International Journal of Cultural Inheritance & Social Sciences (IJCISS)
Vol. 5 Issue 10, September 2023, ISSN: 2632-7597 Impact Factor:5.266
Journal Homepage: http://ijciss.com/, Email: journal.ijciss@gmail.com
Double-Blind, Peer Reviewed, Refereed& Open Access International Journal

AUTOMATIC GENERATION OF CONTROL, FLOW HIJACKING EXPLOITS FOR SOFTWARE VULNERABILITIES

Esam Mohamed Elwan
Professor of Strategic management and computer management
Head of Information Technology and Human Development Consultant
Al ‘Ain University of Science and Technology, U. A. E.
Email: dresamelwan@gmail.com

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.

Keywords: Automatic Generation, Control, Software Vulnerabilities, Software bugs, memory corruption, security vulnerabilities, data-flow analysis, generation, techniques.

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 code1. 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 appli- cation 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).

The challenge of AEG is to construct a program input that results in the execution of shell code. 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 shell code. 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 shell code. 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 categoryised 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 shell code.

Significant challenges are encountered when one attempts to take this step from crashing the program to execution of shell code.

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 deter- mine 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 limits2:

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 and omisation may be enabled on the system but no other exploit prevention mechanisms are in place.

3 – Shell code 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 By akugan 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

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.

3 - We present the results of applying the implementation of the above algorithms to a number of vul- nerabilities. 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.

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.

1 – 6 Overview

consists of a description of how the exploit types we will consider function, followed by a formal- isation of the components required to build such exploits. contains the main description of our algorithm and the theory it is built on.

we outline the implementation details related to the algorithms described in contains the results of running our system on both test and real-world vulnerabilities. Finally, discusses suggestions for further work and our conclusions.

ProblemDefinition

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 Linux1 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

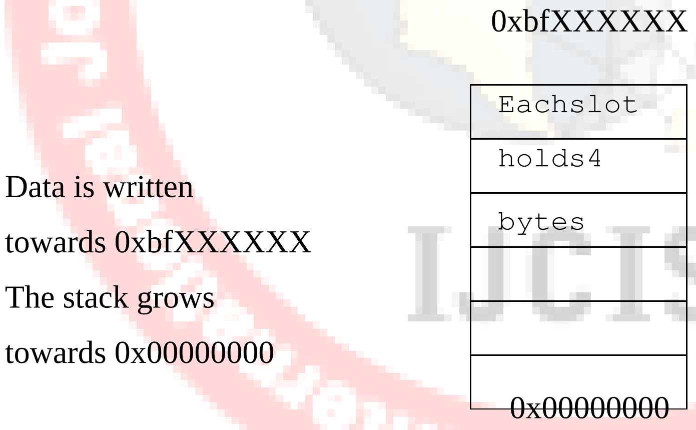


Figure 2.1: Stack convention diagram

required from the ESP. The push and pop instructions insert/remove their operand from the stack and decrement/increment the value in ESP2. 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.

At the end of every function3 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.

Example2.1x86functionepilogue

```
movesp,ebp;MovetheaddressofthestoredEBPintotheESPregisterpopebp;Popthevalueofthest
oredEBPintotheEBPregister
ret;PopthevalueofthestoredEIPintotheEIPregister
```

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 functions4. 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].

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 n∈mber of paths Ω. Each path ω Ω is a potentially infinite sequence of

pairs [i0, i1,..., in,...], where each pair i contains an address iaddr and the associated assembly level instruction at that address iins. Each pair can appear one or more times.
The nth instruction of a path ω is denoted by ω(n). Two paths ωj and ωk are said to be different if there exists a pair ω(n) such that ωj(n) /= ωk(n).

Definition 2 (Definition of an Instruction). When referring to the instruction iins in a pair i we will use the identifier of the pair itself i, and specify the address explicitly as iaddr 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 isrcs, while the set of destination operands is referenced by idsts. The sets idsts and isrcs 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 ωc is a path where the value of each operand of an ∀instruction i is known i ωc. 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) = ωc, where I is an input to the program P resulting in the path ωc 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 P to denote a running instance of P.

Definition 4 (Definition of Program Memory). Assuming A is the set of all valid memory P addresses in and B is the set of all byte values 0x00,..., 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 ainsdsts.

We divide the domain of M, the memory addresses, into two subsets. Those memory∈ addresses a A that can legitimately be used to store data tainted5 by user input constitute the set Mu6. The complement of Mu in M is the set of addresses a A that should not used to store data tainted by user input. We call this set Mm7. A subset of Mm is the set MEIP , 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.

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.

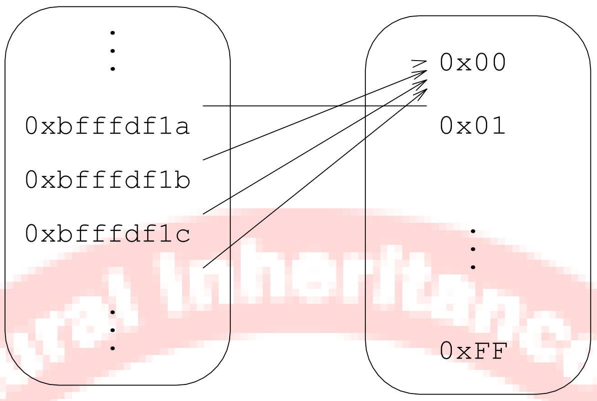


Figure2.2:Updating*M*:movDWORDPTR[eax],0x00000000

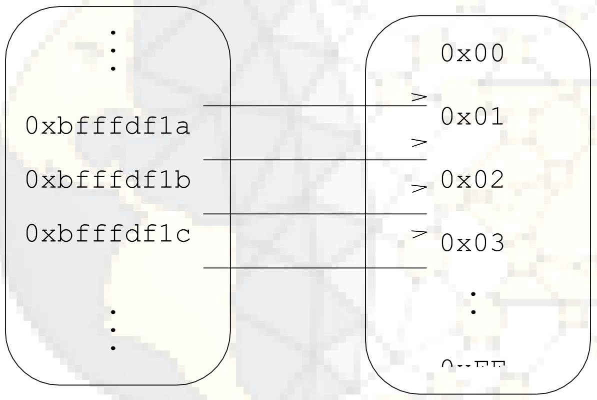


Figure2.3:Updating*M*:movDWORDPTR[eax],0x01020304

CPU under consideration.Its members are register identifiers, e.g. EAX, BX, ESP and so on. The rangeofRisthesetofdwordvalues{0x0,...,0xffffffff}.

