Exploitability Assessment with TEASER

A Thesis Presented

by

Frederick Ulrich

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University

Boston, Massachusetts

May 2017

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Exploitability Assessment with TEASER.

Author: Frederick Ulrich.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: William Robertson	Date
Thesis Reader: Timothy Leek	Date
Thesis Reader: Brendan Dolan-Gavitt	Date
Department Chair: Ms. Danielle DiFazio	Date

Director of the Graduate School:

Dean: Ms. Danielle DiFazio	Date

To my family.

Contents

List of Figures

List of Tables

List of Acronyms

AEG Automated Exploit Generation: Technique for automating the construction of Proofs of Concepts, or exploits.

ASan Address Sanitizer: Fine grained memory instrumentation tool in order to detect various classes of memory corruption errors.

ASLR Address Space Layout Randomization: Exploit mitigation that randomizes location of executable images in memory.

CVE Common Vulnerabilities and Exposures: An identifier for a known vulnerability in a publicly released piece of software.

CVSS Common Vulnerability Scoring System: Metric for assessing software vulnerabilities. Broken up into three parts: Access Vector, Access Authentication, and Access Complexity.

DARPA Defense Advanced Research Projects Agency: United States government agency that invests in scientific projects related to national security.

DEP Data Execution Prevention: Exploit mitigation that marks areas of memory as executable xor writable, or that executable images cannot have write permissions over the course of the program's execution.

DWARF Debugging With Attributed Record Formats: Debugging format for the ELF file format.

ELF Executable and Linkable Format: Executable file format for Linux binaries.

GOT Global Offset Table: Table of pointers from a dynamic symbol to its resolution in a shared library.

IOT Internet of Things: Embedded hardware devices that are becoming increasingly more present in the lives of everyday people.

PANDA Platform for Architecture Neutral Dynamic Analysis: Dynamic Analysis platform for system neutral plugins for analyzing code.

PIRATE Platform for Intermediate Representation based Analyses of Tainted Execution: System for architectural neutral taint analysis on top of LLVM and QEMU.

POC Proof of Concept: Demonstration of an exploit on a program.

RCE Remote Code Execution: A method of exploitation of a vulnerability in which the attacker can execute arbitrary code on a remote system.

RCTP Root Cause Taint Point: A description of $Where$ and $When$ the root cause of a bug happens and what data to label as tainted.

ROP Return Oriented Programming: Exploitation technique that uses code reuse to construct an exploit.

TCG Tiny Code Generator: Intermediary language that QEMU uses to emulate binary instructions.

TEASER Taint-based Exploitation ASsEssment from Root cause: Tainted-based exploitation measurement system created by the authors.

TCN Taint Compute Number: Value that represents the number of arithmetic computations on a tainted value.

UAF Use After Free: Class of vulnerability where data that was previously allocated is accessed after it is freed.

WOFF2 Web Open Font Format: Web font packaging format designed for compression and performance [20].

Acknowledgments

First and foremost, I would like to thank my family for their continual love and support. Their encouragement through this process is what kept me going in the hardest times.

I would also like to thank my advisor, Professor William Robertson, for his never-ending patience and wise direction.

I would also like to thank Cyber System Assessments Group at MIT Lincoln Laboratory for sponsoring this work

I would like to thank my dear friends Leo St. Amour, Ricky T, Andreas Kellas, and Paul Deardorff for taking time out of their busy schedules to proofread this thesis.

Last, I would like to thank the members of the open source software community, who created and maintain most of the software with which this thesis is built.

Abstract of the Thesis

Exploitability Assessment with TEASER[1]

by

Frederick Ulrich

Master of Science in Electrical and Computer Engineering

Northeastern University, May 2017

William Robertson, Advisor

Bugs are still plentiful in software. Furthermore, fixing bugs is difficult, so developing a way to rank bugs based on their severity is essential to save developer time. As a result, security researchers have realized the necessity of pairing their bug with a Proof of Concept (POC), or input to a program demonstrating the ability to use a bug to exploit the application, to demonstrate the relative severity of their bug compared with others. This process of modifying an input that causes a crash such that the input exploits a program is called *exploit development*. For the purpose of this thesis, we are only interested in POCs for memory corruption-based vulnerabilities.

Similar to fixing bugs, exploit development is a difficult problem. As such there has been some research on automating the creation of POCs. Most automated exploit generation techniques use a modified program verification approach, whereas others employ dynamic taint analysis for exploit detection. While these results have been widely disseminated and successful, there is still room for improvement. Both approaches rely on tracking attacker-controlled input which often leads to either computationally difficult constraint solving problems or taint explosion. Given the computational difficulty of exploit development, we advocate for a human-assisted approach. We envision a workflow where a tool and human analyst could inform each other.

In this thesis, we approach exploit development as an iterative process for an analyst with a semi-automated tool called *Taint-based Exploitation ASsEssment from Root cause (TEASER)*. Our tool takes a dynamic taint analysis-based approach to exploit development where the taint source is the data associated with the memory corruption as opposed to the program input. We

[1]This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

develop a model for analyzing the effects of a memory corruption bug on a program. The Root Cause Taint Model taints data that violates a program's memory model (i.e. use-after-free, buffer overflow) and detects usage of this tainted data at seven classes of locations called Exploit Sinks. We implement our model within *TEASER*, a system designed for aiding security researchers in the exploit development process. *TEASER* must be provided with a program and crashing input that triggers a memory corruption bug. We use *TEASER* on a suite of 6 programs from the Google Fuzz Test Corpora. Our results identify several locations in the programs that use tainted data from a memory corruption, demonstrating *TEASER*'s potential for precisely characterizing possible effects of - and thus the severity of - memory corruption vulnerabilities.

Chapter 1

Introduction

Software applications are inundated with bugs. While these mistakes in code may seem innocuous, they can often be misused in order to exploit a program (i.e., disclose sensitive information, execute arbitrary code, gain administrator privileges). An *exploit* is an input that triggers unauthorized behavior in an application, and bugs that enable this unauthorized behavior are *exploitable*. Despite the very real threat of exploitable bugs, many bugs can take an inordinate amount of time to fix, and distinguishing between exploitable and non-exploitable bugs from a crash dump or bug report is difficult.

Sometimes, bug reports will feature a Proof of Concept (POC), or input to a program demonstrating the ability to use a bug to exploit the application. For example, a recent write-up demonstrates a full *JavaScript to root* exploit leveraging a bug in the Chrome OS network manager [3] [5]. Such a case demonstrates the severity of the bug, but in other cases, a developer is not as certain [19] [13]. In these ambiguous situations, developers are forced to rate the security severity as *high* because of the potential abuse of a memory corruption. In these cases we are interested in exploitability measurements or a tool to aid in quickly creating a POC for a bug. This process of creating an exploit from a crashing input is called *exploit development*, and for the purpose of this thesis, we are only interested in developing POCs for memory corruption-based vulnerabilities.

1.1 A Motivating Example

We work out an example (see Figure 1.1) for a program that mismanages heap-allocated memory. Although we assume a GNU C library (glibc) version of `malloc` and `free`, some of the more general ideas of heap exploitation are still relevant to other allocators [22].

```c
#define USAGE "./motivating-example <input_program>\n"
void word_printer(long *data) {
    printf("%lx", *data);
}

// data type definitions
typedef enum {
    RAW_BYTES,
    PRINTER
} obj_type ;

typedef struct printer {
    long id;
    void (*read_disclosure)(long *);
    long *data_ptr;
    obj_type type;
} printer;

typedef struct chunk {
    char *pointer;
    size_t size;
} chunk;

// global data whose address is assigned to data_ptr in struct printer
long global_data = 1;
long printer_counter = 0;
chunk chunk_list[100];

int main_loop(FILE *fp) {
  char *line = NULL;
  size_t linecap = 0;
  ssize_t linelen;
  while ((linelen = getline(&line, &linecap, fp)) > 0) {
    // check that line is of size greater than 0
    switch (line[0]) {
        // create_printer_object -> id
        // allocate a printer object and return it's chunk id
        // adds object pointer and size to chunk_list
        case 'c':
            . . .
            break;
        // create_raw_byte_array len raw_bytes -> id
        // allocate an arbitrary-length string of raw bytes chunk id
        // adds object pointer and size to chunk_list
        case 'b':
            . . .
            break;
        // release id
        // free an object
        // VULN: leads to a UAF because we can still access
        // the chunk data with its id number
        case 'r':
            . . .
            break;
        // print id
        // print the data_ptr associated with the printer object
        case 'p':
            . . .
            break;
        // move dst_id src_id
        // copies the data from one malloc chunk to another
        // VULN: leads to a heap buffer overflow
        case 'm':
            . . .
            break;
        default:
            continue;
    }
  }
  return 0;
}

int main(int argc, char *argv[]) {
  char *fname = argv[1];
  FILE *fp = fopen(fname, "rb");

  return main_loop(fp);
}
```

Figure 1.1: Motivating example for a program with different types of memory corruption bugs.

The example above, Figure 1.1 (the full version is located in Appendix A), is an abridged version for readability of a simple interpreter designed by the authors to have many exploit opportunities via different memory corruption bugs. Two in particular are a use-after-free bug and a heap buffer overflow. Each bug requires different degrees of understanding data flow from the memory corruption to other program points. We were able to construct three proof of concept exploits for Figure 1.1: (1) a remote code execution exploit through a use-after-free, (2) a remote code execution exploit through a heap buffer overflow, and (3) an arbitrary read primitive. But in principle, more could be constructed based off of heap metadata attacks. Each bug that we use for the three exploits signals errors under common memory corruption detectors like Valgrind or Address Sanitizer (ASan) [37] [41].

The interpreter has five commands that are chosen based off of the first character in an input line. The first two commands allocate objects of different types. Once allocated, an object's pointer and its size are placed into a global array called `chunk_list`. The first allocation command is 'c' (`create_printer_object`) which allocates an object with two fields: (1) a function pointer to a function that prints the data pointed to by a `long *` and (2) the `long *` that we would like to print. Second, 'b' (`create_raw_byte_array`) allocates a string of raw bytes on the heap. The rest of the commands take a chunk id as input which is used as an index into an array. First, 'r' (`release`) frees an allocated object according to its id. Second, 'p' (`print`) prints an object according to its id. Last, 'm' (`move`) moves the data from one object to another without checking for a possible buffer overflow. Comments starting with "VULN" signal a potential oversight in the code.

One bug in this program is a dangling pointer in the `print` option on line 57 that can be triggered with the following input: `create_printer_object`, `release 0`, `print 0`. The use-after-free happens because pointers are still available to the user through the `chunk_list` container (defined on line 27) even after they have been released. Even though freed data is accessed, the program will continue without a crash. A tool like ASan will report the dangling pointer use and then abort. Instead, we seek a tool that alerts us of every point in a program trace that uses data related to the dangling pointer usage, not as detectors of exploitation, but to use as hints for exploit development.

