of all reads and writes to discard redundant paths. IPEG[17] finds the unsatisfiable core from one infeasible path to prune a family of infeasible paths. Many heuristics searches[18] are also used to find a path efficiently.

SMT solvers play an important role in symbolic execution for solving constraints. STP[19], CVC3[20], Yices[21] and Z3[22] are the most used solvers for dynamic symbolic execution tools. Complex or a large number of constraints are another bottleneck of symbolic execution. HAMPI[23] helps STP generate structured test cases. Stitched dynamic symbolic execution[24] uses decomposition and re-stitching to bypass complex functions like decryptions and checksum verifications. Cloud9[25, 26] parallelizes symbolic execution to large clusters and aims to speed up constraint solving.

2.3.2 Exploit Generation

APEG (automatic patch-based exploit generation)[27] compares the differences between a buggy version program and a patched version, and generates the exploits to fail the added check in the patched version program. This work needs a patched version program and only feasible when the patch is to fix by adding input sanitization logic. In addition, the exploits generated by

APEG almost are DoS (Denial-Of-Service) attacks, which just crash a program, without executing shellcode or malicious tasks.

AEG (automatic exploit generation)[28] generates exploits by two stages, which are finding bugs on symbolic execution and collecting run-time information on concrete execution. AEG only deals with stack buffer overflow and format string vulnerability because it has to add individual safety check constraints to detect each bug. AEG implements an end-to-end approach for exploit generation, including symbolic files, symbolic sockets, etc., and uses return oriented programming to bypass both W$\oplus$X and ASLR[29].

Heelan *et al.* [30] use binary instrumentation to perform taint propagation and collect run-time information. Their work generates exploits by checking whether EIP register is corrupted by a tainted value, and also handles pointer corruption that corrupt EIP register indirectly. But, a crashing input is essential for taint analysis in their work because symbolic execution doesn't be implemented.

In addition, some work don't generate exploits explicitly, but aim to report a bug is probably exploitable. For example, !exploitable[3] and some projects[31] of BitBlaze analyze a crash and determine whether it is exploitable or not.

[3] !exploitable crash analyzer(`http://msecdbg.codeplex.com/`) is developed by Microsoft, and provides automated crash analysis and security risk assessment.

Chapter 3

Method

AEG is the most similar work with ours for automated exploit generation at present. We explain the weakness of AEG and take an example code to show the case in this chapter. In order to overcome the weakness of AEG, we propose a new exploit generation that can handle more cases than AEG. In addition, we introduce the tool that we use to implement our method, and compare it with the tool that AEG uses.

In addition to exploit generation, we implement some path selection optimizations on S^2E to speed up the process of exploit generation. Concolic-mode simulation explores one potential vulnerable path directly, and code selection filters some complex and uninteresting library functions that are not affect exploit generation to reduce the overhead of SMT solvers.

3.1 The Weakness of AEG

Similar to our work, AEG detects vulnerabilities by symbolic execution and then collects run-time information from concrete execution with the test case generated by the previous step. AEG collects run-time information and computes exploits when vulnerability is triggered. The generated exploit may fail to work, due to the propagation distance between the vulnerable site and the triggered exploit site. AEG cannot guarantee that the program under test arrives at the triggered exploit site successfully if the exploits are not revised accordingly. Consider Listing 5, the buffer overflow vulnerability happens at line 4 where strcpy() function is located, but the exploit is triggered at line 6 where the function returns. Between line 4 and line 6, the exploiting string is reversed at line 5 and fails to work when the function returns. The process is shown in Figure 9.

Listing 5: An example code for AEG

```c
void test(char src[30])
{
    char des[10];
    strcpy(des, src); /* buffer overflow vulnerability */
    reverse(des);
}                     /* control flow is hijacked */
```

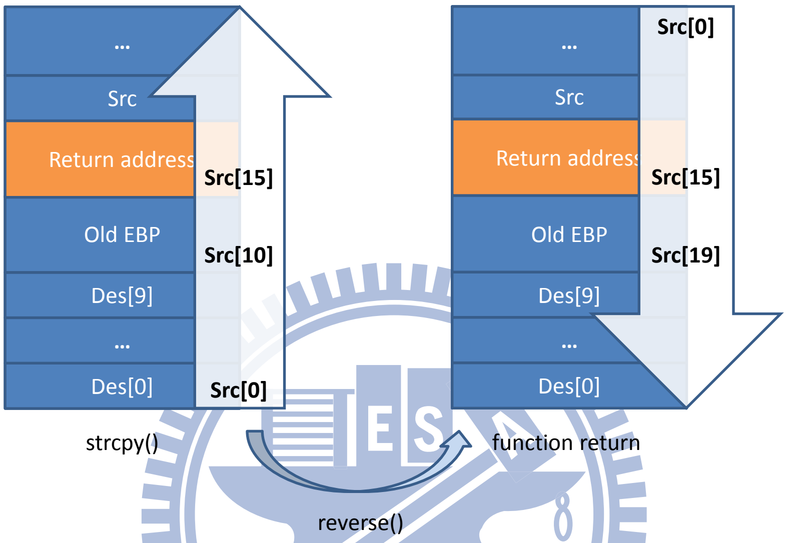


Figure 9: The memory layout before and after reverse() is executed

3.2 The Used Tool and Intuitive Idea

S^{2}E[32] is a platform that consists of QEMU[33] and KLEE for extending scale of symbolic execution ranging from applications to the whole operation system. KLEE is a symbolic execution engine built on LLVM compiler infrastructure. It interprets and performs symbolic execution on LLVM intermediate representation code, which is called *bitcode*. QEMU is a processor emulator using *Dynamic binary translation* to translate instructions between two different CPU architectures. S^{2}E adds a new LLVM back-end[34] to QEMU so that KLEE has the ability to perform symbolic execution on the whole system.

S^{2}E implements *selective symbolic execution*[35] to run as much code natively as possible. It switches from concrete to symbolic execution whenever S^{2}E accesses symbolic data. This technique enables S^{2}E to speed up symbolic execution on real systems. The architecture of S^{2}E is shown in Figure 10.

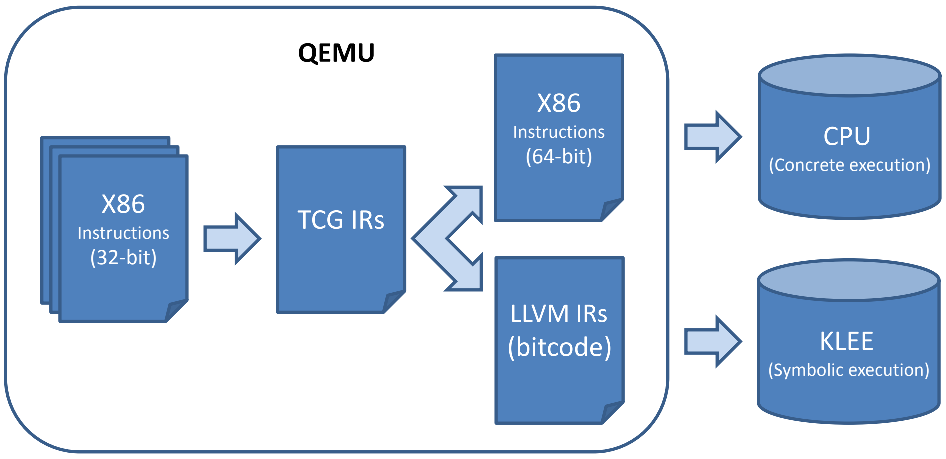

QEMU
X86 Instructions (32-bit)
TCG IRs
X86 Instructions (64-bit)
LLVM IRs (bitcode)
CPU (Concrete execution)
KLEE (Symbolic execution)

Figure 10: The architecture of S^{2}E

The differences between KLEE and S^{2}E are shown in Table 1. The most important advantage of S^{2}E for exploit generation is the low-level and real run-time information, including value of registers, contents of memory, etc. This feature is useful for reasoning out precise exploits.

Table 1: The differences between KLEE and S^{2}E

Tool	Scale	Run-time Information	Programs under Test
KLEE	application	abstract	source code
S^{2}E	operating system	real	binary(x86)

Considering Listing 6, it seems that a buffer overflow vulnerability happens at line 5 because the length of source string is longer than destination length on strcpy() function, and the *num* variable may be corrupted to cause that the *if* branch at line 7 can be controlled. In fact, when GCC compiler compiles this program with optimization, the order of variables are rearranged and the location of *num* variable is in front of *des* variable. Therefore, the *num* variable cannot be touched so that the branch condition in the *if* statement at line 7 is always *false*. For this reason, the real run-time information are critical for exploit generation.

In addition to real run-time information, other features show that S^{2}E is a good platform for exploit generation. For example, S^{2}E can perform symbolic execution on a whole operating

Listing 6: An example code for rearranging variables

```
int num = 0;
char dst[2];
char src[6];

scrcpy(dst, src);

if(num != 0)
{
  assert(num != 0); //GOAL!
}
```

system to generate exploits for vulnerabilities in library or kernel, and can operate directly on binaries to make exploit generation more useful for real-world programs. Therefore, we choose S^2E to implement automated exploit generation, and to evaluate the results.

S^2E emulates the whole operation system so that the run-time information and events can be monitored during program execution. The intuitive idea is to reason out exploits when the exploit is being triggered. Because this idea improves the shortcomings of AEG and guarantees exploits to work. But it will be treated in different ways for different vulnerabilities when the exploit is triggered. For example, stack buffer overflow triggers exploits when functions return, i.e. the *ret* instruction, but heap buffer overflow may corrupt Global Offset Table (GOT) to redirect the library function invocation and it triggers exploits when calling functions, i.e. the *call* instruction. Therefore, based on the intuitive idea, a general method is needed to handle different kinds of vulnerabilities.

3.3 Our Method

3.3.1 EIP Register Corrupted Detection

Monitoring the state of EIP register is the most comprehensive and easy method for dealing with different kinds of control-flow hijacking vulnerabilities, because the common final target of all control hijacking attacks is to control EIP register, which contains the address of next instruction to be executed. If EIP register is corrupted by symbolic data, it means that control flow can be hijacked by program input. Therefore, during symbolic execution explores paths and taints memory, the exploit generation will be started when EIP register is updated with a symbolic data. The exploit generation will search memory to find usable memory regions to inject shellcode and NOP sled, and redirect EIP register to shellcode. The process of EIP register corrupted detection and exploit generation is shown in Figure 11.