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 Mm, 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 [i0,...., in], where one or more instructions result in a write to an address in Mm, and where the instruction in 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 ωc 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 , and instead execute malicious code.

If such an input is possible then the bug is a security vulnerability and we would describe X the input 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 X to be an input to P such that the path

P (X) contains the following three components (B, H, S):

1 - B is a sequence of instructions that directly corrupt one or more bytes in Mm, 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 meip MEIP 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 ≡ B, such as when a buffer overflow directly corrupts a stored instruction pointer. In other cases H ≢ 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 MEIP when this write instruction takes place.

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

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

Listing2.1:“Stack-basedoverflowvulnerability”

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

voidfunc(char*userInput)
 {
  chararr[32];

  strcpy(arr,userInput);
 }

intmain(intargc,char*argv[])
 {
  intres,fd=-1;
  char*heapArr=NULL;
  fd=open(argv[1],O_RDONLY);

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

  return0;
 }
```

In all exploits we consider the goal is to corrupt an address in MEIP with the aim of redirecting execution to attacker specified shellcode. We categorise the type of exploit where H B as a direct exploit, and as an indirect exploit when H B. That is, they differ in how the address in MEIP is corrupted. A direct exploit modifies an address in MEIP as part of the initial memory corruption whereas an indirect exploit modifies an address in Mm, but not in MEIP , that later causes a modification to a value in MEIP. 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 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:

[CCCC*8]+[BBBB]+[AAAA]

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

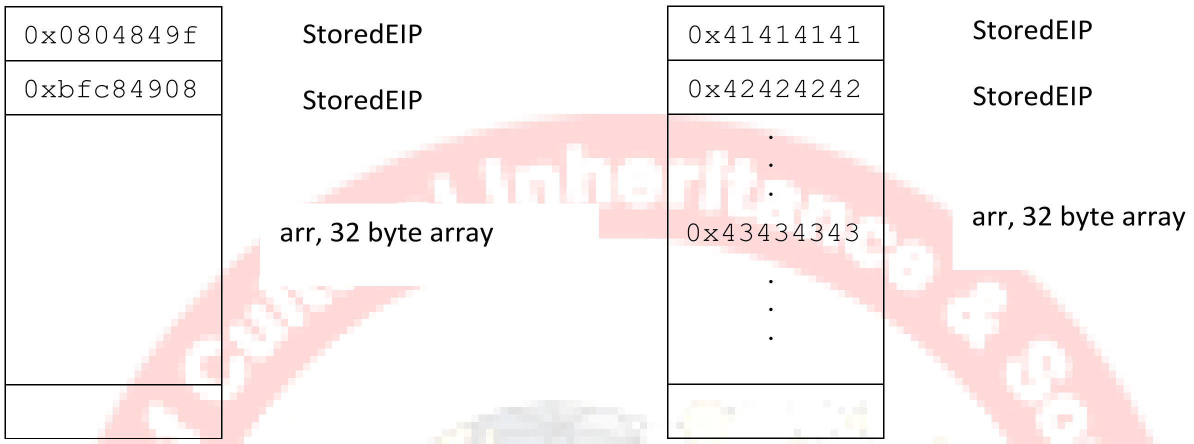


Figure2.4:Stackconfigurationbefore/afterthevulnerablestrcpy

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 X to be considered an exploit for P we must ensure the path P ( ) 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 MEIP , 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 X be the following string:

[CCCC*8] + [BBBB] + [AAAA]

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 shell code.

We will denote the address of the pointer value we aim to corrupt as mEIP , where mEIP ∈ MEIP. 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 (X) |= (B, H). For(P) = (B, H, S) to hold we must change our input to include shell code, 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 shell code and its address used to overwrite mEIP 10. It is possible to hardcode this address into the exploit as there is no randomisation in the address space. Once the value at mEIP is moved to the EIP register by the ret instruction the code in arr will be executed. Our input string could appear as follows:

[32bytesofshellcode]+[4bytes[11](storedEBP)]+[Addressofarr]

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 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 mEIP with the address of b we instead overwrite it with the address of a register trampoline that uses r12.

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 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:

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

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

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.

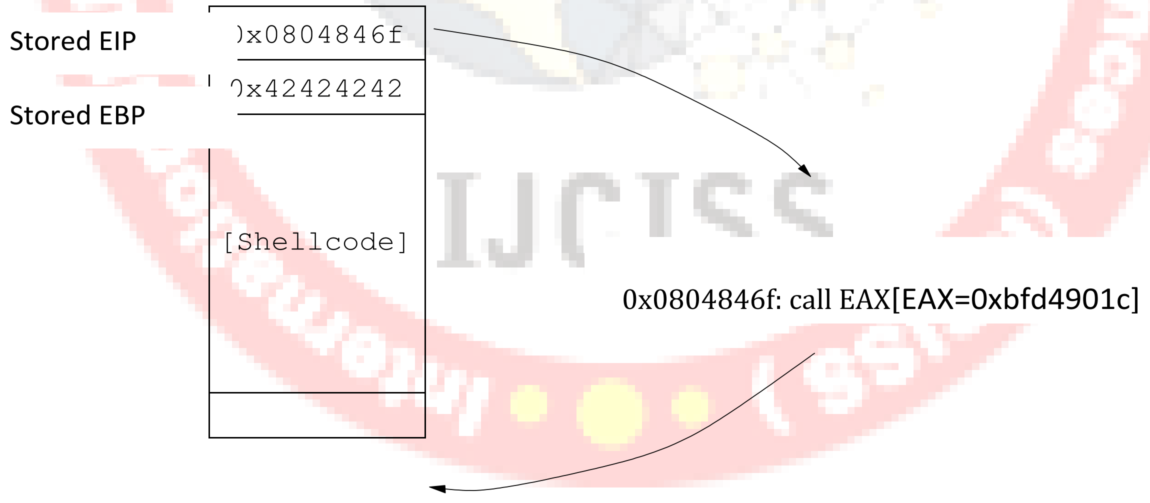


Figure2.5:Returningtoshell codeviaaregistertrampoline

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 shell code 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 mEIP is implicit and will be the address of the overwritten stored instruction pointer or function pointer. In order to generate a direct exploit Xd we require the tuple (P, ι, C, Λ, Γ), where

1 - P is a running process of the program-under-test P.

2 - ι is a valid address in the address space of 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 Xd 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 (Λ).

5 - Γ is our exploit generation function. It takes Λ, ι and C, and analyses P (Λ). The goal of Γ is to produce a formula where a satisfying solution to its constraints is an input Xd such that P (Xd) |= (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 (X) for X to be considered an exploit. An indirect exploit Xi is an input to P such that in the path P (Xi) 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 mEIP ∈ MEIP.

Listing2.2:“Stack-basedoverflow vulnerability (write offsetcorruption)”

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

voidfunc(int*userInput)
 {
  int*ptr;
  intarr[32];
  inti;

  ptr=&arr[31];

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

  *ptr=arr[0];
 }

intmain(intargc,char*argv[])
 {
  intres,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);

  return0;
 }
```

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 mEIP MEIP. 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 wv where wv 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 mEIP 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.