1.2 Our Contribution

Figure 1.2 presents a typical timeline for the execution life of a buggy program. The timeline features three main points: (1) the introduction of input to the program (which can encompass many points in a program trace; for the purpose of this model, we will assume only one input point), (2) a memory corruption, also known as a *root cause*, and (3) the program exit or crash. Between (2) and (3) there are potential exploit sinks. We define an *exploit sink* (E_k) as an instruction that operates on data involved in the memory corruption without causing an immediate crash. Existing taint-based models for exploit detection may attempt to determine if certain exploit sinks are tainted by attacker-controlled input [38]. However, many of these points can be benignly influenced by attacker-controlled data, so such systems are *imprecise*. Additionally, an analyst may want to keep track of more fine-grained detectors, such as reads of data tainted by the memory corruption. But the false positive rate would likely be an obstacle if the taint source is the program input.

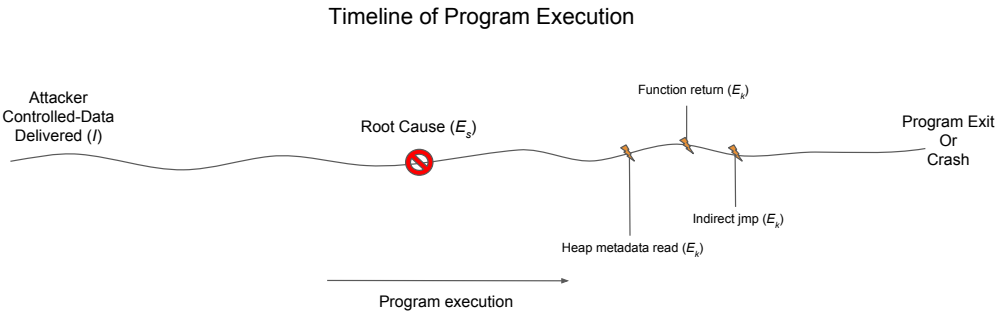

Figure 1.2: Timeline of a program featuring a root cause (a memory corruption) and potential exploit sink points (lightning bolts) after that.

Memory corruption-based exploits, in their most basic form, force certain instructions to operate on data that violates the programmer's intent or program's runtime model. We are particularly interested in what a program does with corrupted data and if such an analysis can inform the exploit development process. Notably, we revise traditional exploit detection systems by defining the taint source as the point of memory corruption rather than all attacker-controlled data.

We pose the question, "Given a program P and an input I that triggers a memory corruption bug, how can the program be exploited?" Our thesis is:

Given a memory corruption bug, tainting from the root cause of the bug provides a more precise exploitability assessment than that of existing systems.

We make contributions to the study of software security in the following ways:

1. Formulating a workflow for iteratively developing exploits.
2. Applying a taint based model for understanding how various classes of memory corruptions can affect a program.
3. Introducing more fine-grained detectors for exploitability measurement.
4. Developing a tool called *TEASER* that implements our model.
5. Employing *TEASER* on 6 memory corruption bugs from Google Fuzz Testing corpus in order to illuminate the severity of each bug [10].

Figure 1.3 summarizes the *TEASER* workflow. A human analyst is required to synthesize the knowledge from a debugger and open source memory corruption detectors. *TEASER* is an iterative process, and when the analyst is finished she will have a collection of exploit sinks that will serve as important targets during the exploit development process. After continued investigation into the exploit sinks, the analyst can make a more effective ruling on the severity of the bug, or she can modify the input I in order to achieve actual exploitation. The chief reason why the analyst can sift through these detectors to make a judgment is because root cause tainting reduces the number of false positives.

1.2.1 Road Map

Chapter Two will cover related solutions to speeding up or automating the POC development process and measuring exploitability. In Chapter Three, we will cover the theoretical foundations to understand our approach. These foundations will cover dynamic taint analysis, the subtlety of memory corruption exploits on modern systems, and how tools like Valgrind and ASan aid in diagnosing memory corruption bugs. Then, we will cover our taint-based approach to assessing memory corruption bugs in Chapter Four. The results of our tool will be presented in Chapter Five.

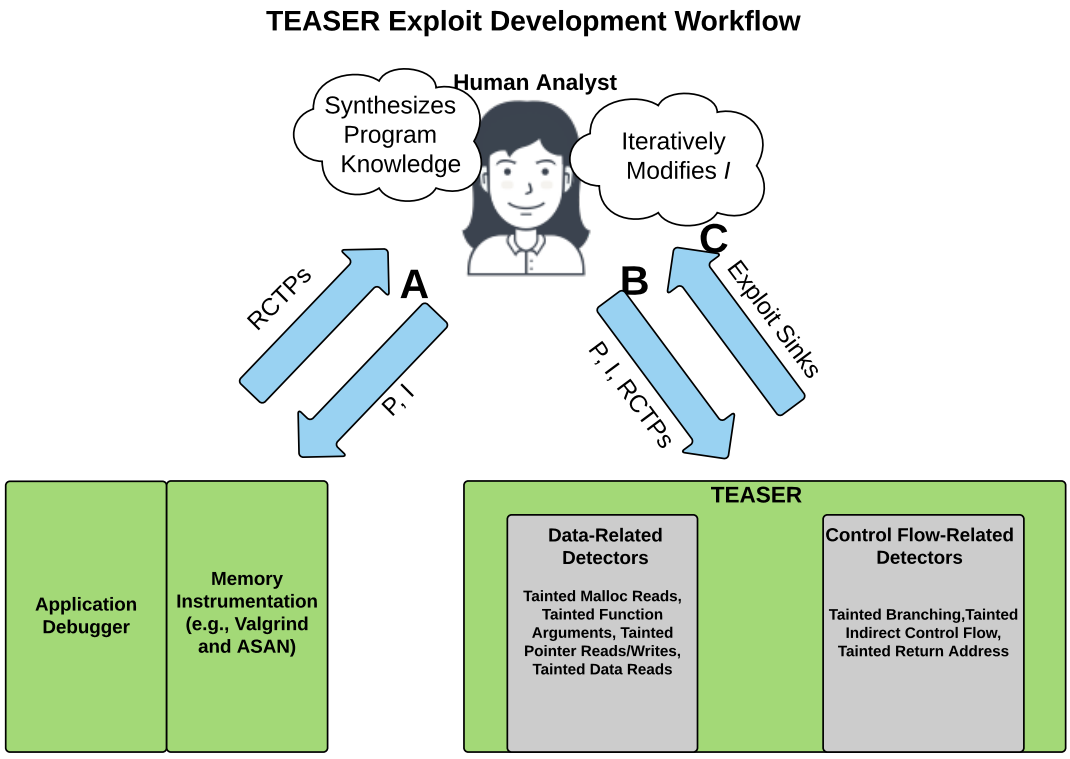

Figure 1.3: *TEASER* is a system meant to complement existing tooling for exploit development, crash triaging, and bug finding. *TEASER*, like exploit development, is an iterative process of aggregating potential exploit paths for a program P on input I. At *A*, the analyst uses a debugger and/or one of various memory instrumentation framework to diagnose the root cause of a memory corruption (e.g., use-after-free and buffer overflow). The goal of the analyst at *A* is to determine *where* and *when* the memory corruption is triggered and *what* data is involved, or the RCTP (Root Cause Taint Point). *B* introduces the *TEASER* system. Here, the analyst feeds *TEASER* the RCTP. Then, *TEASER* reruns P under a dynamic analysis framework called PANDA [30]. Using the information from *B*, PANDA taints the data from the memory corruption and registers detectors at certain program events like the reading of heap metadata and an indirect control flow instruction. In *C*, *TEASER* reports to the analyst what sensitive program areas were tainted by the memory corruption. From here, an analyst will either (1) repeat the process on either a new input $I_{improved}$ or (2) conclude that the program is, indeed, exploitable.

Chapter 2

Background and Related Work

We discuss background and related work to the field of exploitability measurement: root cause analysis, crash triaging, and automated exploit generation.

2.1 Root Cause Analysis

A root cause is the origin of a bug. Usually, it is the first instance of a memory corruption through either a buffer oveflow or accessing data in the heap through a stale pointer. Root cause analysis starts with a program P, a crashing input I, and a crashing instruction C. The analysis asks, *what is the origin of the corrupted data used in C?*

Miller et al. narrate a process for iteratively finding the root cause for a program crash using program analysis tools [35]. They start with a crashing instruction on some invalid operand, which is either a memory location, register, or constant. Their process requires a trace log, produced from the BitBlaze framework. The trace log contains information about tainted data and allows an analyst to do a *backward slice*, a list of instructions that were used to compute the operand in question. The first instruction in the backward slice is the root cause. In ReTracer [28], the faulting memory dereference is marked as tainted. Then a backwards taint analysis is performed to identify the origin of the tainted value. The function that is the source of the corrupted value contains the root cause.

Another area of root cause analysis is fine-grained memory instrumentation for the purpose of catching memory violations early. Two notable works in this area are Valgrind [37] and ASan [41]. The tools rely on maintaining a shadow memory that tracks if each byte in application memory is safe to access. Through dynamic or static instrumentation, both tools pad every alloca-

tion on the heap. The padded region is marked as unsafe, and when allocated chunks are freed, they are also marked as unsafe. Whenever the program tries to access memory at a virtual address, the memory error detector looks up the virtual address in shadow memory. The tools report an error if the shadow memory says that the virtual address is marked unsafe. With this method these tools can detect heap buffer overflows, use-after-free bugs, and double frees.

While both use a dynamic analysis, Valgrind uses dynamic instrumentation whereas ASan inserts instrumentation at compile time (static) which allows it to catch invalid memory accesses of stack variables. Also, ASan implements a more efficient shadow memory encoding scheme [41]. On the other hand, Valgrind does not require the program to be recompiled.

With respect to exploit development, the value of knowing a crash's root cause is in making it easier to understand the space of inputs that trigger a bug. This alerts an attacker to how much freedom they have in manipulating the input to generate the exploit. Even though root cause analysis is not enough to diagnose exploitability, it is a prerequisite for *TEASER*.

2.2 Crash Triaging

Crash triage research centers around labeling the severity of a crash without any intent to explain how the crash came about. Particularly, the dominant questions are over a set of crashes, C, and a set of bugs, B related to such crashes: (1) what is the length of B, or in other words, which crashes share the same root causes, and (2) for a given crash $c \in C$, what is its severity (i.e., *exploitable*, *unexploitable*, or *unknown*) [1] [4].

Microsoft's `!exploitable` employs many heuristics for deciding the severity of a crash [9] . For example, if the crash is a violation of Data Execution Prevention (DEP), then the crash is labeled *exploitable*. On the other hand, `!exploitable` labels a stack exhaustion as *not exploitable*.

A weakness of this approach is that a heap corruption is always considered *exploitable*. Even though this is a safe assumption, not all heap corruptions are exploitable which means that this rule can produce many false positives that misrepresent the severity of a crash. Furthermore, crash triage tools do not provide insight on how to develop an exploit for a crash labeled as exploitable or non-exploitable.

2.3 Automated Exploit Generation

This research views exploit generation as a program verification problem where properties of security (i.e., pointers stay within the bounds of their allocated buffers) are replaced with properties of exploitation. One such property is arbitrary control of a function pointer in memory or return address on the stack such that it can contain a pointer to the attacker's shellcode. Similarly, another property describes a write-what-where primitive.