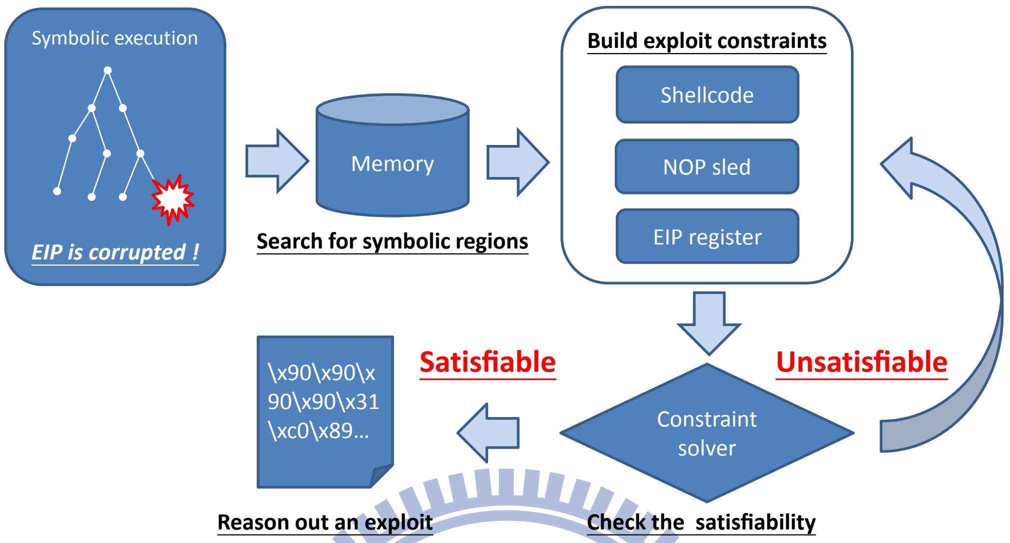


Figure 11: The process of our exploit generation

3.3.2 Exploit Generation

3.3.2.1 Shellcode Injection

For shellcode injection, the first thing is to find all memory blocks tainted by user input and large enough to hold payload. Even if a tainted block consists of many different variables, it still could be used to inject shellcode as long as the block is contiguous. It is very difficult to analyze source code manually to find a contiguous memory region tainted by user input and combined with variables. In addition, compiler often changes the order or allocation size of variables for optimization, and makes it more difficult to find a shellcode buffer manually.

The main regions data stored in memory are stack, heap and data segment. The distinctions among them are shown in Table 2. The data segment is the best region for shellcode injection because its address is decided at compile time, i.e. it is unaffected by ASLR. Stack and heap both are affected by ASLR, but the heap is more suitable than stack for shellcode injection. The primary reason is that there are more protections for stack than for heap, and the second is that the stack stores not only local variables but also other sensitive data for exploitation, such as environment variables. The locations of some variables in stack may differ on different systems. Therefore, the search order for shellcode injection is followed by data segment, heap, and finally stack.

Table 2: The differences among Stack, Heap, and Data segment

Region	Data	ASLR
Stack	local variables	yes
Heap	dynamically allocated variables	yes
Date segment	static or global variables	no

3.3.2.2 Nop Sled and Exploit Generation

When the location of shellcode has determined, *NOP sled* will try to insert a sequence of *NOP* instructions, which do nothings, in front of shellcod closely as many as possible. This padding helps exploits against the inaccurate position of shellcode among different systems, or to extend the entry point of shellocde. Finally, the symbolic data corrupted EIP register will point to the middle of NOP padding. All exploit constraints, including shellcode, NOP sled, and EIP register constraints, are passed to an SMT solver with path conditions to determine whether the exploit is feasible or not. If it is not feasible, the exploit generation goes back to the step of shellocde injection to change the location of shellcode until success or no more usable buffers in memory.

3.3.3 Pointer Corrupted Detection

In addition to EIP register, corrupted pointers may change the control flow indirectly. Particularly, a corrupted data is assigned to a corrupted pointer means that arbitrary data can be wrote to arbitrary addresses. When a corrupted pointer dereference is detected, the target of writing operation will be redirected to sensitive data, such as return addresses, .dtors section, and GOT, to taint EIP register indirectly. Otherwise, if the pointer operation is a reading operation or a writing operation but cannot point to sensitive data, the target is redirected to read from a symbolic data or write to a concrete data to perform tainted data propagation. Consider Listing 7, off-by-one overflow vulnerability will corrupt *ptr* pointer and cause the value of *buf[0]* may write to arbitrary addresses. Even if this vulnerability cannot corrupt return addresses directly, the symbolic pointer can taint EIP register indirectly and hijack control of the program.

Listing 7: An example code for pointer corruption

```
void test(int *input)
{
  int *ptr = array;
  int array[10];
  int i;

  for(i=0 ; i<=10 ; i++)
    array[i] = *(input + i);

  *ptr = array[0];
}
```

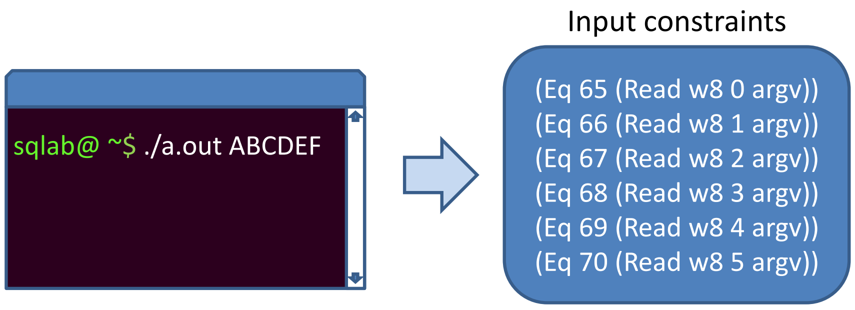
3.4 Path Selection

3.4.1 Concolic-mode Simulation

If an input data crashes a program, the execution path the crash input exploring is very likely exploitable. Exploring the suspicious path directly is more effective than searching all paths. Concolic testing is a kind of symbolic execution, and it explores one path at a time. Concolic testing stores and updates concrete values and symbolic expressions simultaneously. It uses the concrete values to help symbolic execution to determine which branch path will be explored, and uses the symbolic expressions to collect the branch conditions whenever a path is determined to travel at branches.

In contrast with implementing concolic testing on S^{2}E, simulating the behavior of consolic testing on S^{2}E is an easier and flexible method. Whenever a branch is encountered, S^{2}E does not access the concrete value of variables but add *input constraints* to limit the values of all symbolic variables to the values of original input, which are constants. Figure 12 shows an example that executes the program under test with an argument string "ABCDEF" and the input constraints are built to restrict those values of the argument. Because each symbolic variable could be only one possible value after adding the input constraints, it simulates the concrete values to choose one path to explore. This method does not modify the memory model of S^{2}E and based on symbolic execution to provide the ability of concolic testing.

Concolic-mode simulation determines whether a path is exploitable rapidly because it focuses on only one path. To cooperate with Fuzzer tools on getting an input crashing the program



```
sqlab@ ~$ ./a.out ABCDEF
```

```
(Eq 65 (Read w8 0 argv))
(Eq 66 (Read w8 1 argv))
(Eq 67 (Read w8 2 argv))
(Eq 68 (Read w8 3 argv))
(Eq 69 (Read w8 4 argv))
(Eq 70 (Read w8 5 argv))
```

Figure 12: An example for input constraints

under test automatically, it is a very powerful technique to generate exploits. The process is shown in Figure 13.

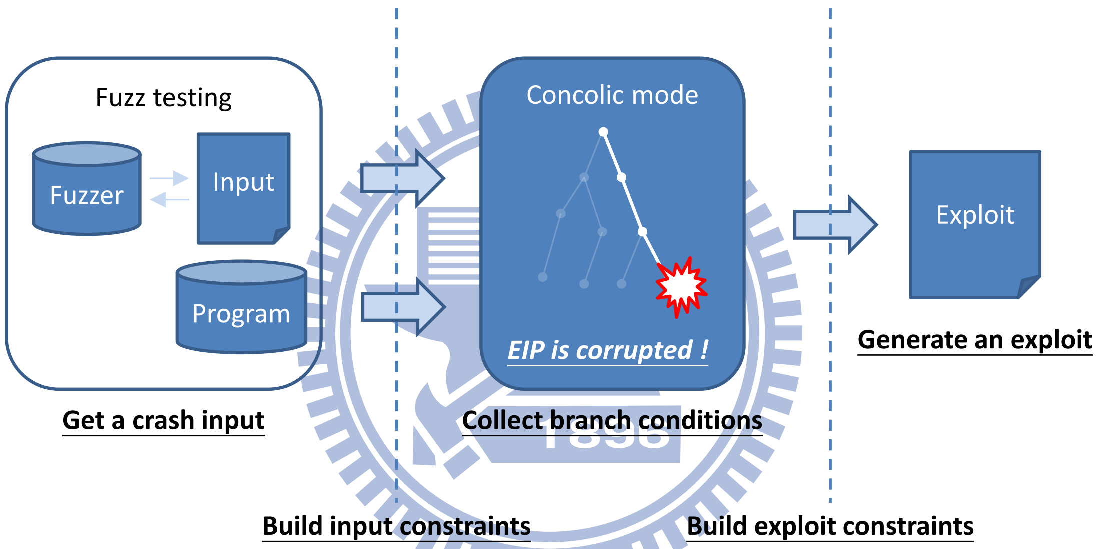


Figure 13: The process of concolic-mode simulation with fuzzer tools

3.4.2 Code Selection

Because S^{2}E performs symbolic execution on a whole operating system, the path explosion is very heavily whenever the symbolic data are passed to library or kernel. On the other hand, the constraints induced by library or kernel are usually complex and huge, and constraint solvers often get stuck in them. For example, if the first argument of fopen() function, which is a path of the file to be opened, is symbolic, the constraint solvers will get a time-out error or hang in S^{2}E. But, those paths in library or kernel are often uninterested for exploit generation. In order to avoid exploring those uninterested paths, those library functions should run on concrete execution.

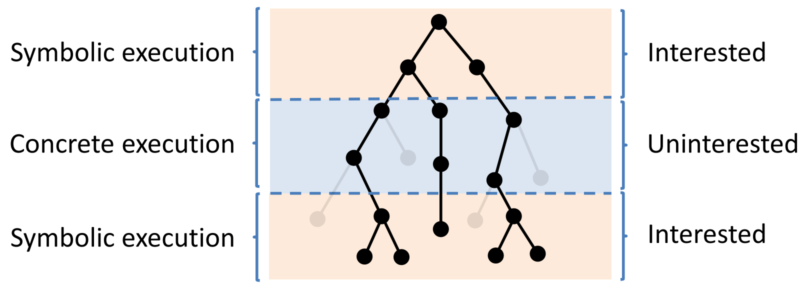


Figure 14: An execution tree with code selection

When the symbolic arguments are changed to concrete values and then passed to those uninterested functions, only one path will be explored. Figure 14 shows that the execution tree is corseted after the program entry the uninterested part and until the program returns to symbolic execution. In order to ensure the return value of those uninterested functions is correct, the concrete values passed to functions must be generated according to current path conditions by constraint solvers. When the function returns, those data that had changed to concrete must be restored to the original symbolic data to keep the symbolic execution.