WriteDestinationCorruption

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 such that P ( ) = (B, H, S).

X X|

In Listing 2.2 the loop bound on line 14 contains an off-by-one vulnerability. The loop will write one extra int14 from the user input beyond the bounds of the array arr. This corrupts the variable ptr, which

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 with a value of our choosing.

P

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 mEIP. By replacing the 33rd int of the input with this mEIP 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 heap Arr + 33*size of(int), as parts of the first 33 bytes are used in the corruption of mEIP. An exploit would look similar to

[heapArr+33*sizeof(int)]+[(31*sizeof(int))byte sofjunk]+[m_{EIP}]+ [Shellcode]

This will hijack the control flow as follows:

1 - On the 33rd iteration of the loop at line 14, the code arr[32] = userInput[32] will be executed, corrupting ptr with the value mEIP.

2 - 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 shell code. This code therefore replaces the stored instruction pointer at mEIP with the address of our shell code.

3 - When the ret instruction executes at the end of the function control flow will be redirected to our shell code 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 mEIP , 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:

Example2.3 Usingobjdumptofindthe. dtorssegme

```
% objdump -s -j.dtors programprogram:fileformatelf32-i386Contents of
section.dtors:804955cffffffff00000000
```

Astherearenodestructorfunctionsregisteredbytheprogram,the. dtors containsnofunctionad-dresses.Wecanstilluseitinourexploit,byreplacingthe0x00000000 withtheaddressofourshell code.The address of these null bytes is 0x0804955c + 4 = 0x08049560. Our previous exploit could now bemodifiedtothefollowing:

[heapArr+33*size of (int)]+[(31*size of (int))bytesofjunk]+[0x08049560]+[Shell code]

Inthisexamplewehaveassumedthattheheapisnotrandomisedandsotheshell codelocation(heapArr+33 *size of (int))canbehardcodedintotheexploit.Iftheheapwererandomisedaswellasthestackthentheregistertrampolinetechnique,describedintheprevioussection,couldonceagainbeused.Theabove bug is considered a *write-4-bytes-anywhere* vulnerability.The attacker can control all 4 bytes of the sourcevalue (*write-4-bytes*) and all 4 bytes of the destination (*anywhere*).It is a specialisation of a bug class knownas*write-n-bytes-anywhere*wheren≥0.Inthisworkwewilljustconsiderthe4-bytecase.

2.6.2 ComponentsRequiredtoGenerateanIndirectExploit

The components required for an indirect exploit are slightly different than those of a direct exploit.AsdemonstratedintheexploitforthecodeinListing2.2,anindirectexploitrequiresonetospecifyoneorboth operands to a write instruction. The function Γ will therefore be different to that required for a directexploit.Also,thevalueform$_{EIP}$mustnowbespecifiedasitisnolongerimplicitintheoverflow.Tobuild anindirectexploitX$_i$werequirethetuple(P,m$_{EIP}$,ι,C,Λ,Γ)where

Pisarunningprocessoftheprogram-under-testP.

m$_{EIP}$isamemoryaddressknowntobeinthesetm$_{EIP}$whenthevulnerablewriteinstructionexecutes.Itwillbethedestinationoperandoftheinstructionthevulnerablewritew$_v$.

is a valid address in the address space$_P$ ofthat will be used as the pivot destination. This could bethe address of a buffer that will contain our shell code at the pivot point or of a register trampoline.Theaimoftheexploitisforιtobethevaluewrittentom $_{EIP}$

Cisourshell code,avalidsequenceofassemblyinstructions.AswithadirectexploittheoverallgoalincreatingX$_i$istheexecutionoftheseinstructions.

Λisaprograminputthattriggersabug.OuranalysiswillbeperformedoverthepathP(Λ).

Γ is a function that takes Λ, m $_{EIP}$, ι and C.The goal of Γ is to produce a formula such that a satisfyingsolutiontoitsconstraintsisaninputX$_i$thatcanbedeemedanexploitforP.

TheProblemweAimtoSolve

WecanformulatetheAEGproblemasbuildingΓ.Γmustanalysethepathgivenby*P* (Λ)inordertogenerate a formula *F* expressing the conditions on an input for it to be an exploit. $\overset{X}{A}$ satisf$\overset{X|}{y}$ing solution to*F*willbeanexploitsuchthat*P*()=(*B*,*H*,*S*).AtitscoreΓmustperformthreetasks.

Firstly, it must be able to analyses the memory state of the program in order to find suitable locations tostore the injected shell code.It must be able to determine the bytes from user input that influence the valuesof such locations.It also must be able to discover all modifications performed on these input bytes by theprogram up to the pivot destination.From this information it must be able to generate an input that willresultintherequiredshell codefillingtheselectedbuffer.

Secondly,Γmustdetectthelocationsin$M_m$thataretaintedbyuserinputinordertoredirectcontrolflo wtotheshell codebuffer.Onceagainthiswillrequirethealgorithmtodeterminetheinputbytesthatinfluence these corrupted locations and the modifications imposed on them before the corruption takes place.Finally,Γmustbeabletocreateaaformulabasedonthepreviousanalysis.Asatisfyingsolutio nforthisformulashouldbeparseabletoaprograminputthatcorruptsthevaluesin$M_m$resultinginthe redirection ofcontrolflowtoinjectedshell code.

The AEG problem is therefore a combination of an old problem, gathering data-flow information fromrun-timeanalysis,andanewproblem,usingthisinformationtoautomaticallygenerateaninput.Thealg orithms that make up Γ must solve this latter problem and will rely on data-flow analysis algorithms toprovidetherequiredinformationontheprogramsexecution.

AlgorithmsforAutomaticExploitGeneration

In this Chapter we will explain the algorithms we have developed for automatic exploit generation.Thesealgorithms are designed to generate exploits that satisfy the definitions provided in Chapter 2. To create suchexploits we utilise methods similar to those described in previous work on exploit generation [11], vulnerabilitydiscovery[38,15,14,27],andotherprogramanalysisproblems[43,9].Thisapproach combinesdynamicdata-flowanalysisandenumerationofthepathconditionwithadecisionprocedure.Theinformationgat heredviadata-flowanalysisisexpressedasalogicalformulaandcanbeprocessedbythedecisionproceduretogiv e a satisfying solution, if one exists.This formula accurately represents the execution of program underanalysis.By modifying the formula one can use a decision procedure to reason about possible paths andvariablevalues.