The automatic exploit generation problem involves designing an algorithm that takes a crashing input I that triggers a exploitable property and a program P and returns a new input $I_{exploit}$. Automatic exploit generation seeks to automate the generation of an exploit that satisfies the same control path as I but with the extra constraint that the instruction pointer contains a reference to the attacker's shellcode or Return Oriented Programming (ROP) chain [23] [33].

There are a few variants, such as Automated Exploit Generation (AEG) [23], Automated Patch Exploit Generation [25], and exploit hardening [40], which aims to create an exploit that bypasses exploit protection mechanisms.

In recent years, automatic exploit generation has gained more popularity in academic cyber security research due to the Cyber Grand Challenge, a Defense Advanced Research Projects Agency (DARPA) sponsored program created to assess cyber reasoning systems [16]. The winning system for the Cyber Grand Challenge, Mayem [26], employs the above research on automated exploit generation for its exploit generation component.

Despite this success, the academic prototypes for symbolic execution frameworks employed by automated exploit generation systems are not usable on all real world programs, especially event-driven and C++ based applications [34]. Additionally, a weakness of automatic exploit generation is that it does not suggest all the possible ways a program can be exploited. Last, current automated exploit generation methods do not realistically model heap metadata or the heap layout which are essential prerequisites for constructing reliable heap-based exploits [31].

Chapter 3

Foundations

Here we discuss two areas that must be understood for the rest of this paper: (1) dynamic taint analysis and (2) common memory corruption exploit techniques.

3.1 Dynamic Taint Analysis

A dynamic taint analysis allows one to precisely follow data that has been marked. When data is marked, we say that it is *tainted*. Taint systems require (1) *taint sources* - a specification of which input sources introduce taint into the program, (2) a model for propagating taint, and (3) *taint sinks* - a specification of instructions that should not operate on the tainted data.

3.1.1 Taint Propagation

Algorithm 1 shows a simple algorithm for propagating taint over the course of an instruction trace. Given an instruction I with a destination operand I_{dst} and source operands I_{srcs}, the taintedness is the union across all $I_{src} \in I_{srcs}$ for the instruction as shown in Algorithm 1. A taint source is introduced by a particular instruction type (i.e., `read` syscall or a specialized hypercall that tells the taint manager to label n bytes starting at a particular address).

Algorithm 1 Simple algorithm for propagating taint throughout a system.

1: **procedure** SIMPLETAINTANALYSIS
2: **for** $I \in InstructionTrace$ **do**
3: $isTainted[I_{dst}] \leftarrow \bigcup_{I_{src} \in I_{srcs}} isTainted[I_{src}]$

However, Algorithm 1 does not track which bytes from the taint source influenced a particular tainted value at a taint sink which is useful for more refined analyses. Also, the taint system described above does not keep track of a notion of complexity for tainted value. For example, if an array of tainted bytes from a taint source is used to compute a checksum, the resulting value will surely be tainted. But since its value is the result of many operations on many different tainted bytes, we might want a way to denote the value as a more complex function of the taint source or derived from a set of bytes from the taint source than as simply tainted or not tainted. We use a measurement from Dolan-Gavitt et. al called Taint Compute Number (TCN), which is the number of arithmetic operations used to compute a tainted value [30]. We also assign each byte in a taint source a unique taint label. We modify Algorithm 1 to track the TCN and the *taint label set* for each register or memory cell, where the taint label set is the set of taint source labels that computed the tainted value [29]. Given an instruction I with destination operand I_{dst} and source operands I_{srcs}, the taint label set is the set union across all I_{srcs} and the new TCN is the max TCN across all I_{srcs} incremented by one. In this case, an address or register is tainted if the length of its taint label set is greater than zero.

The resulting algorithm, shown in Algorithm 2, is also a more realistic taint system. Since we would like to taint at the byte level, but assembly instructions usually execute at the word level (usually four or eight bytes), it can be unclear how to propagate taint. For example, when doing a bitwise `or`, if only the least significant byte of the source operand is tainted than the resulting taint operation should only mark the least significant byte of the destination operand as tainted. While for an instruction like a multiply, we aggregate, or *mix*, the taint label set and TCN information for each byte in the source operands and then propagate those resulting values to each byte in the destination operand. Algorithm 2 makes a distinction between instructions like a bitwise `or` and a multiply with the notion of *parallelComputeOp* and *mixedComputeOp*, respectively [17].

Normally, for operations that dereference memory locations, the resulting dereference is included as a source operand instead of the pointer. However, a taint system that propagates taint through pointer dereferences, also called *tainted pointer*, will include the pointer in the source operand list [17].

3.1.2 Exploit Detection (sinks)

In this section we will describe a taint policy meant to signal exploitation of an application. The model below is borrowed from TaintCheck, a exploit detection tool built on the BitBlaze

Algorithm 2 Algorithm for maintaining taint label sets and the taint compute number throughout an execution trace

1: **procedure** COMPLEXTAINTANALYSIS
2: **for** $I \in InstructionTrace$ **do**
3: **if** $mixedComputeOp(I)$ **then**
4: $mixedLabels \leftarrow \{\}$
5: $maxTCN \leftarrow 0$
6: **for** $I_{src} \in I_{srcs}$ **do**
7: **for** i from 0 to $len(I_{src})$ **do**
8: $mixedLabels \leftarrow mixedLabels \cup LabelSet[I_{src} + i]$
9: $maxTCN \leftarrow \max \left[maxTCN, TCN[I_{src} + i] \right]$
10: **for** i from 0 to $len(I_{dst})$ **do**
11: $LabelSet[I_{dst} + i] \leftarrow mixedLabels$
12: $TCN[I_{dst} + i] \leftarrow 1 + maxTCN$
13: **else if** $parallelComputeOp(I)$ **then**
14: **for** i from 0 to $len(I_{dst})$ **do**
15: $LabelSet[I_{dst} + i] \leftarrow \bigcup_{I_{src} \in I_{srcs}} LabelSet[I_{src} + i]$
16: $TCN[I_{dst} + i] \leftarrow 1 + \max_{I_{src} \in I_{srcs}} TCN[I_{src} + i]$
17: **else if** $assignmentOp(I)$ **then**
18: **for** i from 0 to $len(I_{dst})$ **do**
19: $LabelSet[I_{dst} + i] \leftarrow LabelSet[I_{src} + i]$
20: $TCN[I_{dst} + i] \leftarrow TCN[I_{src} + i]$

framework [38]. The model presented below is not designed to guide the exploitation process and we are not trying to show any fallacies in TaintCheck. However, we will use it as a straw-man against our system in order to show how *TEASER* informs the exploit development process.

Taint-based exploit detection specifies certain sink instructions that must not operate on tainted data. In this section, we define the taint sinks used for exploit detection. We used existing literature to form these exploit detectors [38]. While the intent of these detectors was to detect an exploit event, we will view these detectors as *potential* exploit points (see Section 4.2).

We describe exploit detection in two parts, *control-related detectors* and *data-related detectors*. Primarily, instructions that influence control flow (i.e., `ret`, `jmp [eax]`, `call [eax]`) have the most exploit potential. When the tainted operand has a TCN of 0, we know that the jump or call target is a direct copy of the *taint source*. This is indicative of exploit potential.

A taint policy for *control-related detectors* is:

P_1) A return instruction must not take a tainted value from the stack

P_2) Indirect `jmp` and `call` instructions must not have tainted targets

And a taint policy for *data-related detectors* is:

P_3) A pointer load and store must not be derived from a tainted pointer

P_4) Arguments for system calls should not be tainted

P_5) Arguments for function calls should not be tainted.

Notice that exploit detection does not have a detector for a read of tainted data. While taint-based exploit detection, existing exploit detection frameworks that use program input as the taint source would not use a tainted read as a detector because there are many benign cases for reading tainted input [38]. However, any data from a memory corruption could violate the assumptions made by the programmer, so a more precise exploit development-focused taint system would want to measure reads of tainted data. Section 5.2.5 demonstrates the value of a more fine-grained detector in assessing information leakage exploits.

3.2 Exploit Techniques

The exploit development process involves a series of techniques that the hacker community has curated over the years in order to use a bug or series of bugs to exploit a program. We will

assume that the exploit development process begins with a crashing input, I, on a program P, where C is the crashing instruction with bug type T. T is one of *use-after-free*, stack-based/heap-based buffer overflow (also referred to as buffer overwrite), or stack-based/heap-based buffer overread. In some cases P only crashes on I when it is run under an instrumented version of P that aborts on fine-grained memory violations (i.e., accessing out of the bounds of a heap allocated object). The first section discusses the classes of memory corruption bugs attackers use to construct memory corruption-based exploits. The second section describes the desired end state of an exploit and how our detectors inform that we have come closer to that end state.

3.2.1 Bug Types

We consider three general classes of memory corruption bugs: (1) use-after-free, (2) buffer overflow, and (3) buffer overread. Although there are more types of memory corruption bugs, we found these three a sizable portion of all memory corruption vulnerabilities and also represented in our corpus (see Section 5.1). A more comprehensive report on bug types and memory safety can be found at [42]. These three bug classes introduce memory corrupted-data to unexpected locations in the program. For example, a use-after-free is a dereference of a pointer that points to memory that has been reclaimed (and perhaps subsequently reallocated) by the heap allocator. On the other hand, a heap buffer overflow allows an exploit source to alter heap metadata or the contents of an another object on the heap. A buffer overread reads data outside of the bounds of a buffer. This data could be application/runtime metadata or the contents of other data objects on the heap or stack. However, the action of the buffer overread does not imply that the sensitive data will be leaked to the attacker.

3.2.2 Exploit End States

Using a heap buffer overflow as an example here are four examples of turning a bug into an exploit: (1) achieve an *instruction pointer overwrite* or *pointer write control* by corrupting a nearby heap object, (2) gain a *pointer write control* by corrupting any heap metadata that the allocator uses as pointers, (3) create a *type confusion* by corrupting heap metadata in order to trick the allocator into returning a chunk that overlaps an existing chunk, or (4) achieve a *read disclosure* by triggering control flow that leaks the memory corrupted data (see Section 3.2.3 for a nefarious use of a read disclosure).

Notice, that not all of these end states represent the terminal stage in exploit development, i.e. remote code execution. Once it can be shown that an input triggers (2) or (3), more work must be done to create a working exploit. The purpose of these end states is to connect a taint policy to the stages of exploit development. Even though they are not an exhaustive list of all the ways to exploit a bug, they are some of the most common ways. We chose them because, intuitively, we should be able to measure these end states with a taint-based approach.

3.2.2.1 Instruction Pointer Overwrite

The most powerful exploit end state is the *instruction pointer overwrite*. In this case we are able to overwrite a stored function pointer or a return address on the stack. Call the location of the stored function pointer Loc_{pc}. With an *Instruction Pointer Overwrite* bug, we are interested in the TCN of Loc_{pc}. If it has a low TCN at the point of an *indirect control-flow detector* this indicates a higher chance of an arbitrary control. Ultimately, an attacker would like to create a modified input I_{new} such that the same constraints π are satisfied, but Loc_{pc} contains an address that points to the attacker's shellcode.

The control-related taint policy (P_1 and P_2) in Section 3.1.2 detects this exploit end state. If we detect tainted indirect control flow and the TCN of Loc_{pc} is greater than 0, then we could have a false positive. For example, a program that uses attacker controlled-input as a offset into a jump table will violate the tainted indirect control flow policy (P_2). However, this case can be benign and is a common idiom.