Chapter 4

Implementation

In this chapter, we explain how our method was implemented on S^{2}E, and what algorithms we used. In order to implement automated exploit generation, many run-time information must be collected and many constraints must be built to reason out an exploit, so the memory model in S^{2}E is an important key to achieve these tasks and is shown on follow section.

Many protections are implemented on compilers or operation systems in real-world systems. In addition to return-to-memory exploits, other two types of exploits, which are return-to-libc and jump-to-register exploit, are implemented in our work to bypass some protections. This work makes our exploit generation more useful in real-world systems.

4.1 Register Corruption Detection

In QEMU, the *CPUX86State* structure, which is defined in Listing 8, is used to simulate the states of x86 CPU, and all register references in guest operating system will be turned into memory references on this structure. When S^{2}E is started, this structure is divided into two parts and stored separately, one of which is *CpuRegistersState* and another is *CpuSystemState*. Listing 9 shows the differences between the two parts, and the CpuRegistersState is a symbolic area where stores the all data in front of EIP register in CPUX86State structure, such as general-purpose registers, but the CpuSystemState part is a concrete only area where stores the other data including EIP register.

S^{2}E translates every guest instruction into TCG IRs, and then translates those TCG IRs into host instructions or LLVM IRs. For example, the *ret* instruction is separated into more detailed operations as shown in Figure 15, and the operation of updating EIP register is converted to a *store* instruction. In QEMU, all memory access operations are handled by a *softmmu* model in order to map the guest addresses to host addresses. Whenever accessing a memory data, S^{2}E checks whether the value of data is symbolic or not in softmmu model. If the value is symbolic, S^{2}E will rerun this translated block from current instruction in KLEE to perform symbolic execution. For detecting EIP register corruption, S^{2}E must check whether the writing target is the location of EIP register and whether the source value is symbolic data whenever KLEE deal with a *store* memory operation on symbolic execution.

Listing 8: The structure of CPUX86State

```
typedef struct CPUX86State {
    /* standard registers */
    target_ulong regs[CPU_NB_REGS];
#if 0
    target_ulong eflags; /* eflags register. During CPU emulation, CC
                        flags and DF are set to zero because they are
                        stored elsewhere */
#endif

    /* emulator internal eflags handling */
    uint32_t cc_op; /* outside of cpu loop, CC_OP is always CC_OP_EFLAGS */
    target_ulong cc_src;
    target_ulong cc_dst;
    target_ulong cc_tmp; /* temporary for rcr/rcl */

    ...

    target_ulong eip;

    int32_t df; /* D flag : 1 if D = 0, -1 if D = 1 */
    target_ulong mflags; /* Mode and control flags from eflags */

    ...

}
```

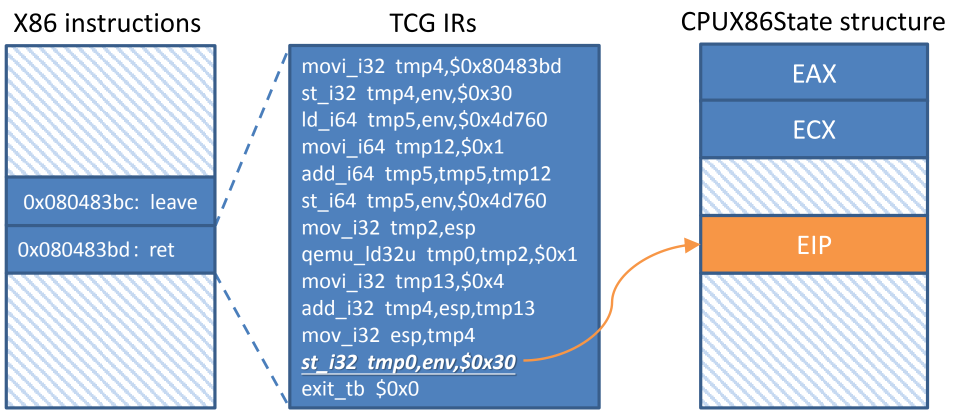


Figure 15: The process of *ret* instruction translation in QEMU

Listing 9: The code of registerCPU() function

```
void S2EExecutor::registerCpu(S2EExecutionState *initialState,
                              CPUX86State *cpuEnv)
{
    ...

    /* Add registers and eflags area as a true symbolic area */
    initialState->m_cpuRegistersState =
        addExternalObject(*initialState, cpuEnv,
                          offsetof(CPUX86State, eip),
                          /* isReadOnly = */ false,
                          /* isUserSpecified = */ false,
                          /* isSharedConcrete = */ false);

    initialState->m_cpuRegistersState->setName("CpuRegistersState");

    /* Add the rest of the structure as concrete-only area */
    initialState->m_cpuSystemState =
        addExternalObject(*initialState,
                          ((uint8_t*)cpuEnv) + offsetof(CPUX86State, eip),
                          sizeof(CPUX86State) - offsetof(CPUX86State, eip),
                          /* isReadOnly = */ false,
                          /* isUserSpecified = */ true,
                          /* isSharedConcrete = */ true);

    initialState->m_cpuSystemState->setName("CpuSystemState");

    ...
}
```

When EIP register is corrupted by symbolic data, the expression of symbolic data must be recorded because it describes which variable and which part of it corrupt EIP register. For example, an expression that represents a 32-bit symbolic data at the first element of an array named *buf* is shown as follow.

$$(ReadLSB\ w32\ 0\ buf)$$

We can build a constraint to control the value of symbolic data. For example, a constraint limit the 32-bit data to zero is shown as follow.

$$(Eq\ 0\ (ReadLSB\ w32\ 0\ buf))$$

Next, we must inject shellcode into memory to determine where EIP register should point to.

4.2 Exploit Generation

4.2.1 Memory Model in S^{2}E

In S^{2}E, memory consists of *MemoryObjects* objects and the actual contents of *MemoryObject* objects are stored in *ObjectState* objects. In an *ObejectState* object, the symbolic data are stored separately from concrete data. The expressions of symbolic data are stored in an array that consists of *Expr* Objects and a pointer named *knownSymbolics* points to it. The concrete data are stored in an array of *uint8_t* and pointed by a pointer named *concreteStore*. In each *ObejectState* object, a *BitArray* object named *concreteMask* is used to record the states of each byte, i.e. the byte is concrete or symbolic. The structure of *ObjectState* objects is shown in Figure 16.

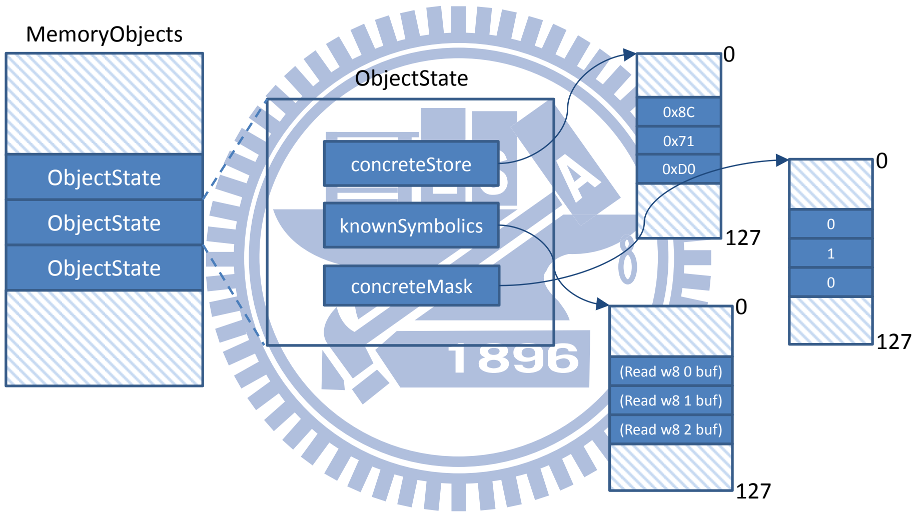


Figure 16: The structure of *ObjectState* object

4.2.2 Finding Symbolic Memory Blocks

The default size of the storage in an *ObjecetState* object is 128 bytes. For finding continuous symbolic data in a memory region, the value of *concreteMask* structures must be checked sequentially object by object. An object can be skipped easily whenever the value of its *concreteMask* structure are all ones, otherwise the locations of every zero in *concreteMask* structure must be recorded to compute the continuous size. For the symbolic blocks crossing objects, it is necessary that to check whether the current symbolic block is connected with the last symbolic block in the last checked object. The algorithm is shown in Algorithm 1.

Algorithm 1: Searching for symbolic blocks

```
Input: Objects : All ObjectState objects to be searched.
Output: V : A set of address and size.
1  foreach obj ∈ Objects do
2      if isNotAllConcrete() then
3          size ← 0
4          for i ← 0 to 127 do
5              if isByteSymbolic(i) then
6                  size ← size + 1
7              else if size ≠ 0 then
8                  address ← getAddress(i)
9                  if V → isConnect(address,size) then
10                     V → updateLastItem(size); /* A part of the last block */
11                 else
12                     V → addNewItem(address,size) /* An independent block */
13                 size ← 0
```

Another program is the search range of memory regions. In Linux memory layout, as shown in Figure 4, the based address of stack is 0xbfffffff and grow downward, so it is very easy to search stack region from this address to down. But heap and data segment are not located at a fixed address for different programs. Therefore, those starting locations must be got dynamically. According to the ELF executable layout, the top of executable files is the program header, which records the all segment information. The program header can be analyzed at address 0x08048000, which is the location where binary is loaded at, to get the location and size of data segment. On the other hand, because heap region is behind data segment and grow upward, the based address of heap can be got by adding the starting address and size of data segment.