Weextendthepreviousworkbyderivingsuitablelogicalconditionstoexpresstheconstraintsrequiredby an exploit. By appending these conditions to a formula describing the programs execution we can use adecision procedure to determine if an exploit is possible. In cases where an exploit is possible we can parsethesatisfyingassignmentfortheformulatoafunctionalexploit.Anexploitgeneratedbyourapproachwill result in shellcode execution in the target process where previous work would have simply caused theprogramtocrash.

Our approach can be divided into three high-level activities as illustrated in figure 3.1. Our contributionsare mainly encapsulated in *Stage 2*, although as part of *Stage 1* we extend the commonly used taint analysistheory.A new taint classification model is introduced that we believe to be advantageous when data-flowanalysisiscombinedwithadecisionprocedure.

*Stage1*consists of iterative instrumentation and analysis,performed on the path generated by *P* (Λ),whereΛisaninputthatcausestheprogram*P*tocrash.Weanalysethispathbytracingtheprogramasit executes, performing data-flow analysis and recording information relevant to the path condition.Thisprocess continues until a potentially exploitable vulnerability is discovered.Section 3.2 describes how wedetectsuchvulnerabilities.

Onceapotentialvulnerabilityisdiscoveredwebegin*Stage2*.ThisstageconsistsoffourtasksandembodiesthefunctionΓ:

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

ii - We then build the first component of our exploit formula,which is a formula constraining a suitablebufferinmemorytothevalueofourshellcode.Beforewecanbuildthisformulawemustfirstanalysetheinformationgathereddduringtaintanalysisandfilteroutsuchabuffer.Thelocationswedecidetouseastheshellcodebufferwilldeterminethetrampolineaddressιusedinthenextformula.

iii - The second formula we construct constrains a stored instruction pointer or function pointer (directexploit), or the operands to a write instruction (indirect exploit).In the case of a direct exploit weconstrainthevaluethatwillbeputinEIPtoι,whereasforanindirectexploitweconstraintthe sourceoperandtoιandthedestinationoperandtom_{EIP},alocationasdescribedinChapter2.

iiv - We then combine the above two formulae and calculate the path condition for all memory locations ineach.Thisfinalformulaexpressestherequiredconditionsonanexploitfor*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.

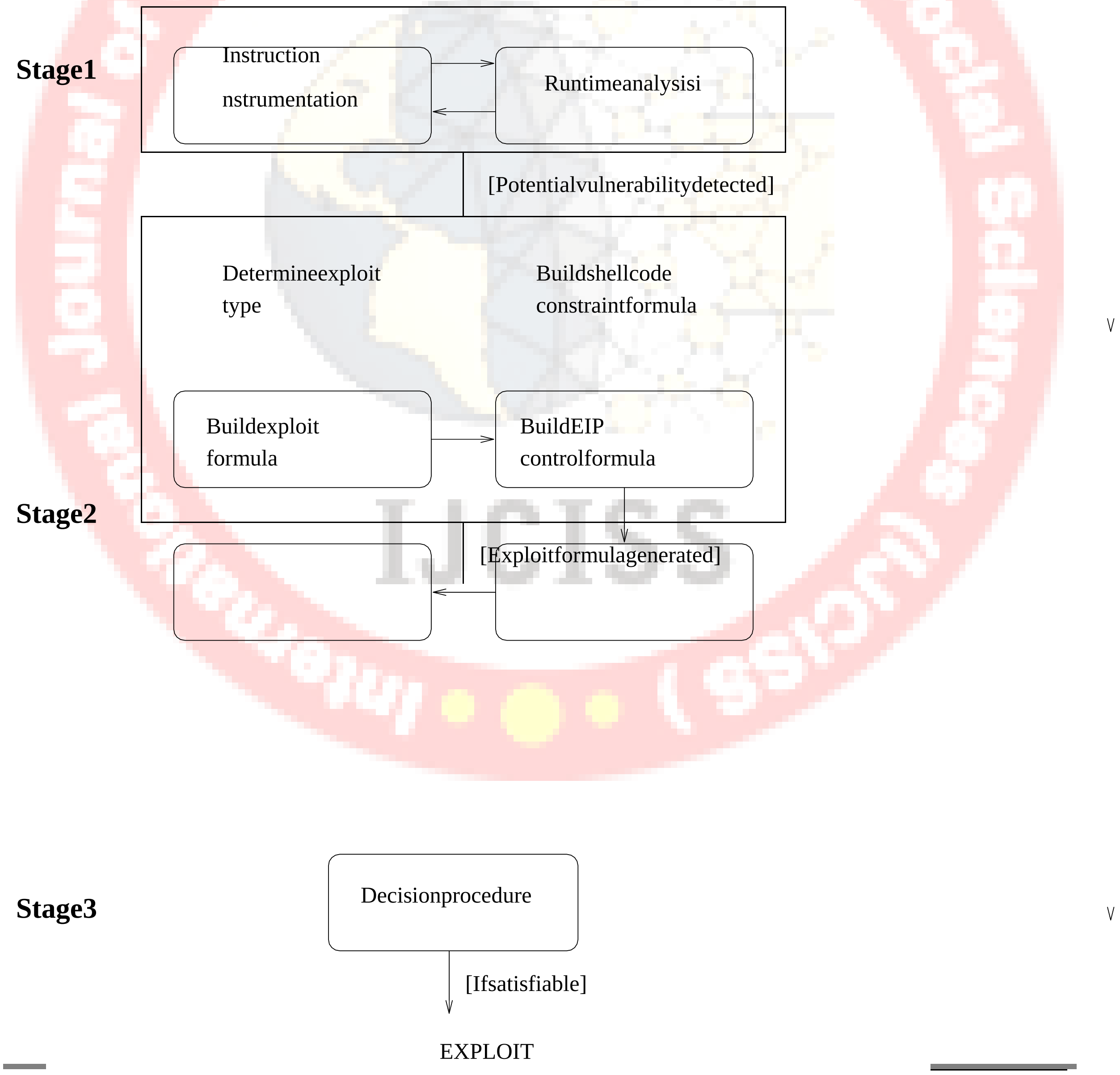


Figure3.1:Highlevelalgorithmforautomaticexploitgeneration

Stage1:Instrumentationandrun-timeAnalysis

The purpose of this stage of our algorithm is to gather sufficient information on data-flow and path conditionsfrom the execution of P (Λ) to allow the later stages of the algorithm to generate an exploit.We gather thisinformationastheapplicationisrunningusingadynamicbinaryinstrumentationframework.