3.2.2.2 Pointer Write Control

Another powerful exploit end state is *pointer write control* also called a write-what-where primitive. In this situation, our crashing instruction C dereferences a pointer, $pointer$, and then writes some data D to the location pointed to by $pointer$. Exploit developers call this control over $pointer$ a *write-what-where* primitive. Turning a *write-what-where primitive*, which is fundamentally a data-related attack, into a *instruction pointer overwrite* involves leveraging knowledge of a program's runtime environment and of application specific data structures. For example, on a Unix-flavored program an attacker can use a write-what-where primitive to overwrite a function pointer in the Global Offset Table (GOT) [36] [39].

We can measure this end state with a tainted pointer write policy (P_3) from Section 3.1.2. Furthermore, if $pointer$ has a TCN of 0, then it is simply a direct copy of the bytes from the *taint*

source, signaling that the memory corruption can give a *write-what-where primitive*.

3.2.2.3 Type Confusion

Type confusion is stage of exploit development that can give the attacker instruction pointer overwrite or pointer write control. Typically type confusion occurs with two objects O_1 and O_2, of runtime types A and B, respectively, that are allocated on the heap. Our program creates and stores a $Pointer_{O_1}$ and $Pointer_{O_2}$, which are the memory addresses returned from the allocator.

We define type confusion as cases where $Pointer_{0_1}$ is treated as a pointer to an object (O_2) of runtime type B where $A \neq B$ and B is not a subtype of A. In this situation, a write to O_1 is actually corrupting O_2, and the next access of O_2 is undefined behavior. The most powerful form of this situation is if O_1 is some array of attacker-controlled binary data and O_2 is an object with a function pointer. From here, an attacker can find offsets into O_2 that contain the ideal data to corrupt (i.e., function pointer or writable pointer) to elevate the *type confusion* technique into an *instruction pointer overwrite* or *pointer write control* respectively.

The exploit detection system described in Section 3.1.2 would have trouble measuring or detecting the presence of type confusion. This is because none of the detectors can resolve pointer ownership or aliasing. On the other hand, the control flow detectors and tainted pointer detector (P_1 - P_3) are still useful in the case where type confusion is used to construct a more powerful exploit primitive.

3.2.3 Read Disclosure

A read or information disclosure is the "intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information" [8]. In particular, we are interested in leveraging a memory corruption to leak sensitive data in memory. Heartbleed is perhaps the most infamous information disclosure. The vulnerability gave attackers the ability to read up to 64K in application memory, which included TLS private keys [15].

Our existing taint policy in Section 3.1.2 cannot measure this exploit end state. First, the taint source is input already available to the attacker. So tracking its data flow to assess if it is leaked is moot. Second, the existing detectors are not general enough to measure tainted data flow to an I/O sink.

Chapter 4

Methodology

In order to apply dynamic taint analysis to our problem we must answer three questions:

1. What data should be labeled as tainted?
2. How is taint propagated during a program's execution?
3. When and where do we query taint?

The first requirement is satisfied by what we call an Exploitation Source, E_s, which is the memory violation that is controllable by the attacker. We handle the second question with a generic taint system explained in 3.1.2. The third question is settled by our notion of Exploit Sinks, E_k, or points in the program that should not operate on data from a memory corruption.

4.1 Exploit Sources

We have devised a general framework for applying taint labels to data involved in a memory corruption. Specifically, we are interested in applying taint labels to data that violates the program's runtime model or assumptions made by the programmer. These taint labels represent each byte in the Exploitation Source, E_s. In some cases, an attacker is given a primitive that allows her to control memory at some relative offset from a program data structure. Other times, an attacker can cause the use of a dangling pointer. Or an attacker can influence one of the arguments in a function call like `memcpy` to cause a buffer overflow on either the stack or the heap. In all these cases, a bug is associated with n bytes in memory that violate the assumptions made by the program's runtime model or the application invariants. We denote the specification of where/when an E_s happens in the

program and the address of the memory corrupted data as the Root Cause Taint Point (RCTP). The information from a fine-grained memory checker, like ASan and Valgrind, and the GNU Debugger guides the user to use program knowledge to identify RCTPs.

Specifically, a RCTP is all the information needed to label the E_s as tainted during a program trace. Therefore, we define RCTP as a four-tuple:

$$RCTP = (Where, When, Buf, BufSize)$$

where $When$ represents the number of times the $Where$ must be reached during the program trace before the taint system labels Buf to accommodate loops. For the purpose of our system we use the program counter to represent $Where$, but in principle another convention could be used (i.e., a hash of the program counter and the callstack).

The end goal of our root cause detection and labeling strategy steps explained below in Section 4.1.1 and Section 4.1.2 is to determine the RCTP. We encode the RCTP in the source code as a call to the function `vm_taint_exploitation_label(char *Buf, ssize_t BufSize, unsigned When)` which communicates to PANDA which bytes to label as tainted.

4.1.1 Root Cause Detection

We use existing tooling to find the location of the root cause of a memory corruption bug. This process entails finding the *where* and *when* components of a RCTP. The tools (Valgrind and ASan) run a program P under fine-grained memory instrumentation in order to find any instance of an out-of-bounds memory read/write or a use-after-free. Then in a debugger we explore what variable the program is trying to access.

We refer to our example, Figure 1.1, from Section 1.2. We assume that we are given a crashing input resembling Figure 4.1.

```
p
c
r 0
p 0
c
```

Figure 4.1: Crashing input for our motivating example. Such an input could have easily have been generated from a fuzzer. The input allocates a print object, frees it, and then uses a dangling pointer to try to print the object. Valgrind and ASan report the memory violation, but the program runs without error when it is not instrumented.

But when we run the program there is no crash. Running it under Valgrind reveals that there is an invalid read in the print case of our interpreter. Upon further analysis, we learn that we are looking at a use of a dangling pointer. Even though the print object had been freed on line 3, the program still uses a function pointer in this object during the print command on line 4. The program does not crash because the object remains mostly intact after the free.

4.1.2 Labeling Strategy

Even though a root cause detection tool gave us the location of an E_s, we still need to determine what bytes we label. The goal of our labeling strategy is to identify the *what* component of a RCTP for a particular bug. In this case, it is useful to determine what type of bug we triggered and apply a labeling strategy based on that bug type. The bug type is informed by a combination of the root cause detection step in Section 4.1.1 and manual analysis. In the case of our example, Figure 1.1 on input Figure 4.1, our root cause detection tool tells us that the bug type is a use-after-free. However, in other cases, we would want to distinguish between a pointer-based buffer overflow (Section 4.1.2.2) and a *n*-byte buffer overflow (Section 4.1.2.3). Determining what to label for a bug still requires human judgment, and the strategies presented in the next three subsections can be thought of as guidelines.

4.1.2.1 Use-After-Free Tainting

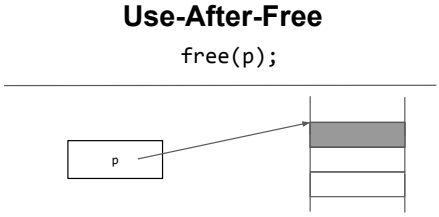

Figure 4.2: Root cause taint model for a dangling pointer. In this example, after the code `free(p)` is executed, we label its malloc chunk as tainted. The gray region represents the memory that is tainted after call to free.

Figure 4.2 demonstrates the root cause taint model for use-after-free bugs. In this situation, Valgrind and ASan emit an error during program execution signaling a use of a pointer after its corresponding malloc chunk has been freed.

In this situation we label the recently freed chunk as tainted. Although it is not clear that we are going to have the ability to freely control what is in that chunk, the idea is that an attacker with an intervening call to malloc could place arbitrary data in that free chunk.

```
1       . . .
        case 'r':
3       . . . // parse arguments for release command
        free(mem_chunk_list[id].pointer);
5       vm_taint_exploiation_label(mem_chunk_list[id].pointer,
        mem_chunk_list[id].size,
7       /* When */ 0);
        . . .
```

Figure 4.3: Our instrumentation for our use-after-free root cause in the motivating example.

Figure 4.3 shows the source code instrumentation to label the recently freed chunk as tainted. Notice that the $Where$ argument to `vm_taint_exploitation_label` is 0. This means that we taint the root cause on the first time we reach the instrumentation function.

4.1.2.2 Pointer Arithmetic-based Buffer Overflow Tainting

We develop a way to model what bytes to taint after a pointer arithmetic-related error. Although pointer arithmetic-based buffer overflows can manifest in many different ways, we focus on two common cases: an out-of-range pointer offset and out-of-bounds pointer increment. In the first case, we index from a pointer using an untrusted value that causes a program to access memory past the bounds allocated for the pointer. In the second case, we increment a pointer too many times such that it points past the bounds of a buffer.

Figure 4.4 represents a root cause tainting decision for our two cases of pointer arithmetic-based memory corruptions. On the left the attacker possesses the ability to write data at an arbitrary offset from some location in memory. In our Root Cause Taint Model, we simply taint the bytes at `dst + foo` after the write happens.

For the example on the right in Figure 4.4, `dst` points outside of the bounds of a buffer after the conditional expression is executed. This data could be valid in the context of other parts of the program. For example, perhaps the data being accessed in the out-of-bounds read is part of an adjacent heap object. We would not want to taint this data because there could potentially be valid accesses of that data. Our solution, depicted on the right side of the figure, is to taint the pointer, `dst`, and enable tainted pointer such that everything dereferenced from `dst` is tainted. However, data out of the bounds of the buffer will only be marked as tainted if it is accessed through the tainted

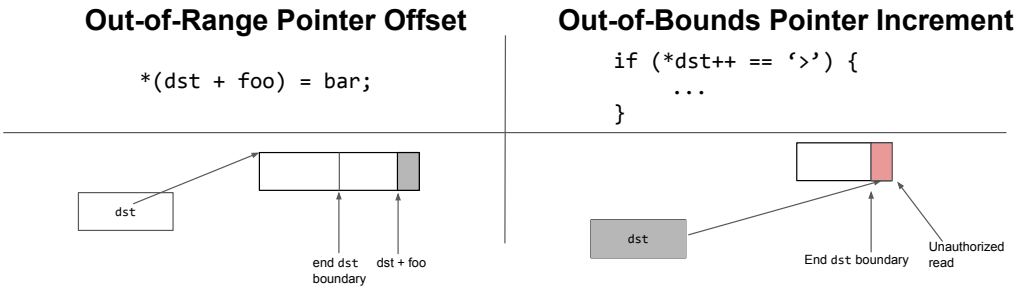

Figure 4.4: Root cause tainting for two pointer arithmetic-related cases. The left, "Relative Buffer Overwrite", represents a write-what-relative-where condition in which the attacker has semi-control over `dst` through the variable `foo`. The right example is an out-of-bounds read. For the right case, after the conditional expression is executed, the pointer begins to point out of the bounds of an allocated buffer. We mark the pointer as tainted instead of the data it points to. The gray regions represents the bytes we label as tainted after the line of code is executed. And the red region represents bytes of the unauthorized read.

pointer. The assumption for this labeling solution is that once a pointer points past the bounds of a buffer it is uncommon for the pointer to be corrected to point back within the bounds of the buffer.