4.2.3 Shellcode Injection

In order to determine whether shellcode can be stored in the potential buffers found by previous step, each symbolic expression of a symbolic block must be read to build constraints that restrict each byte of symbolic data to each byte of shellcode sequentially byte by byte. For example, the constraints that inject shellocde into an array named *buf* is shown as follow.

$$(Eq\ 31\ (Read\ w8\ 0\ buf))$$
$$(Eq\ C0\ (Read\ w8\ 1\ buf))$$
$$(Eq\ 89\ (Read\ w8\ 2\ buf))$$
$$(Eq\ C2\ (Read\ w8\ 3\ buf))$$
$$\vdots$$

Next, the shellcode constraints are passed to an SMT solver with path conditions to validate their feasibility.

For the best location of shellcode, the rule is that NOP sled is as larger as possible. Therefore, all the symbolic blocks are sorted by size, and shellcode is injected from the end of the largest symbolic block firstly. In addition to building the shellcode constraints, a new constraint must be added to ensure EIP register can point to the range between the starting address of shellcode and the top of the symbolic block. Even if EIP register cannot point to the starting location of shellcode precisely, it may feasible because NOP sled will extend the entry point later. If those constraints are infeasible, the location of shellcode injection is always shifted one byte forward to try the new location.

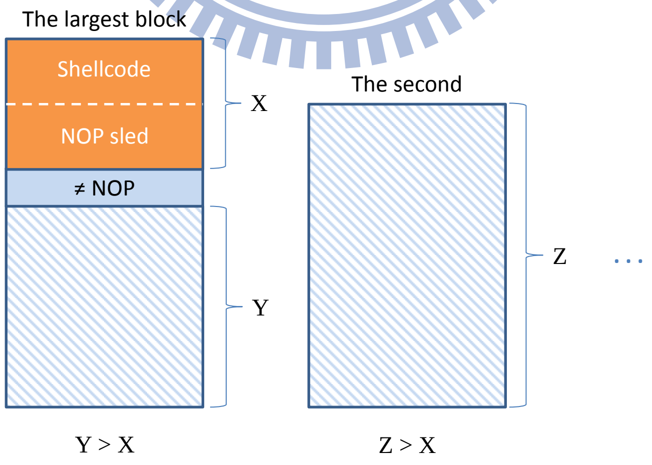


Figure 17: The process of searching symbolic blocks

In addition, shellcode will keep to be injected to the remain room of the current block or next blocks when those sizes are larger than the sum of the shellcode size and the current longest NOP sled size. For example, consider Figure 17, the sum of shellcode size and current NOP size is X, but it is smaller than Y and Z, so shellcode and NOP sled will keep to be injected in next blocks and the remain room of the current block. The algorithm is shown in Algorithm 2.

Algorithm 2: Injecting shellcode

Input: V : A set of address and size of symbolic blocks. Shellcode : A shellocde string. PC : Path conditions.

Output: ShellcodeAddress : The starting location of shellcode injection. MaxNopSize : The max size of NOP sled.

```
1  foreach $v \in V$ do
2      if $size_v \geq strlen(Shellcode)$ then
3          address ← $address_v$ + $size_v$ - strlen(Shellcode)
4          MaxNopSize ← -1
5          while $address \geq address_v$ do
6              c1 ← injectShellcodeAt(address) /* Build shellcode constraints */
7              c2 ← eipBetween(address, $address_v$) /* Build eip constraints */
8              if $Verify(PC \wedge c1 \wedge c2)$ then
9                  nopSize ← NOPSled(address, $address_v$)
10                 if $nopSize > MaxNopsize$ then
11                     MaxNopSize ← NopSize
12                     ShellcodeAddress ← address
13                 if $(address - address_v) > strlen(shellcode) + MaxNopSize)$ then
14                     address ← address - nopSize
15                 else
16                     break
17             else
18                 address ← address - 1
```

4.2.4 NOP Sled

NOP sled aims to generate the more reliable exploits that increase chances of success. The method is to insert NOP instructions in front of the shllecode as many as possible, and make EIP register point to the range. For efficiency, binary search-like algorithm is used to determine the longest length of NOP sled rather than insert NOP instructions byte by byte. Whenever binary search finds a range that EIP register can point to, NOP instructions will be tried to fill this range sequentially to check whether both conditions are feasible simultaneously. If it is infeasible, the range is reduced, otherwise extend, and so on. The detail algorithm is shown in Algorithm 3 and process in Figure 18.

Algorithm 3: NOP sled

```
Input: Start : The starting address of NOP sled. End : The end address of NOP sled. PC
       : Path conditions.
Output: NopSize : The size of NOP sled.

1  min ← End
2  max ← Start
3  mid ← min + (max-min)/2
4  while $min \leq max$ do
5      c1 → eipBetween(Start,mid) /* Build eip constraints */
6      if Verify($PC \wedge c1$) then
7          c2 → injectNopBetween(Start,mid)/* Build NOP constraints */
8          if Verify($PC \wedge c2$) then
9              NopSize ← Start - mid
10             max ← mid - 1
11         else
12             min ← mid + 1
13     else
14         max ← mid - 1
15     mid ← min + (max-min)/2
```

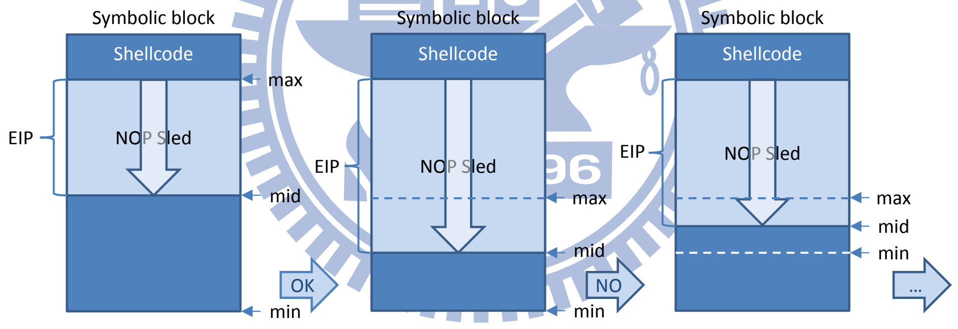


Figure 18: The process of NOP sled

After the longest length of NOP sled is gotten, the next step is to make EIP register point to the middle of NOP seld as close as possible. Because the number of NOP sled maybe is large, the constraint solver is used to reason out the suitable location that EIP register points to. To help a constraint solver to compute the address as close the middle of NOP sled as possible, a constraint is added to limit the range. First, the range is a point in the middle of NOP sled, and the constraints are passed to a constraint solver to get the solution. If it is infeasible, the range is extended twice each time, and so on. This process always can get a solution, because the previous step guarantees that the EIP register can point to the range of NOP sled. The process as shown in Figure 19 and the algorithm is shown in Algorithm 4.

Algorithm 4: Determining the value of EIP register

Input: NopSize : The size of NOP sled. Start : The starting address. End : The end address. PC : Path conditions.

Output: EipAddress : The address where EIP register points at.

```
1  mid ← Start - (NopSize/2)
2  range ← 0
3  repeat
4      if $mid$ - $range \leq Start$ - $NopSize$ then
5          low ← Start-NopSize
6      else
7          low ← mid-range
8      if $mid + range \geq Start$ then
9          hi ← Start
10     else
11         hi ← mid + range
12     c ← eipBetween(low, hi) /* Build eip constraints */
13     if $range = 0$ then
14         range ← 1
15     else
16         range ← range * 2
17 until $Verify(PC \wedge c)$ ;
18 EipAddress ← getValue(PC ∧ c)
```

Finally, when the staring address of shellcode, the size of NOP sled and the location where EIP register points to all are determined and feasible, the constraint solver will solve the final path conditions to generate the exploit that performs the malicious task in the shellocde.

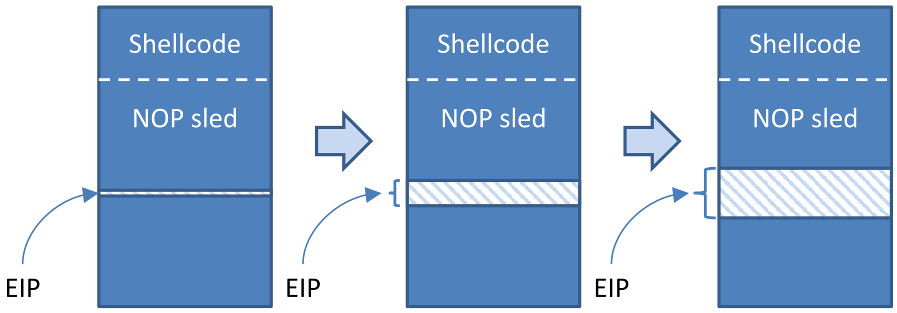


Figure 19: The process of determining where EIP register point to

4.2.5 Other Types of Exploit

4.2.5.1 Return-to-libc

A return-to-libc attack is a technique to bypass non-executable memory regions, such as W⊕X protection. It redirects control flow to functions in C runtime library, such as system(), and injects the arguments of the function into stack manually to fake the behavior of function callers. Because runtime library is always executable and loaded by operating systems, a return-to-libc attack can perform malicious tasks by executing library code and bypass executable space protection. Consider Figure 5, function callers have to push arguments and return address into stack when calling functions. About exploits, it doesn't really matter where the libc function call returns to, but the arguments are key to perform the tasks we desired.

Table 3: The differences between return-to-memory and return-to-libc exploit

Exploit	Shellocde Injection	NOP Sled
Return-to-memory	shellocde	NOP instruction(0x90)
Return-to-libc	"/bin/sh"	whitespace(0x20)

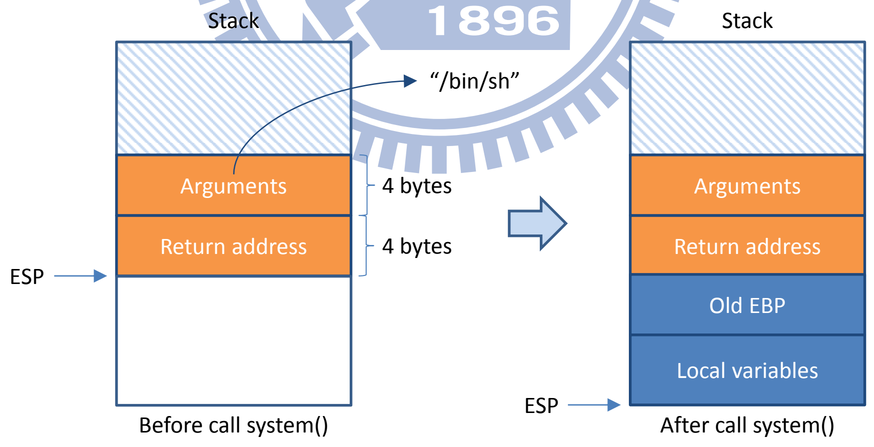


Figure 20: The process of return-to-libc exploit generation

Taking system("/bin/sh") for example, which will open a shell, the only one argument is a pointer points to the string "/bin/sh" as shown in Figure 20. The process of return-to-libc exploit generation is very similar to return-to-memory. As Table 3 shows that the steps of shellcode injection and NOP sled are still applying to return-to-libc exploit generation. But, shellcode injection injects the string "/bin/sh" instead of a shellcode, and NOP slep fills whitespace characters rather than NOP instructions. Finally, the exploit constraints limit the argument to the location of the middle of whitespace sled and redirect EIP register to the location of system() function in C runtime library.