DynamicBinaryInstrumentation

DescriptionandTheory

To gather information on data-flow and path conditions we use dynamic analysis.This is run-time infor-mation gathering so no approximations have to be made for variable values. Common sources of inaccuracyencounteredduringstaticanalysisareavoided,e.g.unknownloopbounds[7]andpoint eraliasuncertainty
are not an issue for dynamic analysis.There are essentially three common approaches to gathering therequiredrun-timeinformationfromaprogram.

Executiontracelisting- Inthisapproachalistoftheinstructionsexecuted,registervaluesandmemory locations modified are logged to a database at run-time.The Nirvana/iDNA framework [6] isdesigned to facilitate this kind of analysis. Some debuggers also provided a limited form of the requiredfunctionality, e.g.Ollydbg[1].This approach is useful due to its lower impact on the run-time performanceof the program under test in comparison to some other methods. On the downside the generated tracescaneasilyrequiregigabytesofstoragespace.

Emulation -A number of different tools now exist that allow one to run a program within a virtualoperatingsystem,orruntheoperatingsystemitselfontopofemulatedhardware.Emulatorst hatprovide an API, such as QEMU [5], allow us to programmatically control the execution of the emulatedsystemandanalysetherequireddata-flowandpathconditioninformationatrun-time.

Binaryinstrumentation-Binary instrumentation is a technique whereby extra code is injected intothe normal execution flow of a binary. The injected code is responsible for observing the instrumentedprogramandcanusuallyperformarbitraryanalysisoftheexecutingprogram.Thism ethodofgath-ering run-time information is provided by a number of different frameworks, including Valgrind [41],DynamoRIO[8]andPin[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 ofdifferentarchitecturesarerepresentedinValgrind ,Dynamo RIO,andPinbutthehigh-levelconceptsarethesame.Eachframeworkprovidesavirtualenvironmentinwhichtheprogramundertest*P*isrun.TheDBI framework will typically provide a mechanism by which analysis code, called the client, can observe andmodify each instruction in the running program before it is executed by the CPU. This is called instructionlevel instrumentation.Some frameworks also allow instrumentation at the function level, whereby analysiscode is triggered on function calls, or on events such as the starting of a thread or the loading of a sharedlibrary.

OneoftheprimarydifferencesbetweenValgrindandPin/ Dynamo RIOisthatValgrindconvertstheassemblycodeoftheprogramintoanIntermediateRepresentation (IR)beforeitisgiventotheclientforanalysis.ThisIRisaRISC-likelanguageinwhicheachassemblyinstructionisconvertedtooneormoreIRinstructions,witheveryimplicitread/writeoperationintheassemblyinstructionbecominganexplicitlyIRinstruction. ThisisillustratedinExample3.1fromthefileVEX/pub/libvexir.hintheValgrindsource.TheuseofanIRmeansthatValgrindanalysisclientsdonothavetoexplicitlyaddsupportforallrequiredx86assemblyinstructionsastheyareconvertedtoamuchsmallersetofIRoperations.DuetothetransformationsrequiredtogeneratethisIR,Valgrindhasamorenoticeableeffectontherun-timeoftheprogramundertest[36]thanDynamo RIOandPin.

After considering the benefits of the IR versus its performance impact and the fact that Pin/Dynamo RIObothhavecross-platformandC++supportwedecidedagainstusingValgrind.WeinsteadchosePinasit provides a library of functions to analyse each x86 instruction,rather than using an IR, and thus incursmuchlessofaperformanceoverhead.IncomparisontoDynamo RIOithasbettersupportforC++andis available for more operating systems.The main disadvantage, over Valgrind, is that the analysis code ismoreverboseasthegeneralizationsintroducedbytheIRarelost.

Example 3.1 Valgrind Intermediate Representation

Forexample,considerthisx86instruction:addl%eax,%ebx

OneVexIRtranslationforthiscodewouldbethis:

```
------IMark(0x24F275,7)------
t3=GET:I32(0)                  #get%eax,a32-bitinteger
t2=GET:I32(12)                 #get%ebx,a32-bitinteger
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 programundertest.Pinthenloadstheanalysisclientintothesameaddressspaceandinterceptstheexecutionof

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 Pare noinstructions that alter the control flow, such as jumps or calls, except for the last instruction in the sequence.Pin takes these instructions from, injects any code specified by the analysis client and then recompilesthis into a new series of instructions using a just-in-time compiler. These instructions are then executed onthe CPU, with Pin regaining control of the execution when a branching instruction is hit. Pin uses a cachetostorepreviouslyanalysedandcompiledcodeandthusreducetheperformanceimpact.

ADynamicInstrumentationAlgorithm

Thepurposeofourdynamicinstrumentationalgorithmistoparseagiveninstructionandinsertcallbacksto analysis routines that are executed before the instruction is actually run on the CPU. Every instructionin a program may provide information relevant to the path condition and data-flow analysis. By examiningeachinstructionthealgorithmcandecidewhatanalysisroutinesmustbecalled,andwhatargumentsmustbe passed to these routines. This algorithm makes up the first part of *Stage 1* from diagram 3.1, labelled“Instructioninstrumentation”.

Algorithm3.1

Lines1-3: Thepurposeofthispartofthealgorithmistoparsetheregistersandmemorylocationsused in the instruction into a list of operands, a list of sources and a list of destinations.Pin providesfunctionstodeterminetheselocationsregistersandmemorylocationswritten.Letusassumethat$ins_{dsts}$is a vector of objects representing a destination operand and $ins_{dsts[x]}$accesses element x of thevector.Similarly,$ins_{srcs}$and$ins_{operands}$arevectorsofsourceoperandsandalloperandsrespectively.

Lines 4-11: While Pin provides functions to determine the locations read and written by an instruction,extraprocessingisrequiredinordertoaccuratelyrepresentthesemanticsofeachinstruction.

Asmentioned,weelectedtouseaDBIframeworkwithoutanIRandasaresulteachinstructionwe process may write to one or more destinations, using one or more of the instruction sources.Toaccurately analyse the data-flow we must take into account the semantics of individual instructions andextract the mapping from instruction sources to destinations. This means we have to relate one or moreelements of the vector of sources to each element of the vector of destination.We begin, on line 5, byusing the *extractSources* function to get the vector of source indices that effect the destination beingprocessed. This function is essentially a large map of destination indices to source indices that must beupdatedforeveryx86instructionwewishtoprocess.

Chapter3.1instrument Instruction (ins)