4.1.2.3 *N*-byte Buffer Overflow Tainting

The last category of root cause taint labeling, Figure 4.5, shows how we label buffer *n*-byte overreads and overwrites as tainted. Note that we use the function `memcpy` in these examples, but in principle, we could use any function that causes an *n*-byte overflow.

Figure 4.6 represents the difference in instrumentation for an *n*-byte overread versus overwrite. In both cases we label the bytes in the `dst` buffer. Consider the alternative of labeling the bytes that were accessed outside of the `src` buffer before the call to `memcpy`.

However, those bytes could be part of another chunk on the heap. Any valid use of data from that chunk would be falsely marked as tainted. We believe our taint model is superior because it only taints the bytes outside of the bounds of the `src` buffer when they are copied over to the `dst` buffer.

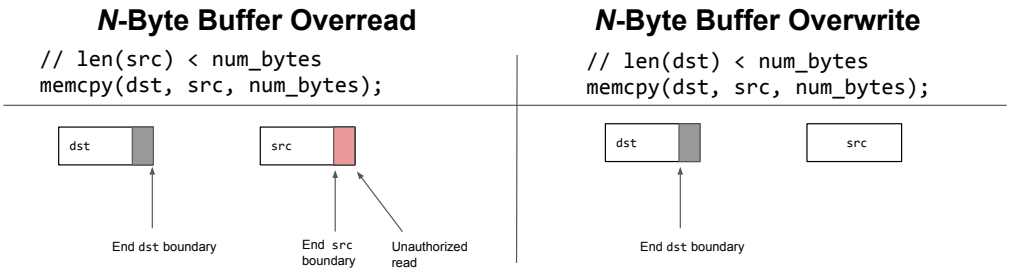

Figure 4.5: The left example shows an unauthorized read of n bytes past the `src` buffer. The right example shows a usage of `memcpy` where the destination buffer is overflowed. Again, the gray region represents the bytes we taint after the line of code, whereas the red regions represents the bytes of the unauthorized read.

```
        . . .
// labeling buffer overread
taint_label(dst + len(src),
            num_bytes - len(src),
            /* When */ 0);


// labeling buffer overwrite
taint_label(dst + len(dst),
            num_bytes - len(dst),
            /* When */ 0);
        . . .
```

Figure 4.6: Our instrumentation for buffer overread and overwrites using the source code from Figure 4.5 as an example.

4.2 Exploit Sinks

An Exploitation Sink, E_k, is a program point that operates on sensitive data controlled by E_s. This can be one of many different kinds of desirable exploitation events, such as the use of a return address or function pointer found to be controlled by an E_s. The proposed exploitation detectors D are heavily influenced by the taint policy (P_1 - P_5) described in Section 3.1.2, our section on exploit detection.

D_1) *tainted return address* - Is the return address on the stack tainted when a function executes the `ret` instruction? (P_1)

D_2) *tainted indirect control flow* - Does tainted data influence an indirect `call` or indirect `jmp` instruction? (P_2)

D_3) *tainted branch* - Are flags used in a branching instruction tainted?

D_4) *tainted heap metadata read* - Is data read by the allocator tainted?

D_5) *tainted function argument* - Are any function arguments on the stack tainted? (P_5)

D_6) *tainted pointer read/write* - Is a pointer that is used to read or write data tainted? (P_3)

D_7) *tainted read* - Is the data from a virtual memory read tainted?

Given the trigger of a detector $D_i \in D$ at a program counter PC, we can declare an Exploitation Sink as a three-tuple:

$$E_k = (Where, When, D_i)$$

where $Where$ represents the program counter at the detector and $When$ represents the same value as $When$ in Section 4.1. Given this model, we observe how the program interacts with memory corrupted data that clearly violates programmer assumptions.

4.3 *TEASER*

TEASER uses the following open source software: PANDA, Valgrind, ASan, and LLVM. The system is user driven, meaning each step requires the user to interpret the output from the previous component and transform it in an intelligent way for the next step. The system architecture

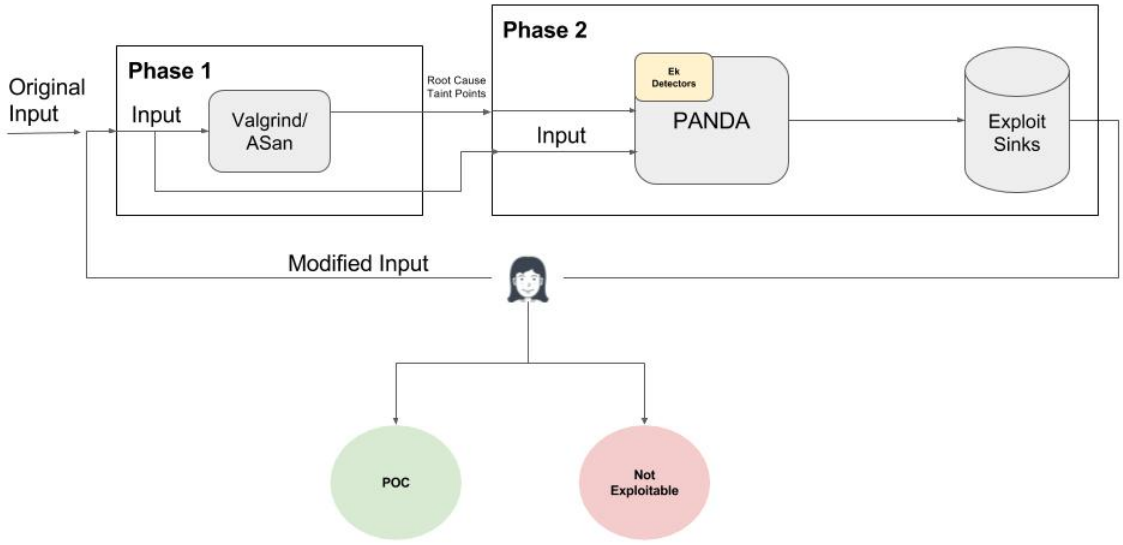

Figure 4.7: *TEASER* architecture

is described in Figure 4.7. *TEASER* is executed in two phases that are driven by an analyst. The goal of Phase One is to calculate the Root Cause Taint Point for the program input. The goal of Phase Two is to measure the impact or exploitability of the root cause by monitoring our seven detectors outlined in Section 4.2. After Phase Two the analyst chooses to either refine the input in order to increase the number of detectors reached from a root cause or conclude whether or not the program is exploitable.

4.3.1 Phase 1: Root Cause Taint Modeling with Valgrind/ASan

We use two open source dynamic analysis tools, Valgrind and ASan, for detecting the root cause of a memory violation. Section 2.1 on root cause analysis explains how these tools work. Compared with other tools in the root cause field, we use these tools because they are popular in the community and reliable. Also, when run on programs with debug information enabled, these tools will provide line and filename in the source code associated with the memory corruption. Having source code level information makes it easier to diagnose the root cause.

We found that both tools complemented each other for *TEASER*. We were able to run Valgrind on the actual binaries that we ran on the PANDA guest machine. We used this feature to confirm that the bug existed in the code. On the other hand, ASan was useful because it gave more

detailed information about the root cause such as a hex dump of memory surrounding the memory violation. Since none of our bugs in our corpus were stack-based buffer overflows, Valgrind and ASan were consistent in the errors they reported. However, in principle, having ASan is useful because it reports memory violations for stack-based variables.

When we get a filename and line number tied to a heap-related error, a human analyst manually instruments the source code with a hypercall that instructs the taint system to label the newly corrupted data as tainted. These tools report the presence of errors like a heap buffer overread, heap buffer overwrite, or use-after-free. These bugs are further explored and diagnosed with a debugger. We used the GNU Debugger (GDB) [11] for our evaluation, but in principle any debugger can be used.

4.3.2 Phase 2: Taint Propagation and Exploit Sinks with PANDA

PANDA is an architecture neutral dynamic analysis system built on top of the QEMU emulator [30]. PANDA introduces record replay and a plugin architecture to QEMU.

Primarily, we use PANDA as a whole system, architectural neutral taint system with the ability to add callbacks at certain points during a program trace. The taint system is loosely based off of the one in PIRATE, from Wheelan et al. [43]. PANDA introduces a plugin for lifting Tiny Code Generator (TCG), QEMU's emulation Intermediary Language, to LLVM instructions. This plugin uses the TCG to LLVM lifter in S2E, a project for analyzing software using selective symbolic execution [27]. We rely on a set of hooks for a subset of LLVM instructions in order to perform certain taint operations such as taint labeling, taint propagation, and taint querying [24] [43]. Furthermore, the PANDA taint system has the ability to propagate taint through dereferences of tainted pointers. We call this feature *tainted pointer mode*. Tainted pointer mode is turned off by default except for the case where we taint a corrupted pointer as our root cause (see Section 5.2.3).

For this project, we have developed two PANDA plugins to facilitate labeling tainted data at an E_s and querying the taint status of an operand at an E_k. Primarily we use Debugging With Attributed Record Formats (DWARF) information, the debugging file format for Executable and Linkable Format (ELF) binaries, to determine the virtual address of the `malloc` and `free` functions. We trace calls to `malloc` and `free` to figure out when the allocator is executed. We query the taint status of all bytes read, using PANDA's facilities for hooking virtual memory reads and writes, and also specify if the read is of heap metadata (D_4 and D_7 from Section 4.2). We know this if we are in the allocator when the read happens.

In PANDA we can also hook every `call` and `ret` instruction. At the call points we check for tainted function arguments (D_5) assuming the C calling convention `cdecl`, and at `ret` points we check for the taintedness of the return address on the stack (D_1).

Last, our LLVM instrumentation provides callbacks on instructions involving branching (D_3), indirect control flow (D_2), and pointer dereferences (D_6). We use these to query taint on the related operands.

4.3.3 User Component - Exploit Refinement

The result of Phase Two is a list of *exploit sinks* that are derived from data in a memory corruption. The exploit sinks contain the location in the source code in which they occur. They also contain the TCN of the tainted bytes used. After Phase Two, the analyst attempts to use this information to make the input more exploitable or rule that that bug is exploitable or not exploitable. The goal of the refinement process is to (1) assess the amount of control the exploit source has over the program and then (2) modify the input to gain more control.

An analyst may refine the exploit by attempting to answer the following questions:

1. If we have a use-after-free, what data is accessed through the dangling pointer?
2. If we have a buffer overflow, are we able to control what chunk we are overreading or overwriting?
3. Similarly if we have a buffer overflow, can we change the number of bytes written to or read from past the bounds of the buffer?
4. How does corrupted heap metadata shape the behavior of the allocator?

Notice that these questions can be framed in terms of our detectors in Section 4.2. Questions one and two can be answered by observing which detectors (excluding D_4) the exploit source triggers. The last item is measured by our tainted heap metadata detector, D_4.

Sometimes we are testing the exploitability of an API provided by a library, rather than a standalone application. In these cases we run the API in a harness that loads input from a file and then provides it to a *target* function that implements select functions from the API. With this setup, we can manipulate the heap before and after the target function in order to ensure certain heap chunks get corrupted by a heap buffer overflow or use-after-free.