4.2.5.2 Jump-to-register

Stack is the most common memory region for shellocode injection, but ASLR randomizes the based address of stack so that control flow is very difficult to jump to shellcode accurately. A large NOP sled may bypass ALSR, but it not always feasible. A jump-to-register attack is a technique to bypass ASLR. It uses a register that points to a shellcode as a trampoline to execute the malicious tasks. For example, EAX register is usually used to store the return value of functions. Strcpy() function returns a pointer points to the location of buffer, and EAX register is often used to store the address. If a "call %eax" instruction can be found in code segment, which is very common, and shellcode can be injected into the buffer EAX register points to, control flow will be redirected to execute this instruction and jump to shellcode.

In addition, a jump-to-esp attack is also a common and reliable technique without NOP sled and guessing stack offset in Windows and old version of Linux. Because return addresses are always popped to make ESP register points to their next address when functions return, shellcode can be injected behind the return address and uses ESP register as a trampoline. If a "jmp %esp" instruction can be found in code segment, a jump-to-esp exploit can be generated to bypass ASLR. The process as shown in Figure 21.

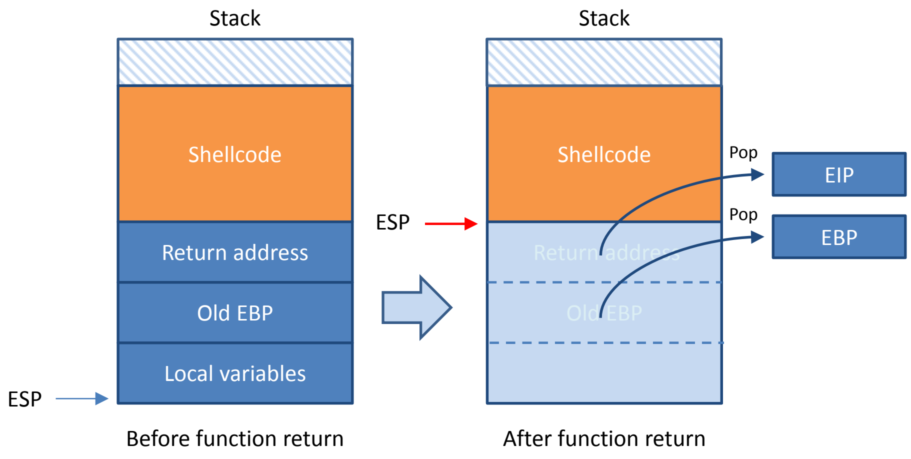


Figure 21: The process of jump-to-register exploit generation

In order to generate jump-to-register exploits, code segment must be searched to find the related instructions such as "call %eax" and "jmp %esp". If the related instructions are found and the memory region that register points to is symbolic, shellcode will be injected into the location, and EIP register will be redirected to execute the related instruction. In addition, if there is no any usable related instruction in code segment, data segment may be searched to find a two-byte symbolic data to inject the related instruction because data segment is unaffected by ASLR. For example, "jmp %esp" instruction is 0xffe4 and "call %eax" instruction is 0xffd0.

4.3 Pointer Corruption Detection

When accessing a memory address, S^2E will check whether the address is symbolic or not. If it is symbolic, S^2E must determine an explicit location before keeping program execution. S^2E uses a binary search to find all locations that the symbolic address can point to, and forks executions to explore each address. Before S^2E handles a symbolic address, the address can try to point to sensitive data, such as return address and GOT. If it is feasible, those sensitive data will be tainted and corrupt EIP register later. Otherwise, the symbolic address will be changed to taint other concrete data because it may help exploit generation if other vulnerabilities corrupt EIP register later. For example, we can change the address to taint data segment so that shellcode can inject to there to bypass ASLR protection.

4.4 Concolic-mode Simulation

To simulate concolic testing on S^2E, we execute the program under test with input data, such as arguments and environment variables, and the main tasks are building input constraints and collecting branch conditions. As the previous note about memory model shown in Figure 16, the concrete values are stored separately from symbolic data, but the concrete values are ignored since the variables are marked as symbolic. In order to get the input data at run time, we can read the last concrete value of symbolic variables from *concreteStore* structure by hand and build constraints to limit the each symbolic expression to its concrete value. A vector container is used to save all input constraints because it is very easy to delete some constraints when they are unnecessary, and to combine with every constraint into a complete input constraint when it is needed.

In addition, a branch condition that takes only one path won't be added to path conditions on symbolic execution because of redundancy. But concolic testing may need to collect those branch conditions because it always explores only one path in branches and the result is caused by adding extra input constraints. Therefore, whenever a branch is encountered and its both paths are feasible on symbolic execution, the SMT solver redetermines the branch conditions associate with input constraints and adds the branch conditions by force. The process is shown in Figure 22.

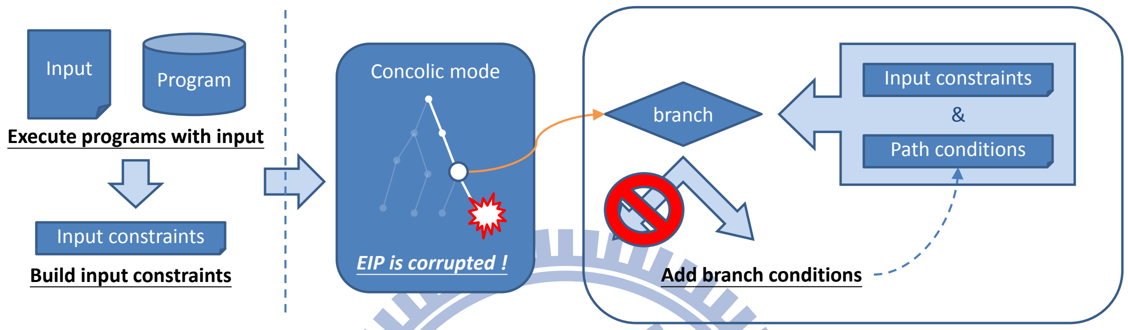


Figure 22: The process of concolic-mode simulation

In addition to branches, symbolic addresses also fork executions on symbolic execution. When accessing a memory address whose value is symbolic, S^2E cannot determine where it should access. So, S^2E forks executions to try to access every address where the symbolic address can point to. In concolic mode, input constraints are still used to help SMT solvers to determine a location on symbolic address. But, when pointer corruption happen, the address could be redirected to other addresses as section 4.3 notes. So, when a symbolic address is redirected, the input constraints that associate with symbolic address will be deleted from the vector container in order to avoid conflict.

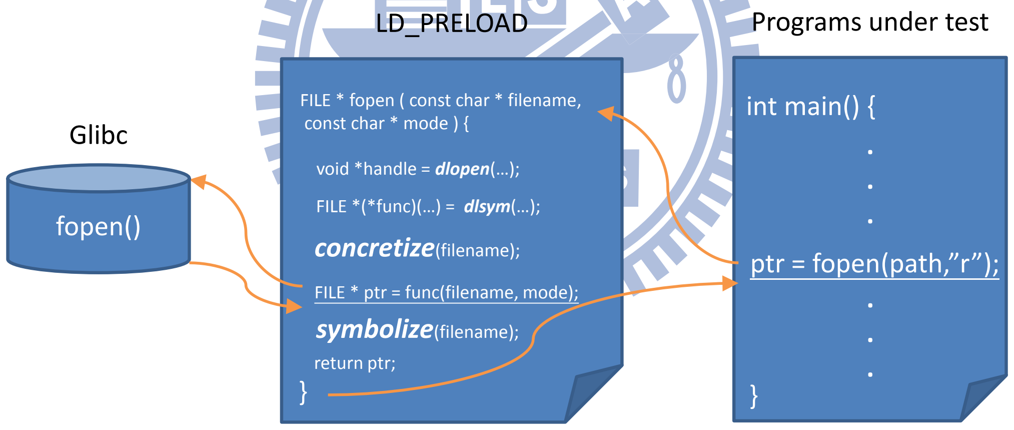
It is very easy to switch between concolic-mode simulation and original symbolic execution because the memory model of S^2E doesn't be modified. If symbolic execution will be performed, we just set the input constraints to be "true" because path conditions won't be changed when they perform an "AND" logical operation with a "true" expression.

4.5 Code Selection

Because the concrete value of a variable is stored separately from symbolic data, the switch of a variable between concrete and symbolic is very easy. The *concreteMask* structure is used to record the states of a memory region, and we can modify the boolean value to change a byte value from concrete to symbolic or vice versa.

To concretize a symbolic data, we pass the symbolic expression and path conditions to an SMT solver to find a concrete solution and write the concrete value back to the address of symbolic data. Because the variable has became concrete, all the code accessing the variable will be executed concretely. So, we concretize the arguments of uninterested functions to stop symbolic execution to explore the paths in them.

To restore a concrete variable to symbolic, we invert the boolean value in *concreteMask* structure to ignore the concrete value and keep the original symbolic expression to perform symbolic execution on this variable.



```
FILE * fopen ( const char * filename,
 const char * mode ) {
   void *handle = dlopen(...);
   FILE *(*func)(...) = dlsym(...);
   concretize(filename);
   FILE * ptr = func(filename, mode);
   symbolize(filename);
   return ptr;
}
```

```
int main() {
        .
        .
        .
 ptr = fopen(path,"r");
        .
        .
        .
}
```

Figure 23: An example for code selection

It is very easy that using this method to select a code to run on concrete execution or symbolic execution. In Linux, LD_PRELOAD environment variable can intercept the library functions and jump to the functions we writing. To cooperate with this environment variable, those uninterested library functions can be intercepted automatically, and run them on concrete execution. Figure 23 shows an example that intercepts *fopen* function, and performs concrete execution on it. In addition, some functions just print messages to screen without return value, such as perror(), so those functions can be skipped using this method directly to speed the process of exploit generation.