```
1: insoperands= extract Operands(ins)
2:  insdsts= extract Destinations(ins)
3: inssrcs= extractSources(ins)
4: for idx ∈ len($ins_{dsts}$) do
5:      srcIndices=extractSources(ins,idx)
6:      sources = vector()
7:      for idx ∈ srcIndices do
8:        sources.append($ins_{srcs[idx]}$)
9:      endfor
10:     insdsts[idx].sources= sources
11: endfor
12: if setsEFlags(ins) then
13:     eflags=eflagsWritten(ins)
```

```
14:   - PINInsertCall(AFTER,updateEflagsOperands(eflags,ins))
15:endif
16:ifisConditionalBranch(ins)then
17:     cond=getCondition(ins)
18:    operands=get Condition Operands( eflags Read(condition))
19:   - PINInsert Call(BRANCHTAKEN,add Conditional Constraints(cond,operands))
20:   - PINInsert Call(AFTER,add Conditional Constraints(!cond,operands))
21:elseifwrites Memory(ins)orwrites Register(ins)then
22:    ifwrites Memory(ins)then
23:     - PINInsert Call(BEFORE,ins,check Write Integrity(ins))
24:    endif
25:   - PINInsert Call(BEFORE,ins,taint Analysis(ins))
26:   - PINInsert Call(BEFORE,ins,convert To Formula(ins))
27:endif
28:instruction Type=get Instruction Type(ins)
29:ifinstruction Type==retthen
30:   - PINInsert Call(BEFORE,ins,check RETI ntegrity(ins))
31:elseifinstruction Type==callthen
32:   - PINInsert Call(BEFORE,ins,check CALLI ntegroty(ins))
33:endif
```

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 line10thisvectorisstoredinthe*sources*attributeofthecurrentdestination.

Lines12-15: ForeachinstructionwemustdeterminewhetheritmodifiestheEFLAGSregisterornot.The values of the flags in the EFLAGS register are used to determine the outcome of conditionalinstructions.Tocorrectlyidentifytheoperandsthataconditionalinstructiondepends onwemustthereforedeterminetheoperandsinvolvedinsettingtherelevantEFLAGS. IfthecurrentinstructiondoeswritetotheEFLAGSregisterweextractthoseindicesthataremodified.Online14weuseafunctionprovidedbyPin[2]toinsertacalltotheupdate Eflags Operandsfunctionafter[3]the current instruction executes on the CPU. For each index in the EFLAGS register we store avector containing the operands used in the last instruction to set that index.This vector can containmemorylocationsandregisters.

Lines 16-20: In this part of the algorithm we process conditional branching instructions. A conditionalinstruction is one for which the outcome depends on the value of one or more indices in the EFLAGSregister.In this algorithm we are only considering the effects of conditional instructions that directlyalter the control flow i.e.conditional branches.Some conditional branching instructions in the x86instructionsetarejl,jg,jbandsoon.

Line17usesthe*get*
*Condition*functiontoextracttheconditiontheinstructionexpresses.Forexample,if the instruction is jl[4] then the condition extracted will be less-than.On line 18 we retrieve theoperandsoftheinstructionthatlastsettheEFLAGSindicesonwhichthisinstructionisdependent.Forexample,thejl
instructioncheckswhetherthesignflagisnotequaltotheoverflowflag.Thelastinstruction to set these flags will have updated the list of operands associated with each by triggeringlines12-15ofouralgorithm.

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

Thealgorithmfor*add Conditional Constraints* willbegiveninthesectiononTaintAnalysis,asitcontainsfunctionalitythatpertainstothatsection.

Lines 21-27: If an instruction is not a conditional branch then we check whether it writes to a memorylocationoraregister.

Iftheinstructionwritestoamemorylocationwefirstinsertacallbacktocheck Write Integrity.This function will determine if the instructions arguments have potentially beentaintedbyuserinputinsuchawayastoallowforanexploittobegenerated.

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 taint Analysis and convert To Formula are given in the follow two sections, as well 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".

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.

3.1.1 TaintAnalysis

DescriptionandTheory

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.

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 I dsts and (y isrcs y T) is not the empty set. An element x T is removed from T during the execution of an instruction i if and only if x idsts and (y isrcs y 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 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:

$$\begin{tikzcd}
\top \arrow[d, dash] \\
\text{Untainted} \arrow[d, dash] \\
\text{Tainted} \arrow[d, dash] \\
\perp
\end{tikzcd}$$

Figure3.2:Asimplelatticedescribingtainte dness

In this representation, each memory location or register is associated with a ∈ point in the lattice L describing its tainte dness, where that point is given ∀∈ by L(x) for H some memory location or register x. Given a location x idsts, for some instruction i, we can calculate its point in the ∈ lattice based on the lattice points occupied by y, y isrcs. There are two functions typically defined to perform H this operation over points in a lattice. These functions are join ( ), which returns the supremum (greatest upper bound), and meet ( ) which returns the infimum (greatest lower bound); both functions take two operands. To calculate L(x) for x idsts 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) = (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 taint Analysis algorithm is triggered immediately before theinstructiontobeanalysedisexecutedontheCPU.Algorithm3.1isresponsibleforensuringt hisoccurs.Asmentioned, the purpose of taint Analysis is to update the taint lattice position of each memory location orregisterinins$_{dsts}$.

It is worth noting at this point that in the implementation of our approach we decided to perform our taintanalysis and other algorithms at the byte level instead of the 4-byte dword level, as done in the Catch Convproject [38]. While this results in larger constraint formulae, it also makes it easier for us to directly controlthevalueofmemoryandregistersatthebytelevelwhichiscrucialforexploitgeneration.

Chapter3.2taint Analysis(ins)

```
1: for ∈dst insdsts do
2:     latticePos = TOP
3:     fo∈r src dst.sources do
4:         lattice Pos = meet( lattice Pos, 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

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.

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.

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 $a \times b$ 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.

$$\begin{tikzcd}
\top \arrow[d, dash] \\
\text{Untainted} \arrow[d, dash] \\
\text{Tainted(Assignment)} \arrow[d, dash] \\
\text{Tainted(Linear Arithmetic)} \arrow[d, dash] \\
\text{Tainted(Non−linear Arithmetic)} \arrow[d, dash] \\
\bot
\end{tikzcd}$$

Figure 3.3: A taint lattice ordered on arithmetic complexity

We have sorted the x86 instruction set into three categories of instructions - assignment, linear arith- metic 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

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

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

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 taint Analysis(ins)

```
1:ins Complexity=get Ins Complexity(ins)
2: for dst   $ins_{dsts}$ do
3:      lattice Pos=TOP

4:      for src  dst.sources do
5:        latticePos=meet(latticePos,L(src))
6:      endfor
7:      if latticePos!=untainted then
8:        lattice Pos=meet( lattice Pos ,ins Complexity)
9:      endif
10:     L(dst)= lattice Pos
11:endfor
```

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 add Conditional Constraints 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 add Conditional Constraints updates the lattice positions of such variables to their correct value.