By changing the input or the heap layout before the target, the analyst can better assess exploitability. This process is iterative, and the analyst is done when they have reasonable evidence about the exploitability of a program.

4.4 Implementation

Our system runs on x64 Ubuntu 16.04, and we analyze programs that have been compiled for an i386-based Debian Wheezy machine. The reason for the architecture disparity is that PANDA must run on a 64-bit system but it only has operating system introspection support for 32-bit systems.

Chapter 5

Results

We ran *TEASER* on six open source programs: BoringSSL, OpenSSL, SQLite, Web Open Font Format (WOFF2), libxml2 and c-ares. We compare the results of our system with a modified version of our tool that uses the input (instead of the root cause) as the taint source. The results demonstrate that root cause tainting gives a more precise picture of the possible exploit paths for a bug. We also provide case studies for how *TEASER* and our proposed exploit development workflow can be employed in a real-life situation.

5.1 Experimental

We obtained six programs from the Google Fuzz Testing corpus [10]. The corpus is meant to used as a benchmark for fuzzers. The corpus consists of 13 open source libraries with expected outputs from a fuzzer that trigger either a memory corruption, information leak, or an out-of-memory error in each library. We trimmed the test corpus to 6 programs that had a memory corruption bug and could be triggered on 32-bit architectures. This is because PANDA, our dynamic analysis framework of choice, does not have operating system introspection support for 64-bit Linux systems.

5.2 Buggy Programs

Two of the six programs (c-ares and OpenSSL) have known exploits while the others were marked as CVEs but do not have publicly released exploits. We saw this as an advantage of our dataset because we had to confirm that either a bug was exploitable or not exploitable. For the

exploitable programs, we also ask if the bug yields multiple paths to exploitation. On the other hand if a bug is not known to be exploitable we seek to show that no control flow (detectors D_1 - D_3) is derived from the memory corruption. It is especially important to show that a bug is not exploitable because that is the more common and ambiguous task for an analyst.

5.2.1 Results Overview

Table 5.1 and Table 5.2 summarize the results of our experiments on our corpus of 6 programs. Table 5.1 describes the difference between the information gleaned from root cause detection tools, *TEASER*, and the information security community. We looked at bug disclosures and released proof of concept exploits (if available) for each bug in order to gauge the community reaction. In all, this table shows that *TEASER* confirms that the exploit sources for all of the bugs do not influence any of the control flow detectors (D_1 - D_3). However, many of the bugs induce reads of tainted data (D_7) and the c-ares bug causes a read of invalid heap metadata (D_4) which indicates exploitability.

Program	**Valgrind/ASan**	***TEASER***	**Community**
BoringSSL	Identify Use After Free (UAF) at `asn1_lib.c:459` in the `ASN1_STRING_free` function	Confirms that no control flow decisions are made based off the freed data.	Bug report shows that bug possibly only affects WebCrypto in Chrome. Marked high severity, but no known exploit. [14]
Libxml	Identify 1 byte heap buffer overread at `parser.c:10666` and various uninitialized reads.	Confirms that no control flow decisions are made off of data from buffer overread.	Ruled as memory leak, and low impact.
C-ares	Identify 1 byte heap overflow in `ares_create_query.c:196`	Identifies instructions in allocator that use tainted data (useful in identifying how a heap metadata overwrite influence program execution).	Known POC that leverages one byte overflow to change heap header information in order to gain root on Chrome OS [5]
OpenSSL (Heartbleed)	Identify Heap buffer overread at `ssl/t1_lib.c:2586`	Identifies read disclosure sinks, confirms that there are no indirect control flow decisions made based off of data from read disclosure.	Created an exploit that leaks 64k of memory including private keys
Sqlite3	Identify UAF at `sqlite3.c:127739` in `exprAnalyze`.	Confirms that data in UAF does not influence control flow.	Security severity high, but no known exploit [19]
Woff2	Identify a buffer overwrite at `woff2_dec.cc:500` in the function `ReconstructGlyf`.	Checksum function returns tainted value. Tainted data access in glibc malloc function.	Marked as severe on Chromium, but no known exploit

Table 5.1: Summary of results of *TEASER* compared with other tools and the "community"

For five of the six programs in the dataset, we saw that labeling from the root cause rather than the program input generated fewer exploit sinks (see Table 5.2). And in the case of c-ares and SQLite, an input-based taint source would not have caught *any* of the same exploit sinks as a root cause-based taint source. This demonstrates that root cause-based tainting also provides better information than input-based tainting.

Program	Number of E_ss file input (bytes labeled)	Number of E_ss root cause (bytes labeled)	Number of unique E_ks for file input	Number of unique E_ks for root cause	Number of E_ks in common	Instructions
BoringSSL	41	16	952	7	5	722
Libxml	60	4	76	43	4	24
C-ares	48	1	24	2	0	68
OpenSSL	41	46,790	22	28	4	42967
Sqlite3	88	776	335	5	0	2051
Woff2	61,644	10	4,430	26	13	276

Table 5.2: Comparison of exploit sinks collected with the taint source as the program input versus the taint source as the root cause. The column labeled "Instructions" represents the number of instructions between the root cause and the first exploit sink.

5.2.2 BoringSSL

BoringSSL is a fork of OpenSSL developed by Google. It is intended to be shipped with the Google Chrome/Chromium browsers and a number of other applications [2].

The fuzzing target calls the function `EVP_PKEY *d2i_AutoPrivateKey(EVP_PKEY **a, const unsigned char **pp, long length)` which takes in a string of bytes and returns a decoded key using one of various key encoding formats. Valgrind reports a use of four bytes of freed data in `ASN1_STRING_free` at `asn1_lib.c:459` (see Figure 5.1). It also tells us the address of the data being accessed.

In GDB we set a break point on this function and track the argument `a` knowing that `a` points to a 16 byte chunk. With each call to `ASN1_STRING_free` we check if `a` points to the chunk where the dangling pointer was used. Through our analysis, we find that it is the fifth function call that releases the target chunk. Therefore, our Root Cause Taint Point is 16 bytes at the variable `a` on the fifth return from `ASN1_STRING_free`.

```
454
    void ASN1_STRING_free(ASN1_STRING *a)
456 {
        if (a == NULL)
458         return;
        if (a->data && !(a->flags & ASN1_STRING_FLAG_NDEF))
460         OPENSSL_free(a->data);
        OPENSSL_free(a);
462 }
```

Figure 5.1: Root Cause Taint Point at the fifth exit of `ASN1_STRING_free`

We feed the Root Cause Taint Point, the program, and the crashing input to *TEASER*. Our system, in this case, reports less information than Valgrind and ASan. While all tools agree that there is an invalid use of data on line `asn1_lib.c:459` (D_7), Valgrind and ASan also report a double free on line 461. *TEASER* also reports usage of malloc metadata (D_4), but closer analysis

reveals that this is because the allocator replaces the first two words in a malloc chunk with two pointers to other freed chunks. Therefore, usage of this data within the allocator is valid in the context of the program. For more information on the free list, refer to the data definition of a malloc chunk in the glibc malloc internals [22].

TEASER is still valuable here because it reveals that there is only one place where the dangling data is used, and the attacker's best route of exploitation is using an intervening allocation to corrupt `a` such that `a->data` contains an arbitrary pointer to stage an arbitrary pointer read or write. We were able to draw this conclusion by manually examining only seven locations in the source code instead of 952 (see Table 5.2).

5.2.3 Libxml

Libxml, also called libxml2, is a parser for the metalanguage XML, a popular language for designing markup languages. The library is extremely portable and is embedded in a wide variety of applications including web browsers [21].

```
10644 . . .
      // from include/libxml/parserInternals.h
10646 #define MOVETO_ENDTAG(p)                                                \
          while ((*p) && (*(p) != '>')) (p)++
10648
      // parser.c
10650 . . .
      #define RAW (*ctxt->input->cur)
10652 . . .
      void
10654 xmlParseXMLDecl(xmlParserCtxtPtr ctxt) {

10656     . . .

10658     if ((RAW == '?') && (NXT(1) == '>')) {
              SKIP(2);
10660     } else if (RAW == '>') {
              /* Deprecated old WD ... */
10662         xmlFatalErr(ctxt, XML_ERR_XMLDECL_NOT_FINISHED, NULL);
              NEXT;
10664     } else {
              xmlFatalErr(ctxt, XML_ERR_XMLDECL_NOT_FINISHED, NULL);
10666         MOVETO_ENDTAG(CUR_PTR);
              vm_taint_exploitation_label(&CUR_PTR, sizeof(CUR_PTR), 0);
10668         NEXT;
          }
10670 }
```

Figure 5.2: Root Cause Taint Point on line 10667 of parser.c right after `MOVE_TO_ENDTAG`

The crashing input for libxml triggers several memory violations labeled as "uninitialized

reads" under Valgrind. Despite these designations, the bug is really a heap buffer overread that is labeled as having low severity. The overread results from parsing an XML document with an unfinished XML declaration [7].

Figure 5.2 shows the code that triggers the uninitialized reads in libxml. The function `xmlParseXMLDecl` parses an XML declaration header. Line 10658 shows the conditions for a valid parse of the XML declaration header. If, the parse fails, and `RAW` (the byte that `CUR_PTR` points to) does not equal an XML closing tag (`'>'`), the execution will fall through to the else clause on line 10664. Line 10666 in `parser.c` in the `xmlParseXMLDecl` function is the line of code that triggers the overread. `MOVETO_ENDTAG` is a macro that advances `CUR_PTR` until it reaches a closing tag. From this macro definition at the top of Figure 5.2, it is clear that `p` will be incremented potentially past the bounds of the buffer if there is no closing tag.

The libxml case study allows us to demonstrate an alternative labeling strategy defined by our root cause taint model (see 4.1.2.2). Since `CUR_PTR`, or `ctxt->input->cur`, does not point into a valid memory heap chunk after the macro on line `parser.c:10666` is executed and the pointer is used later on in the program, we say that this is a pointer arithmetic-related error. Since it is unlikely that the pointer will be decremented, we can assume any dereference of this pointer is invalid data (either uninitialized or from an adjacent heap chunk). Given the strategy in 4.1.2.2, we instead label the location where `CUR_PTR` is stored rather than the data it points to during the heap read.

The results of our root cause labeling strategy show that there are 43 unique exploit sinks compared to 76 (see Table 5.2) when the program input is the taint source. After investigating the exploit sinks (non unique exploit sinks), we find that 631 are tainted reads (D_7), 240 are tainted pointer reads (D_6), and 50 are tainted function arguments (D_6). Most of the tainted reads appeared to be from accessing `CUR_PTR`. Since there were no control flow-related detections we are able to rule out any ability to use this bug for remote code execution. However, the copious amount of operations on `CUR_PTR`, when it clearly points out of the bounds of a buffer, signals high potential for a read disclosure.

5.2.4 C-ares

C-ares is a library for asynchronous domain name resolution and is used in many applications including, notably, `shill`, the Chrome OS network manager that was exploited in order to gain root on Chrome OS systems [5].