Chapter 5

Experimental Results

In order to evaluate our work, three experiments were completed. The first part generated exploits for five different control-flow hijacking vulnerabilities to show our method can handle all vulnerabilities that corrupt EIP register and some vulnerabilities that corrupt pointers. The second experiment demonstrated that return-to-libc and jump-to-register exploits we generated can bypass some protections in real-world systems.

In the final experiment, we generated exploits for ten real-word programs to evaluate the results and prove our method can apply in real-world applications. In these ten real-world programs, seven programs were chosen from the benchmarks of AEG to demonstrate that our method can handle at least all cases that AEG can address.

5.1 Testing Method and Environment

In this chapter, we demonstrate the results of automated exploit generation for vulnerable sample code and real-world programs. All experiments were performed on a 2.66 GHz Intel Core 2 Quad CPU with 4 GB of RAM, and the host environment is Ubuntu 10.10 64-bit. The guest environment is Debian 5.0 32-bit with default settings of QEMU, which is a 266MHz Pentium II (Klamath) CPU with 128 MB of RAM.

Mostly, the programs under test were compiled by GCC 4.3.2 and ran on Glibc 2.7, which are the default in Debian 5.0. But some programs used GCC 3.4.6 or Glibc 2.3.2 to generate exploits because the default version protects main function against stack buffer overflow or performs heap hardening integrity checks to stop heap overflow.

We used an end-to-end approach to generate exploits on binary executables without modifying the source code, i.e. the source code are not necessary. The method we used is forking a new process to execute the program under test and passing the symbolic data to it from outside. In Debian 5.0, ASLR is enabled by default so that the based address of stack and heap is randomized. Therefore, ASLR was disabled for generating and testing all exploits except jump-to-register exploits.

5.2 Sample Code

In the first part of our experiments, we generated exploits for five common types of vulnerabilities. Because our method is based on EIP register corruption detection instead of specific vulnerabilities, it can handle different types of vulnerabilities. We designed five sample code for five different vulnerabilities and four corrupted data, and performed automated exploit generation on them. The results are shown in Table 4, and the all sample code and exploits are in Appendix A. In the results, the format of wall time is *(exploit reason time / total time)*.

In this experiment, the source input of all sample code was argument and its length was 100 characters. We experimented on concolic-mode simulation and traditional symbolic execution to compare the efficiency of both. In symbolic execution, depth-first search (DFS) was used to explore a symbolic execution tree. The heap overflow code was executed on Glibc 2.3.2, because some protections that check pointer consistency have included since version of Glibc 2.3.6. In addition, this exploit generation cooperated with *libfmtb*[1] library to build format strings to exploit format string vulnerabilities.

Stack buffer overflow and uninitialized variable vulnerability corrupt EIP register directly, and other three vulnerabilities taint EBP register or pointers to corrupt EIP register indirectly. As the results show, the average of total time was 3.67 seconds in concolic mode and the exploit reason time was 0.35 seconds. On average, symbolic execution spent 302.67 seconds on getting an exploit and 0.47 seconds on reasoning out it. Concolic mode was faster about 100 times than symbolic execution because it just explored only one suspicious path. In the experiments of symbolic execution, format string vulnerability got an out-of-memory error because symbolic execution attempted to explore all paths in snprintf() function, which performs a complex behavior.

[1]`http://packetstormsecurity.org/files/26173/`

Table 4: The results of exploit generation for sample code

Vulnerability	Corrupted Data	Concolic Wall Time(sec.)	Symbolic Wall Time(sec.)
Stack buffer overflow	Return address	0.61/3.59	0.69/303.59
Heap buffer overflow	Pointer	0.24/3.16	0.25/301.73
Off-by-one overflow	EBP register	0.46/3.24	0.51/302.14
Uninitialized variable	Function pointer	0.41/3.59	0.46/303.23
Format string	Pointer	0.05/4.81	–
Average		0.35/3.67	0.47/302.67

5.3 Other Types of Exploits

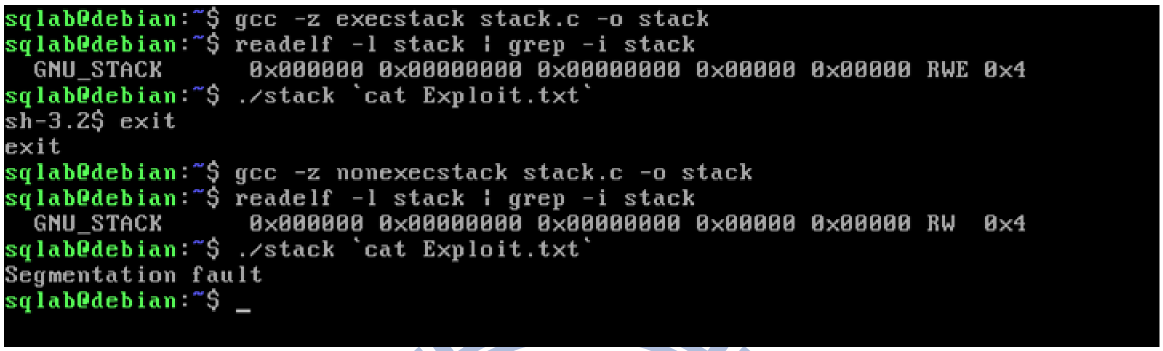
In this experiment, we demonstrated exploit generation for other types of exploits to bypass some protection mechanisms. Because our work implemented return-to-libc and jump-to-register exploit generation, we showed the results of both exploit generation and used the generated exploits to bypass non-executable stack or ASLR protection.

Because return-to-libc and jump-to-register exploit generation don't apply to all cases, i.e. only suitable for some special cases, we chose the sample code of stack buffer overflow vulnerability to demonstrate these experiments, and all experiments were all performed on concolic mode. The two generated exploits are also shown in Appendix A.

Table 5: The run-time information of rerun-to-libc exploit generation

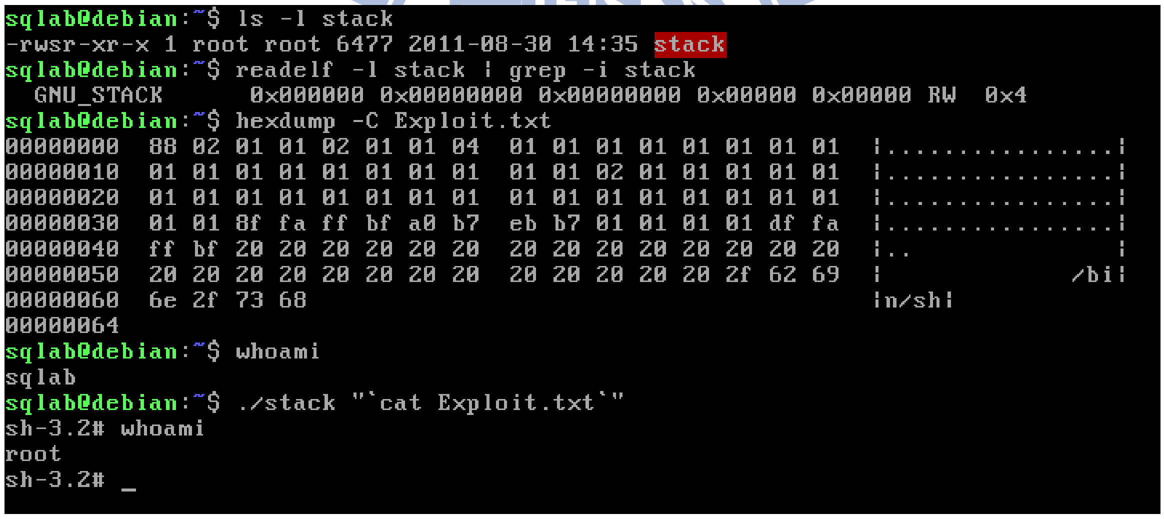
Run-time Information	
EIP register	(ReadLSB w32 54 arg)
ESP register	0xbffff8e0 (value:(ReadLSB w32 58 arg))
ESP register + 4	0xbffff8e4 (value:(ReadLSB w32 62 arg))
Address of system()	0xb7ebb7a0
Potential shellcode buffers	0xbffffa8f (size:100 bytes)
	0xbffff8a6 (size:100 bytes)

In the return-to-libc experiment, we aimed to use system() function to execute "/bin/sh" command. The run-time information at exploit generation are shown in Table 5. The location where ESP register pointed at and ESP register + 4, which is the address to insert argument, were all symbolic and the potential shellcode buffers were large enough to insert the string “/bin/sh”, so this vulnerable program satisfied the all conditions for return-to-libc exploit generation. The total time of this experiment was 3.25 seconds, in which the time spent on exploit reason was 0.34 seconds. Figure 24 shows the return-to-stack exploit got a “Sementation fault” error in non-executable stack, and Figure 25 shows the process of using the return-to-libc exploit to bypass non-executable stack protection.



```
sqlab@debian:~$ gcc -z execstack stack.c -o stack
sqlab@debian:~$ readelf -l stack | grep -i stack
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
sqlab@debian:~$ ./stack `cat Exploit.txt`
sh-3.2$ exit
exit
sqlab@debian:~$ gcc -z nonexecstack stack.c -o stack
sqlab@debian:~$ readelf -l stack | grep -i stack
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
sqlab@debian:~$ ./stack `cat Exploit.txt`
Segmentation fault
sqlab@debian:~$ _
```

Figure 24: A return-to-stack exploit is used in executable and non-executable stack