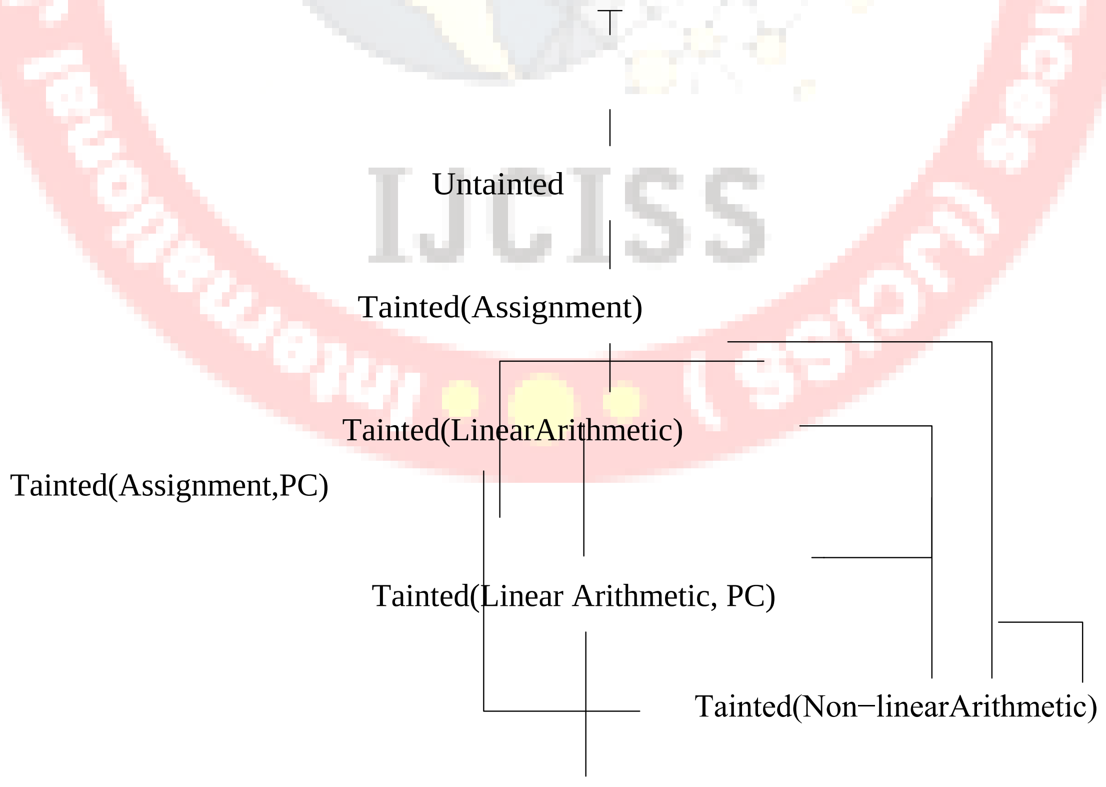
Chapter3.4add Conditional Constraints(cond,operands)

```
1:Conditions.add ( cond,operands)
2:for var      operands do
3:    L(var)=meet(L(var),PC)
4:endfor
```

Algorithm 3.4

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.

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.


⊤
Untainted
Tainted(Assignment)
Tainted(LinearArithmetic)
Tainted(Assignment,PC)
Tainted(Linear Arithmetic, PC)
Tainted(Non−linearArithmetic)

$$\begin{tikzcd}\text{Tainted(Non−linearArithmetic,PC)} \arrow[d, no head] \\ \bot\end{tikzcd}$$

Figure3.4:Ataintlatticeorderedonarithmeticandconditionalcomplexity

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 **P** ω in 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 ω(x) and then using a decision procedure to discover the variable assignments required at an earlier instruction ω(y) such that at ω(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 ω(y) and ω(x). For Listing 3.2 the resulting path condition looks as follows:

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

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:

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

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 i∈dx      len(insdsts) do
2: varId = generateUniqueId()
3: updateVarId(insdsts[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

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.

Line 4: The function make Form ulaFrom Rhs is used to generate a formula representing the right hand side of the assignment to the current destination. Similar to the extract Sources 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 insdsts[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.

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

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 build Path Condition (varId)

1: srcs Formula = build Sources Formula (src VarId)

2: path Condition = add Conditional Constraints (srcs Formula)

3: return path Condition

Chapter 3.7 build Sources Formula (varId)

```
1: if varId user Input Vars then
2:     return Empty Formula
3: end if
4: symbolic Formula = get Assignment(varId)
5: sources = extract Source Ids (symbolic Formula)
6: for src VarId sources do          ∈
7:     src Formula = build Sources Formula (src VarId)
8:     if len (src Formula) != 0 then
9:         symbolic Formula = create Conjunct(symbolic Formula, src Formula)
10:    end if
11: end for
12: return symbolic Formula
```

Chapter 3.8 add Conditional Constraints(symbolic Formula)

```
1: for var∈Id      symbolic Formula. variables do
2:     constraints = get Constraints On VarId (varId)
3:     for constrai∈t      constraints do
4:         symbolic Formula = create Conjunct(symbolic Formula, constraint)
5:     end for
6: end for
7: return symbolic Formula
```

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

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.

Lines 4-5: On line 4 we retrieve the symbolic formula in which var Id 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.

Lines 5-11: For each source ID we then recursively call the algorithm build Sources Formula 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

Lines 1-2: This algorithm iterates over all the unique variable IDs in symbolic Formula.On line 2 it uses the function get Constraints On Var Id to retrieve any constraints that were stored relating to var Id by algorithm 3.1.

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.

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.

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.

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.

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.

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 ivp. The purpose of this stage is to build a formula F such that a satisfying solution to F is an input X to P such
that P (X) |= (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.

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 X to P such that P (X) = (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 shell code.

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 satis fiability 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 shell code.

If both requirements are satis fiable we can then generate a formula to constrain the required values of the stored instruction pointer/write operands and the shell code 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:

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.

Shell code: The code we wish to inject into the process and run.

Trampoline Addrs: A map of registers to addresses of valid trampolines.

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.