The bug that Valgrind and ASan detect is a one byte overflow in the `ares_create_query` function which is used to compose DNS queries. The bug, shown in Figure 5.3, results from a violation of the parser routine which takes names as period-separated labels where the periods can be escaped with a backslash. In lines 9 to 17, the parser calculates the number of bytes needed for the resulting DNS query, making sure to count escaped characters as one instead of two bytes. In lines 19 to 20, the program is supposed to allocate an extra byte for the last character if the last character is not a period. However, it does not account for the case where the last character is an *escaped* period. Because of this the parsing routine allocates one fewer byte of space for the resulting DNS query string, leading to a one byte overflow. In lines 29 to 43, the parser copies data from `name` to `buf` using `p` and `q` as aliased pointers.

The root cause taint model for a buffer overwrite is intuitive. After the buffer overflow, we taint the bytes that were illegally written past the bounds of the buffer. In the case of `c-ares`, that buffer overflow occurs after line 196 in `ares_create_query.c` (line 47 in Figure 5.3). Line 49 in Figure 5.3 reveals the instrumentation code to label the one byte overflow as tainted.

The first time *TEASER* was used, no detectors were triggered, which meant that even though there was a memory corruption, the data involved was never accessed again. Manual analysis revealed that since the size of the DNS query in the crashing input was less than 80 bytes, the allocated chunk fell inside of the allocator's fastbin when freed. Malloc chunks within the fastbin are padded to align on an 8 byte boundary [22] [32]. We needed to change the heap layout before/after the call to `ares_create_query` and the *input* in order to trigger our various exploit detectors. The exploit refinement process involved two parts:

1. We decided to allocate chunks before the c-ares target and then free those chunks after the c-ares target.

2. We edited the crashing input such that chunks of a specific size that required no padding were allocated. It turns out that this size was 60 for this crashing input.

Once we made these two changes, two detectors for the same instruction were triggered (tainted malloc metadata read, D_4, and a tainted branch instruction, D_3). These detectors occurred only 68 and 80 instructions after the corruption which demonstrates that there is very little activity within `c-ares` to take advantage of after the heap corruption. This idea is in line with the explanation of the c-ares exploit which requires considerable heap manipulation in preparation for the one byte overflow [5].

```
1
  int ares_create_query(const char *name, int dnsclass, int type,
3                       unsigned short id, int rd, unsigned char **buf,
                        int *buflen, int max_udp_size)
5 {
    int len;
7   unsigned char *q;
    const char *p;
9
    // compute length of destination buffer
11  len = 1;
    for (p = name; *p; p++)
13    {
        if (*p == '\\' && *(p + 1) != 0)
15        p++;
        len++;
17    }
    // VULN
19  if (*name && *(p - 1) != '.')
      len++;
21
    // allocate space for *buf variable
23  q = *buf;

25  // Set up the header and advance q past header
    . . .
27  // for each label, we count the data
    // and then copy the length_byte + data into the new buffer, q
29  while (*name)
      {
31      // calculate len, or number of bytes, of label

33      *q++ = (unsigned char) len;
        for (p = name; *p && *p != '.'; p++)
35        {
            if (*p == '\\' && *(p + 1) != 0)
37            p++;
            *q++ = *p;
39        }

41      if (!*p)
          break;
43      name = p + 1;
      }
45  /* Finish off the question with the type and class. */
    DNS_QUESTION_SET_TYPE(q, type);
47  DNS_QUESTION_SET_CLASS(q, dnsclass);
    // root cause taint labeling
49  vm_taint_exploitation_label(q, 1, 0);
    . . .
51 }
```

Figure 5.3: Root Cause Taint Point in for `ares_create_query` in ares_create_query.c

We found two exploit sinks when we tainted from the root cause and upon further investigation, these exploit sinks demonstrated the ability to reliably corrupt heap metadata. On the contrary, tainting from the program input revealed 24 exploit sinks, but these were all benign reads

of the program input in the parsing routine (see Table 5.2). The c-ares buffer overwrite clearly shows the benefit of our approach. Tainting from the root cause highlights to the analyst what influence the one byte overflow has on the program. When a memory corruption does not explicitly use bytes from the input, tainting from the program input is not as informative as tainting from the root cause.

5.2.5 OpenSSL

The crash for this program is tied to CVE-2014-0160, also known as Heartbleed, which allows an attacker with a specially crafted packet to illegally read up to 64k of an application's memory. The bug is a heap buffer overread and generally not exploitable as remote code execution but rather as an unauthorized disclosure of sensitive information such as TLS private keys [15, 6]. Line 10 in Figure 5.4 shows the allocation of memory for `bp`. The size `payload` is extracted from the `pl` packet, but this size is not checked to make sure that it is consistent with the actual size allocated for `pl` on the heap. Line 16 is the location of the buffer overread. Here, `payload` number of bytes are copied from `pl` into `bp`. Line 18 to 19 are the instrumentation code to label the bytes that are illegally read into the variable `bp` (see 4.1.2.3 for a description of buffer overread/overwrite tainting).

```
1  int
   tls1_process_heartbeat(SSL *s)
3          {

5      . . .
       /* Allocate memory for the response, size is 1 bytes
7       * message type, plus 2 bytes payload length, plus
        * payload, plus padding
9       */
       buffer = OPENSSL_malloc(1 + 2 + payload + padding);
11     bp = buffer;

13     /* Enter response type, length and copy payload */
       *bp++ = TLS1_HB_RESPONSE;
15     s2n(payload, bp);
       memcpy(bp, pl, payload);
17     // root cause taint labeling: taint the end of bp
       // where the out of bounds read data goes
19     vm_taint_exploitation_label(bp + 17725, payload - 17725, 0);
       bp += payload;
21     /* Random padding */
       RAND_pseudo_bytes(bp, padding);
23
       r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Figure 5.4: Root Cause Taint Point for OpenSSL heap buffer overread bug in ssl/t1_lib.c

This is the only example where more exploit sinks were generated from the root cause as the taint source than the program input as the taint source. Both taint sources produced only tainted read exploit sinks, and these detectors were only triggered in code related to the `memcpy` or `memmove` functions.

Using exclusively the tainted read detector (D_7) we found 28 unique locations in OpenSSL that read the data derived from the root cause (see Table 5.2). Furthermore the detectors at these locations report the TCN of the tainted bytes to be 0 which indicates that little computation is done on these copied bytes. Therefore, we have no further indication that Heartbleed is exploitable as remote code execution, but the detectors reveal other locations in the code that move around this data. This could be useful to an analyst to figure out alternative ways to exfiltrate data from a read disclosure. But for the purpose of this bug, these detectors do not reveal any new ways of exfiltrating the data from the buffer overread.

5.2.6 SQLite

SQLite is an embedded SQL database engine that is used in a variety of applications [18]. The library is used in a variety of applications like Adobe's Photoshop Lightroom product, the flight software for Airbus, and the client side data store for Dropbox's file archiving and synchronization service.

When we run our program on the crashing input under Valgrind, we get the report that there is a use-after-free at `sqlite3.c:127739` in the function `exprAnalyze`. After manual analysis using the method described in 5.2.2 we calculate how many times we hit the `sqlite3MemFree` function before the program frees the target chunk. We assign taint labels to this chunk after the tenth call to `sqlite3MemFree` in `sqlite3.c:20510`. Figure 5.5 shows the result of our instrumentation in the `sqlite3MemFree` function.

```
2  static void sqlite3MemFree(void *pPrior){
     . . .
4    sqlite3_int64 *p = (sqlite3_int64*)pPrior;
     assert( pPrior!=0 );
6    p--;
     SQLITE_FREE(p);
8    // root cause tainting after 10th free
     vm_taint_exploitation_label(p, 776, 9);
10
   }
```

Figure 5.5: Root Cause Taint Point for a use-after-free in sqlite3.c

As expected, the tool detects the usage of tainted data at `sqlite3.c:127739` with the tainted read detector (D_7). This is the only category of detectors that is triggered besides a read of tainted malloc metadata (D_4), but similar to the lesson learned in 5.2.2, we rule this out as a useful exploit detector because the allocator is only reading valid pointers in the free list. We determined that the use-after-free is of a field in the variable `pTerm` (of type `struct *WhereTerm`) called `prereqRight` which is an 8 byte quantity of type `SQLITE_BITMASK_TYPE`. Given that `pTerm` is used after it has been freed, a potential analyst may try to change control flow to provoke uses of other fields in the `pTerm`, but the program trace reveals that this one location on line 127739 is the last use of `pTerm`. In other words, there is no indirect control flow decision made as a result of any of the fields in `pTerm`.

Root cause tainting leads to only five unique exploit sinks: two of them are related to controlling the `pTerm->prereqRight` data and the other three are related to the reading of freelist pointers. Conversely, tainting from the program input leads to 776 exploit detectors (see Table 5.2).

5.2.7 Woff2

WOFF2 is a format for packaging and compressing font data, with one of the primary goals being to access various fonts in Web documents through the `CSS @font-face` rules [20].

We are given a crashing input that triggers an *n*-byte heap buffer overwrite [13]. Figure 5.6 shows the code for reconstructing the `glyf` table, which is a table for "defining the appearance of glyphs in font" [12]. Glyphs require specifications of the contours of the character and the "instructions that grid-fit that glyph" [12]. On line 7, the program begins a loop over an array of glyphs. On line 10, the program reads `instruction_size`, but it does not perform any checks on its value. On lines 17 and 22, `glyph_buf` receives data from `bbox_stream` and `composite_stream` respectively. Last, line 29 (which translates to line 500 in the actual source code) contains the `Read` that triggers the heap buffer overflow, reading `instruction_size` number of bytes.

Figure 5.7 shows the location of the vulnerable `memcpy` along with the RTCP instrumentation. The bug is a buffer overwrite in the `memcpy` function (`buffer.h:89`) in the `Buffer` class which is called from `woff2_dec.cc:500` in the function `woff2::ReconstructGlyf`. This test case is the only example in our corpus where the program crashes when it is not under any memory instrumentation. There are 992,647 instructions executed between the root cause and

```
bool ReconstructGlyf(const uint8_t* data, Table* glyf_table,
                     uint32_t* glyf_checksum, Table * loca_table,
                     uint32_t* loca_checksum, WOFF2FontInfo* info,
                     WOFF2Out* out) {
  . . .
  info->x_mins.resize(info->num_glyphs);
  for (unsigned int i = 0; i < info->num_glyphs; ++i) {
      . . .
      if (have_instructions) {
        if (PREDICT_FALSE(!Read255UShort(&glyph_stream, &instruction_size))) {
          return FONT_COMPRESSION_FAILURE();
        }
      }

      . . .
      glyph_size = Store16(glyph_buf.get(), glyph_size, n_contours);
      if (PREDICT_FALSE(!bbox_stream.Read(glyph_buf.get() + glyph_size, 8))) {
        return FONT_COMPRESSION_FAILURE();
      }
      glyph_size += 8;

      if (PREDICT_FALSE(!composite_stream.Read(glyph_buf.get() + glyph_size,
            composite_size))) {
        return FONT_COMPRESSION_FAILURE();
      }
      glyph_size += composite_size;
      if (have_instructions) {
        glyph_size = Store16(glyph_buf.get(), glyph_size, instruction_size);
        if (PREDICT_FALSE(!instruction_stream.Read(glyph_buf.get() + glyph_size,
              instruction_size))) {
          return FONT_COMPRESSION_FAILURE();
        }
        glyph_size += instruction_size;
      }
      . . .
  }
  . . .
}
```

Figure 5.6: Demonstration of the buffer overread for the woff2 bug in `woff2_dec.cc`

the malloc consistency check that eventually causes the program to abort. This is notable because it provides plenty of distance between the root cause and the crash in order to aggregate a large number of potential exploit sinks.