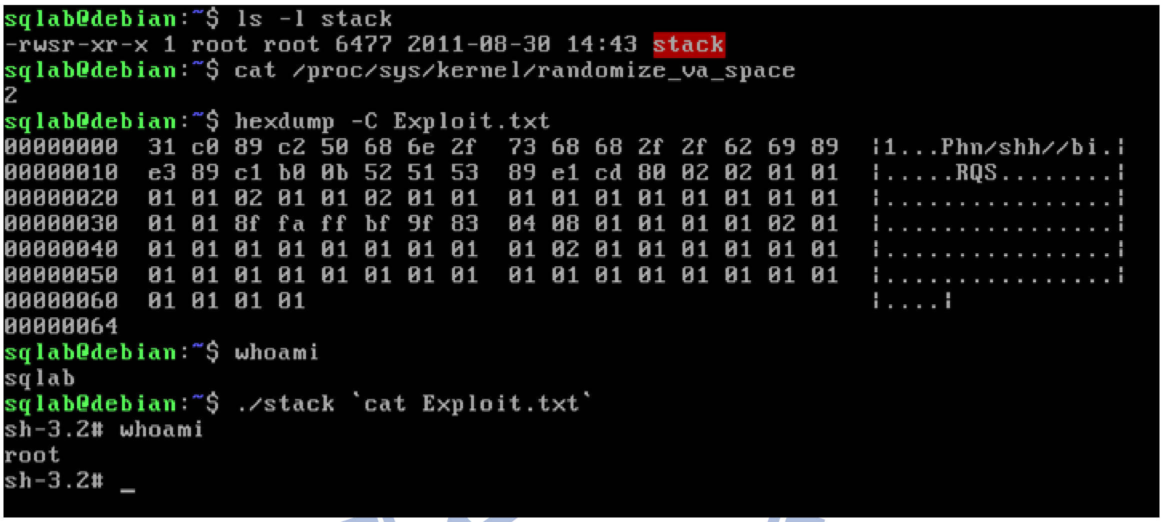
```
sqlab@debian:~$ ls -l stack
-rwsr-xr-x 1 root root 6477 2011-08-30 14:35 stack
sqlab@debian:~$ readelf -l stack | grep -i stack
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
sqlab@debian:~$ hexdump -C Exploit.txt
00000000  88 02 01 01 02 01 01 04  01 01 01 01 01 01 01 01  |................|
00000010  01 01 01 01 01 01 01 01  01 01 02 01 01 01 01 01  |................|
00000020  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000030  01 01 8f fa ff bf a0 b7  eb b7 01 01 01 01 df fa  |................|
00000040  ff bf 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |..              |
00000050  20 20 20 20 20 20 20 20  20 20 20 20 20 2f 62 69  |             /bi|
00000060  6e 2f 73 68                                       |n/sh|
00000064
sqlab@debian:~$ whoami
sqlab
sqlab@debian:~$ ./stack "`cat Exploit.txt`"
sh-3.2# whoami
root
sh-3.2# _
```

Figure 25: A return-to-libc exploit bypasses non-executable stack in Debian 5

Table 6 shows the run-time information at jump-to-register exploit generation. A “call %eax” instruction was found at address 0x0804839f and EAX register pointed to the starting location of a symbolic region exactly, so this vulnerable program can generate jump-to-register exploit to bypass ASLR. The total time of this experiment was 3.16 seconds, in which the time spent on reasoning out the exploit was 0.06 seconds. The process of using this exploit to bypass ASLR in Debian 5.0 is shown in Figure 26.

Table 6: The run-time information of jump-to-register exploit generation

Run-time Information	
EIP register	(ReadLSB w32 54 arg)
Usable trampoline registers	EAX
EAX register	0xbffff8a6 (value:(ReadLSB w32 0 arg))
Address of “call %eax” instruction	0x0804839f
Potential shellcode buffers	0xbffffa8f (size:100) 0xbffff8a6 (size:100)



```
sqlab@debian:~$ ls -l stack
-rwsr-xr-x 1 root root 6477 2011-08-30 14:43 stack
sqlab@debian:~$ cat /proc/sys/kernel/randomize_va_space
2
sqlab@debian:~$ hexdump -C Exploit.txt
00000000  31 c0 89 c2 50 68 6e 2f  73 68 68 2f 2f 62 69 89  |1...Phn/shh//bi.|
00000010  e3 89 c1 b0 0b 52 51 53  89 e1 cd 80 02 02 01 01  |.....RQS........|
00000020  01 01 02 01 01 02 01 01  01 01 01 01 01 01 01 01  |................|
00000030  01 01 8f fa ff bf 9f 83  04 08 01 01 01 01 02 01  |................|
00000040  01 01 01 01 01 01 01 01  01 02 01 01 01 01 01 01  |................|
00000050  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000060  01 01 01 01                                       |....|
00000064
sqlab@debian:~$ whoami
sqlab
sqlab@debian:~$ ./stack `cat Exploit.txt`
sh-3.2# whoami
root
sh-3.2# _
```

Figure 26: A jump-to-register exploit bypasses ASLR in Debian 5

5.4 Real-world Programs

In the final part of experiments, we generated exploits for real-world programs. Because real-world programs are more large and complex than the sample code, this experiment demonstrated that our method is effective and practical in real-world applications.

We chose seven programs from benchmarks of AEG and found three new vulnerable programs released in recent years to perform this experiment. The 10 real-world programs were evaluated by an end-to-end approach, i.e. all programs were tested in binary forms, and the vulnerabilities of these programs were all stack buffer overflow. Concolic-mode simulation was used to perform exploit generation on all programs, and code selection intercepted functions associating with file-related operations or pure error feedback, such as fopen() and perror(), to speed up the process. Table 7 shows the results of 10 real-word programs.

Table 7: The results of exploit generation for real-world programs

Program	Version	Input Source	Input Length	Wall Time(sec.)	Advisory ID.
aeon	0.2a	Env. Var.	550	12.17/310.29	CVE-2005-1019
iwconfig	V.26	Arguments	85	1.39/34.49	BID-8901
glftpd	1.24	Arguments	300	2.66/52.73	OSVDB-16373
ncompress	4.2.4	Arguments	1050	46.07/2046.48	CVE-2001-1413
htget	0.93	Arguments	276	6.38/153.10	CVE-2004-0852
expect	5.43	Env. Var.	300	7.49/179.99	OSVDB-60979
rsync	2.5.7	Env. Var.	201	2.18/36.58	CVE-2004-2093
acon	1.0.5	Env. Var.	1300	95.25/3877.75	CVE-2008-1994
gif2png	2.5.3	Arguments	1080	65.69/2320.57	CVE-2009-5018
hsolink	1.0.118	Arguments	1050	82.59/2504.66	CVE-2010-2930

As the results show, *iwconfig* spent 34.49 seconds on exploit generation, and it was the shortest one. On the other hand, the slowest was *acon* which spent about 64 minutes. According to the results, the speed was proportional to the length of program input, because the more symbolic data exist, the more code may perform on symbolic execution. In addition, the longer symbolic data will bring huge and complex constraints, and SMT solvers must spend a lot of time on constraint solving.

In order to reduce the overhead of SMT solvers and speed up the process, code selection was used to concretize arguments of uninteresting functions. In this experiment, *aeon*, *htget* and *acon* intercepted fopen(); *ncompress* intercepted __lxstat() and perror(); *gif2png* intercepted fopen() and perror(); *expect* intercepted open(); *rsync* intercepted vsnprintf(); *hsolink* intercepted system(). Those functions related with file operations were often make constraint solvers stick, and perror() function just printed error messages without return value and doesn't influence exploit generation, so we filtered these functions to speed up the process.

In this experiment, we performed exploit generation on real-world programs and produced exploits for those applications successfully. The results show that the average of total time was about 19 minutes to generate an exploit for real-world programs, and the quality of exploits was good because they contained the longest NOP sled to increase the chances of successful attacks. For this reason, our method is powerful and practical in automated exploit generation for real-world programs.

Chapter 6

Conclusion and Further Work

In this final chapter, we summarize our work and list some further work to perfect our method. Because the related work for automated exploit generation is few at present, our work is a major contribution for this issue. The last section lists some issues that make our work support more applications and operating systems, and more useful for real-world environments.

6.1 Conclusion

In this thesis, we implemented the automated exploit generation and built it on S^2E, which is a new platform for symbolic execution. We aimed at generating control flow hijacking attacks, and proposed an easy and precise method to achieve this goal. In contract with AEG, EIP register corruption detection is a comprehensive idea to reason out precise exploits and to deal with many control flow hijacking vulnerabilities.

We implemented concolic-mode simulation to perform concolic testing on symbolic execution. This is a flexible and easy method to switch between symbolic execution and concolic testing mode without modifying memory model. In addition, code selection helped S^2E to filter uninteresting functions so that symbolic execution explored interesting code more effectiveness and speeded up the process of exploit generation.

In order to evaluate our method, we experimented on a variety of vulnerable sample code to demonstrate that it can address different kinds of control flow hijacking vulnerabilities, and on real-world programs to prove that it is feasible and powerful. In addition, we generated other types of exploits to bypass some protections to show it is useful for real environments.

6.2 Further Work

To perfect the automated exploit generation, many further work can be implemented as follow.

- A more perfect end-to-end system
    - Real-world programs usually are released in the form of binary executables without source code. In order to operate on binaries directly, many input sources must be handled, such as standard input, environment variables, sockets, and files. Therefore, this is an important further work to generate exploits for more real-world applications.
- Exploit generation for other operating systems
    - For example, Windows is the most common operating system on personal computers, so exploit generation for Windows applications is very useful. Fortunately, our method is most suitable for other operating systems. The differences are memory layout and protection mechanisms for different systems. For example, Windows reserves 2 GB space for kernel, but Linux only reserves 1 GB. Therefore, the search range of memory for shellcode injection is different from Linux.
- More types of exploits
    - As we tried to generate return-to-lib and jump-to-register exploits, other types of exploits can attack other different protections. For example, return-oriented programming is a technique to bypass ASLR and W$\oplus$X without shellcode. In addition, shellocde design is also useful for exploits. For example, it is very powerful that shellcode is divided into many small parts, and injected to different regions. Even if there is no any symbolic block in memory large enough to injection a complete shellcode, this skill makes the exploits still work by chaining those small parts of shellcode together.

References

[1] T. Wang, T. Wei, Z. Lin, and W. Zou, “IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS’09)*, San Diego, California, USA, February 2009.

[2] D. Molnar, X. C. Li, and D. Wagner, “Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs,” in *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009, pp. 67–82.

[3] C. Cadar and D. R. Engler, “Execution Generated Test Cases: How to Make Systems Code Crash Itself,” in *Proceedings of the 12th International SPIN Workshop on Model Checking Software*, San Francisco, CA, USA, August 2005, pp. 2–23.

[4] C. S. Păsăreanu and W. Visser, “A survey of new trends in symbolic execution for software testing and analysis,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, no. 4, pp. 339–353, 2009.

[5] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask),” in *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010)*, Berleley/Oakland, California, USA, May 2010, pp. 317–331.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, “Symbolic execution for software testing in practice: preliminary assessment,” in *Proceedings of the 33rd International Conference on Software Engineering (ICSE’11)*, Waikiki, Honolulu , HI, USA, May 2011, pp. 1066–1071.

[7] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI’05)*, Chicago, IL, USA, June 2005, pp. 213–223.

[8] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE’05)*, Lisbon, Portugal, September 2005, pp. 263–272.

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated Whitebox Fuzz Testing,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS’08)*, San Diego, California, USA, February 2008.

[10] D. A. Molnar and D. Wagner, “Catchconv: Symbolic execution and run-time type inference for integer conversion errors,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-23, February 2007.

[11] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI’07)*, San Diego, California, USA, June 2007, pp. 89–100.