Indirect Contro lIns: 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.registers: A map of registers to their value at crashIns.

Algorithm 3.9

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 indirect Contro lIns.

Our analysis algorithm then continues execution on line 7 at this address.

Chapter 3.9 Γ(crashIns, shell code, tramp oline Addrs, indirect m EIP, indirect ControlIns, registers)

```
1: if crashIns.type == WRITE then
2:exploitType = indirect
3:continueExecutionUntil(indirect ControlIns)
4: else
5:exploitType = direct
6: end if
7: shell code Buffers = build Shell code Buffers(registers) Alg. 3.10
8: sc Buf = None
```

```
9: for buf ∈ shell code Buffers do
10: if buf.size ≥ len (shell code) then
11: sc Buf = buf
12:break
13:end if
14: end for
15: if sc Buf == None then
16:exit("No shell code buffer large enough for shell code")
17: end if
18: sc Constraint Formula = build Formula Shell code({shell code, sc Buf) Alg. 3.11
19: ι = get Trampoline For Register(sc Buf. Jmp Register, tramp oline Addrs)

20: if exploit Type == direct then
21:eip Control Formula = build Formula Ins Pointer Control{(registers}.ESP, ι) Alg. 3.12
22: else
23:dst = crash Insdsts[0]
24:src = dst. sources[0]
25:eip Control Formula = build Formula Write Control(dst, src, {indirect }nEIP, ι)  Alg. 3.13
26: end if
27:  eip And Sc Constraints = create Conjunct (eip Control Formula, sc Constraint Formula)
28: exploit Formula = eip And Sc Constraints
29: for varId ∈ eip And Sc Constraints. variables do
30:pc = build Path Condition{(varId) }Alg. 3.6
31:exploit Formula = create Conjunct (exploit Formula, pc)
32: end for
33: return exploit Formula
```

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.

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

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].

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

Lines 27-32: The conjunction of the shell code 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.

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.

Processing Taint Analysis Information to Find Suitable Shell code Buffers

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

A potential shell code 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 rando mised we can speed up this process. By observing that we only need consider shell code 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 shell code 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

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

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).

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 build Shell code Buffers(registers)

```
1: buffer Set = Sorted Buffer Set(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 = Shell code Buffer()
5: buffer.jmp Register = reg
6: buffer. start = reg.value
7: buffer .lattice Val = L (reg.value)
8: counter = 0
9: while L( reg. value + counter) != untainted do
10: buffer. lattice Val = meet( buffer. lattice Val, L( reg.value + counter))
11: counter +=1
12: end while
13: buffer.size = counter
14: buffer Set.insert(buffer)
15: end if
16: end for
17: return buffer Set
```

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 shell code buffer. Selecting the first buffer that is large enough to store our desired shell code S will also be the buffer with the least complex path condition for that size.

Building the Shell code 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 shell code S.

Chapter 3.11 build Formula Shell code(shell code, shellcodeBuffer)

```
1: formula = Empty Formula()
2: offset = 0
3: while offset < shell code.size do
4: byte Id = get Var Id (shell code Buffer.start + offset)
5: byte Formula = create Equality Formula(byteId, shellcode[offset])
6: formula = create Conjunct(formula, byteFormula)
7: offset += 1
8: end while
9: return formula
```

Algorithm 3.11

Line 3: This algorithm proceeds by iterating over every memory location in the shell code buffer con- straining its value to the value required by the shell code.

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

Line 5: Using the create Equality Formula function we generate the constraint to assign the correct shell code value to the current buffer location

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 shell code be the string ABCD. If the shell code buffer spanned the address range 1000 - 1010 then algorithm 3.11 would generate the following formula:

$$(1000).id = 0x41 \land (1001).id = 0x42 \land (1003).id = 0x43 \land (1004).id = 0x44$$

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 mEIP. In order to determine if the vulnerability is exploitable we need to find out if we can control the value at mEIP. 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 mEIP is tainted. If it is we can generate a formula expressing the constraint that the value at mEIP equals ι, where ι is the address of the shell code buffer.

The following algorithm illustrates the process:

Chapter 3.12 build Formula Ins Pointer Control(mEIP, ι)

```
1: formula = Empty Formula()
2: offset = 0
3: while offset < 4 do
4: if L(mEIP+ offset) == untainted then
5: return Empty Formula()
6: end if
7: byteId = get VarId (mEIP+ offset)
8: byte Formula = create Equality Formula(byteId, ι[offset])
9: formula = create Conjunct(formula, byte Formula)
10: offset += 1
11: end while
12: return formula
```

Algorithm 3.12

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.

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.

Lines 8-9: We use the createEqualityFormula function to generate a formula that expresses the constraint byteId = ι[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.

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

$$(x + 0).id = ι[0] \land (x + 1).id = ι[1] \land (x + 2).id = ι[2] \land (x + 3).id = ι[3]$$

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 address8, the location (usually a register) w src containing the value to be written, and mEIP and ι as described in Chapter 2.

Chapter 3.13 buildFormula WriteControl(w ID, w src, mEIP, ι)

```
1: formula = EmptyFormula()
2: offset = 0
3: while offset < 4 do
4: if L(w src + offset) == untainted then
5: return Empty Formula()
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, mEIP )
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 $w_{ID} = mEIP$, created on line 12. The first formula constrains the source of the write instruction to ι, while the second constrains the destination address to mEIP.

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 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.

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 for P. By providing 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.

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 Lb9 for manipulating these bit-vectors and its syntax is as follows:

Example 3.3 A syntax for bit-vector arithmetic

formula: formula∨formula|formula∧formula|¬formula|atomatom: termrelterm|Boolean-Identifier

rel: =|=|≤|≥|>|<

term:term op term | identifier | ∼ term | constant | atom ?term:term

op: ⊕|g|⊗|ø|>>|<<|&|||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 Lb. 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.

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 Lb, 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:

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

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:

Example3.4SMT-LIBformattedQF-BVformula

```
 (benchmarktest
:statusunknown
:logicQF_BV
:extrafuns((aBitVec[8])(bBitVec[8])(cBitVec[8])(dBitVec[8])(eBitVec[8]))
:assumption(=ab)
:assumption(=cbv10[8])
:assumption(=d(bvaddac))
:assumption(=ed)
:formula(=ebv20[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.

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.

Example3.5 SolvingaQF-BVformulawithYices

```
%./yices-e-
smt<new.smtsat
(=b0b00001010)
```

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 and P should be a path that satisfies (B, H, S) as described in Chapter 2.

References

[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 Drini´c, Darek Miho˘cka, 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.

[22] Commun. ACM, 5(7):394–397, 1962.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.