[12] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI’08)*, San Diego, California, USA, December 2008, pp. 209–224.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “EXE: automatically generating inputs of death,” in *Proceedings of the 13th ACM Conference on Computer and Communications Security(CCS’06)*, Alexandria, VA, USA, October - November 2006, pp. 322–335.

[14] C. Lattner and V. S. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO’04)*, San Jose, CA, USA, March 2004, pp. 75–88.

[15] R. A. Santelices and M. J. Harrold, “Exploiting program dependencies for scalable multiple-path symbolic execution,” in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis (ISSTA’10)*, Trento, Italy, July 2010, pp. 195–206.

[16] P. Boonstoppel, C. Cadar, and D. R. Engler, “RWset: Attacking Path Explosion in Constraint-Based Test Generation,” in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’08)*, Budapest, Hungary, March - April 2008, pp. 351–366.

[17] M. Delahaye, B. Botella, and A. Gotlieb, “Explanation-Based Generalization of Infeasible Path,” in *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST’10)*, Paris, France, April 2010, pp. 215–224.

[18] S. Bardin and P. Herrmann, “Pruning the Search Space in Path-Based Test Generation,” in *Proceedings of the Second International Conference on Software Testing Verification and Validation (ICST’09)*, Denver, Colorado, USA, April 2009, pp. 240–249.

[19] V. Ganesh and D. L. Dill, “A Decision Procedure for Bit-Vectors and Arrays,” in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV’07)*, Berlin, Germany, July 2007, pp. 519–531.

[20] C. Barrett and C. Tinelli, “CVC3,” in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV’07)*, Berlin, Germany, July 2007, pp. 298–302.

[21] B. Dutertre and L. de Moura, “The Yices SMT solver,” Computer Science Laboratory, SRI International, Tech. Rep., August 2006.

[22] L. M. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’08)*, Budapest, Hungary, March - April 2008, pp. 337–340.

[23] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, “HAMPI: a solver for string constraints,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA’09)*, Chicago, IL, USA, July 2009, pp. 105–116.

[24] J. Caballero, P. Poosankam, S. McCamant, D. Babić, and D. Song, “Input generation via decomposition and re-stitching: finding bugs in Malware,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS’10)*, Chicago, Illinois, USA, October 2010, pp. 413–425.

[25] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, “Cloud9: a software testing service,” *Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2009.

[26] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, “Parallel symbolic execution for automated real-world software testing,” in *Proceedings of the sixth conference on Computer systems (EuroSys ’11)*, Salzburg, Austria, April 2011, pp. 183–198.

[27] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, “Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications,” in *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, Oakland, California, USA, May 2008, pp. 143–157.

[28] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “AEG: Automatic Exploit Generation,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS’11)*, San Diego, California, USA, February 2011.

[29] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit Hardening Made Easy,” in *Proceedings of the 20th USENIX Security Symposium (USENIX’11)*, San Francisco, CA, USA, August 2011.

[30] S. Heelan and D. Kroening, “Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities,” *MSc Computer Science Dissertation, University of Oxford, UK*, 2009.

[31] C. Miller, J. Caballero, N. M. Johnson, M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “Crash Analysis using BitBlaze,” in *Proceedings of the Black Hat USA 2010*, Las Vegas, US, July 2010.

[32] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: a platform for in-vivo multi-path analysis of software systems,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’11)*, Newport Beach, CA, USA, March 2011, pp. 265–278.

[33] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, Anaheim, CA, USA, April 2005, pp. 41–46.

[34] V. Chipounov and G. Candea, “Dynamically Translating x86 to LLVM using QEMU,” School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, Tech. Rep. EPFL-TR-149975, March 2010.

[35] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, “Selective Symbolic Execution,” in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, Lisbon, Portugal, June 2009.

Appendix A

Sample Code and Exploits

A.1 Shellcode

Listing 10: The used shellcode

```
00000000  31 c0 89 c2 50 68 6e 2f  73 68 68 2f 2f 62 69 89  |1...Phn/shh//bi.|
00000010  e3 89 c1 b0 0b 52 51 53  89 e1 cd 80              |.....RQS....|
0000001c
```

A.2 Stack Buffer Overflow Vulnerability

Listing 11: A sample code for stack buffer overflow vulnerability

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

void a(char *argv)
{
  char buf[50];

  strcpy(buf, argv);
}

int main(int argc, char **argv)
{
  if(argc > 1)
    a(argv[1]);

  return 0;
}
```

Listing 12: A return-to-stack exploit for Listing 11

```
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000010  90 90 90 90 90 90 31 c0  89 c2 50 68 6e 2f 73 68  |......1...Phn/sh|
00000020  68 2f 2f 62 69 89 e3 89  c1 b0 0b 52 51 53 89 e1  |h//bi......RQS..|
00000030  cd 80 8f fa ff bf 9a fa  ff bf 01 01 01 01 01 01  |................|
00000040  01 01 01 01 01 01 01 01  02 01 01 01 01 01 01 01  |................|
00000050  01 01 01 01 01 01 01 01  01 01 01 02 01 01 01 01  |................|
00000060  01 01 01 02                                       |....|
00000064
```

Listing 13: A return-to-libc exploit for Listing 11

```
00000000  88 02 01 01 02 01 01 04  01 01 01 01 01 01 01 01  |................|
00000010  01 01 01 01 01 01 01 01  01 01 02 01 01 01 01 01  |................|
00000020  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000030  01 01 8f fa ff bf a0 b7  eb b7 01 01 01 01 df fa  |................|
00000040  ff bf 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |..              |
00000050  20 20 20 20 20 20 20 20  20 20 20 20 20 2f 62 69  |             /bi|
00000060  6e 2f 73 68                                       |n/sh|
00000064
```

Listing 14: A jump-to-register exploit for Listing 11

```
00000000  31 c0 89 c2 50 68 6e 2f  73 68 68 2f 2f 62 69 89  |1...Phn/shh//bi.|
00000010  e3 89 c1 b0 0b 52 51 53  89 e1 cd 80 02 02 01 01  |.....RQS........|
00000020  01 01 02 01 01 02 01 01  01 01 01 01 01 01 01 01  |................|
00000030  01 01 8f fa ff bf 9f 83  04 08 01 01 01 01 02 01  |................|
00000040  01 01 01 01 01 01 01 01  01 02 01 01 01 01 01 01  |................|
00000050  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
*
00000060
```

A.3 Heap Buffer Overflow Vulnerability

Listing 15: A sample code for heap buffer overflow vulnerability

```c
#include <stdlib.h>
#include <string.h>

void a(char *argv)
{
  char * first, * second;

  first = malloc(80);
  second = malloc(80);
  strcpy(first, argv);
  free(first);
  free(second);
}

int main(int argc, char **argv)
{
  if(argc > 1)
    a(argv[1]);

  return(0);
}
```

Listing 16: An exploit for Listing 15

```
00000000  88 01 01 02 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000010  01 01 04 01 01 01 01 02  01 01 01 01 01 01 02 01  |................|
00000020  01 01 01 01 01 01 02 01  01 01 01 eb 0a 90 90 90  |................|
00000030  90 90 90 90 90 90 90 31  c0 89 c2 50 68 6e 2f 73  |.......1...Phn/s|
00000040  68 68 2f 2f 62 69 89 e3  89 c1 b0 0b 52 51 53 89  |hh//bi......RQS.|
00000050  e1 cd 80 02 fc ff ff ff  a4 fa ff bf 92 fc ff bf  |................|
00000060  01 01 01 40                                       |...@|
00000064
```

A.4 Off-by-one Buffer Overflow Vulnerability

Listing 17: A sample code for off-by-one buffer overflow vulnerability

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

void b(int * text, int i)
{
  int temp[20];

  for( ; i <= 20 ; i++)
    temp[i] = *(text+i);
}

void a(int * argv, int i)
{
  b(argv, i);
}

int main(int argc, char **argv)
{
  int text[25];

  if(argc > 1)
    memcpy(text, argv[1], 100);

  a(text, 0);

  return 0;
}
```

Listing 18: An exploit for Listing 17

```
00000000  8f fa ff bf ad fa ff bf  90 90 90 90 90 90 90 90  |................|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000030  90 90 90 90 31 c0 89 c2  50 68 6e 2f 73 68 68 2f  |....1...Phn/shh/|
00000040  2f 62 69 89 e3 89 c1 b0  0b 52 51 53 89 e1 cd 80  |/bi......RQS....|
00000050  8f fa ff bf 01 01 01 02  01 01 01 01 40 01 80 04  |............@...|
00000060  08 04 04 40                                       |...@|
00000064
```

A.5 Uninitialized Variable Vulnerability

Listing 19: A sample code for uninitialized variables vulnerability

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

void b()
{
  void (* ptr)(void );

  if(ptr != NULL)
    ptr();
}

void a(char *argv)
{
  char text[100];

  strncpy(text,argv, 100);
}

int main(int argc, char **argv)
{
  if(argc > 1)
    a(argv[1]);

  b();

  return 0;
}
```

Listing 20: An exploit for Listing 19

```
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000040  90 90 90 90 31 c0 89 c2  50 68 6e 2f 73 68 68 2f  |....1...Phn/shh/|
00000050  2f 62 69 89 e3 89 c1 b0  0b 52 51 53 89 e1 cd 80  |/bi......RQS....|
00000060  b1 fa ff bf                                       |....|
00000064
```

A.6 Format String Vulnerability

Listing 21: A sample code for format string vulnerability

```c
#include <stdio.h>

void a(char * argv)
{
  char fmt[100];

  snprintf(fmt, sizeof(fmt), argv);
  printf("%s", fmt);
}

int main(int argc, char **argv)
{
  if(argc > 1)
    a(argv[1]);

  return 0;
}
```

Listing 22: An exploit for Listing 21

```
00000000  14 96 04 08 15 96 04 08  16 96 04 08 17 96 04 08  |................|
00000010  25 34 34 36 78 25 37 24  6e 25 33 30 30 78 25 38  |%446x%7$n%300x%8|
00000020  24 6e 25 32 36 31 78 25  39 24 6e 25 31 39 32 78  |$n%261x%9$n%192x|
00000030  25 31 30 24 6e 90 90 90  90 90 90 90 90 90 90 90  |%10$n...........|
00000040  90 90 90 90 90 90 90 90  31 c0 89 c2 50 68 6e 2f  |........1...Phn/|
00000050  73 68 68 2f 2f 62 69 89  e3 89 c1 b0 0b 52 51 53  |shh//bi......RQS|
00000060  89 e1 cd 80                                       |....|
00000064
```