```
class Buffer {
 public:
  Buffer(const uint8_t *buffer, size_t len)
      : buffer_(buffer),
        length_(len),
        offset_(0) { }

  bool Skip(size_t n_bytes) {
    return Read(NULL, n_bytes);
  }

  bool Read(uint8_t *buffer, size_t n_bytes) {
    if (n_bytes > 1024 * 1024 * 1024) {
      return FONT_COMPRESSION_FAILURE();
    }
    if ((offset_ + n_bytes > length_) ||
        (offset_ > length_ - n_bytes)) {
      return FONT_COMPRESSION_FAILURE();
    }
    if (buffer) {
      std::memcpy(buffer, buffer_ + offset_, n_bytes);
      // root cause taint labeling
      vm_taint_exploitation_label(buffer + 0x3296, 10, 392);
    }
    offset_ += n_bytes;
    return true;
  }
```

Figure 5.7: Root Cause Taint instrumentation for woff2 in file woff2-buffer.h

Examining the exploit sinks for the root cause-based tainting, we see that 180 of our 276 detectors (non-unique) are a tainted read (D_7) of the same stack address and triggered at `woff2_dec.cc:603`, the following line:

```
*glyf_checksum += ComputeULongSum(glyph_buf.get(), glyph_size);
```

where `glyph_buf` is a `std::unique_ptr` to the malloc chunk that gets overflown.

The tainted value is the value pointed to by `glyf_checksum` which is a stack variable from the `woff2::ReconstructFont` function which calls `woff2::ReconstructGlyf`. It is obvious that when `glyf_checksum` is tainted each time the code on line 603 is executed, the TCN of the computed value continues to increase. A TCN as high as 53 is observed as a result of this checksum operation. A learning point here is that the tainted read detector can give superfluous information. Particularly, code to compute checksums propagates taint, but not in a reversible way, so they are not typically useful for exploit development. However, *TEASER* handles

these superfluous detectors by providing the TCN which can serve as an effective way to discount some exploit detectors.

We observe 26 versus 4,430 unique exploit sinks when we use root cause-based taint labeling instead of input taint labeling (see Table 5.2). Just like the other test cases, this smaller number of exploit sinks allows an analyst to manually go through the exploit sinks in order to better understand how to exploit the program.

5.3 Summary and Discussion of Results

We ran *TEASER* on six programs with known bugs. We were able to reaffirm the exploit potential of the bugs in c-ares and OpenSSL. For c-ares, our results confirmed that a one byte overflow of a constant value (\x01) can affect the execution of the heap allocator (D_4). For OpenSSL we identified the location of our information disclosure by examining the callstack at one of our tainted read detectors (D_7).

The exploitability of the rest of our bugs was harder to determine. But, in each case, we were usually able to distill the effect of the memory corruption to several lines in the source code. We would not have been able to do this if we had tainted from the program input. Table 5.2 shows the large difference in the number of exploit sinks for file input versus root cause taint for each program (excluding OpenSSL).

5.3.1 Unexploitable Uses of Memory Corrupted Data

There are instances where a buffer overwrite can happen but conditions in the program, particularly locations of the allocation of heap chunks, prevent the corruption of heap metadata or adjacent allocated chunks. Consider Figure 5.8.

```
. . .
char *p;
p = malloc(40);
// get data clearly more than space allocated for p
fgets(p, 50, STDIN);
foo(p);
. . .
```

Figure 5.8: Demonstration of malloc overflow whose exploitability is shaped by the environment

Although this code above is clearly a heap buffer overwrite its exploitability depends on the program code executed before and after it.

Particularly, there are two heap layout conditions that the attacker should consider in order to turn a heap bug into a reliable heap exploit:

1. Data - sizes and contents of heap chunks.

2. Control - the sequence of calls to malloc and free before and after the root cause

While it is not possible to have complete control over these two principles, they serve as a guide for manipulating the heap environment. Figure 1.1, our motivating example, shows almost arbitrary control over control and data, since an arbitrary sequence of mallocs and frees can be constructed. Additionally, two types of objects, a string object and a printer object, can be created. On the other hand, more ambiguous cases like Figure 5.8 do not clearly have the ability to shape the heap.

5.3.2 Simulating the Heap Environment and the Realism of the Fuzzing Target Harness

All of our programs are each one target function that demonstrates usage of a library's API. We create a program that runs each of these target functions on input from a file representing our crashing input. We call this our crash harness. The crash harness can be used to simulate a heap environment before the target function is called in order to provoke a heap overflow to corrupt a certain chunk.

Our test case with c-ares shows the importance of simulating the heap to reveal the exploitability of the heap bug. At first, our detectors were not catching any use of the one byte overflow. We determined that lack of detectors being triggered was due to no adjacent malloc chunks to the chunk with the overflow.

We were able to utilize our crash harness to create a sequence of malloc and frees before a call to our target function. This is useful, because we were able to guarantee that the result of `ares_create_query` was allocated directly before an in-use chunk from our crash harness. The detectors demonstrated that this chunk's metadata was now under the control of the attacker through this one byte overflow.

Chapter 6

Conclusion

We created a model for exploring the effects of a memory corruption on a program's execution for the purpose of assessing the exploitability of a bug. Our Root Cause Taint Model allows us to label data from a memory corruption and we use seven Exploit Sink types to show how the tainted data can be leveraged for an exploit. We implement this model in a tool called *TEASER*. We tested our model and measurements on six real-world vulnerabilities from the Google Fuzz Test corpus [10]. Our tool confirmed the exploitability of two of our six test cases. For the rest of the cases, the tool demonstrated the lack of the exploitability of a program or possible exploit paths given some change to the input.

6.1 Future Work

First, we will make *TEASER* more usable. We can automate the labeling of exploit sources. Our approach for automating exploit source labeling would be to create a memory error detector in PANDA that automatically taints bytes involved in memory corruptions. Another form of usability would be to extend *TEASER* to work on stripped binaries. Finally, we will integrate the results of *TEASER* with a popular disassembler.

Second, we will improve the testing and evaluation of *TEASER*. We will use a bigger data set in order to quantitatively show that *TEASER* produces less exploit sinks than the alternative of labeling the program input. We would also like to use *TEASER* on smaller programs (i.e., binaries from Capture the Flag competitions) in order to demonstrate *TEASER*'s ability to aid in creating remote code execution POCs.

Bibliography

[1] Analyze crashes to find security vulnerabilities in your apps. https://blogs.msdn.microsoft.com/msdnmagazine/2007/10/22/analyze-crashes-to-find-security-vulnerabilities-in-your-apps/. Accessed: 2017-04-13.

[2] Boringssl. https://boringssl.googlesource.com/boringssl/. Accessed: 2017-05-07.

[3] c-ares. https://c-ares.haxx.se/. Accessed: 2017-03-29.

[4] Cert triage tools. http://www.cert.org/vulnerability-analysis/tools/triage.cfm. Accessed: 2017-05-01.

[5] Chrome os exploit: c-ares oob write + dump_vpd_log ¿ symlink. https://bugs.chromium.org/p/chromium/issues/detail?id=648971. Accessed: 2017-03-29.

[6] Cve-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. Accessed: 2017-03-28.

[7] Cve-2015-8317. https://access.redhat.com/security/cve/cve-2015-8317. Accessed: 2017-03-30.

[8] Cwe-200: Information exposure. https://cwe.mitre.org/data/definitions/200.html. Accessed: 2017-05-15.

[9] !exploitable. https://msecdbg.codeplex.com/. Accessed: 2017-04-13.

[10] fuzzer-test-suite. https://github.com/google/fuzzer-test-suite. Accessed: 2017-03-30.

[11] Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/.

[12] The 'glyf' table. https://developer.apple.com/fonts/TrueType-Reference-Manual/RM06/Chap6glyf.html. Accessed: 2017-05-14.

[13] Heap-buffer-overflow in read. https://bugs.chromium.org/p/chromium/issues/detail?id=609042. Accessed: 2017-04-13.

[14] Heap-use-after-free in asn1_string_free. https://bugs.chromium.org/p/chromium/issues/detail?id=586798. Accessed: 2017-03-30.

[15] Heartbleed. http://heartbleed.com/. Accessed: 2017-03-28.

[16] "mayhem" declared preliminary winner of historic cyber grand challenge. http://www.darpa.mil/news-events/2016-08-04. Accessed: 2017-05-01.

[17] Platform for architecture-neutral dynamic analysis. https://www.github.com/panda-re/panda. Accessed: 2017-05-19.

[18] Sqlite. https://www.sqlite.org/. Accessed: 2017-05-07.

[19] sqlite3: Heap-use-after-free in expranalyze. https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=199. Accessed: 2017-03-30.

[20] Woff file format 2.0. https://www.w3.org/TR/WOFF2/. Accessed: 2017-03-29.

[21] The xml c parser and toolkit of gnome: libxml. http://xmlsoft.org/. Accessed: 2017-05-07.

[22] Malloc internals. https://sourceware.org/glibc/wiki/MallocInternals, 2016.

[23] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Commun. ACM*, 57(2):74–84, February 2014.

[24] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[25] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[26] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[27] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[28] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. Retracer: triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, pages 820–831. ACM, 2016.

[29] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[30] Brendan F Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering for the greater good with panda.

[31] Chris Evans. What is a "good" memory corruption vulnerability. `https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html`. Accessed: 2017-04-14.

[32] Justin Ferguson. http://www.blackhat.com/presentations/bh-usa-07/ferguson/whitepaper/bh-usa-07-ferguson-wp.pdf. `http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf`. Accessed: 2017-05-08.

[33] Sean Heelan. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. 2009.

[34] Sean Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, 2011.

[35] Charlie Miller, Juan Caballero, Noah Johnson, Min Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Crash analysis with bitblaze. `https://media.blackhat.com/bh-us-10/whitepapers/Miller/BlackHat-USA-2010-CMiller-Bitblaze-wp.pdf`. Accessed: 2017-04-13.

[36] Matt Miller. Modeling the exploitation and mitigation of memory safety vulnerabilities. `http://2012.ruxconbreakpoint.com/assets/Uploads/bpx/Modeling%20the%20exploitation%20and%20mitigation%20of%20memory%20safety%20vulnerabilities.pptx`. Accessed: 2017-04-13.

[37] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[38] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[39] Phantasmal Phantasmagoria. The malloc maleficarum, glibc malloc exploitation techniques. `http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt`, 2005.

[40] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, pages 25–41, 2011.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[42] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[43] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-independent dynamic information flow tracking. In *International Conference on Compiler Construction*, pages 144–163. Springer, 2013.

Appendix A

Motivating Example

```
#include <assert.h>

#define USAGE "./motivating-example <input>\n"
void exit_usage(int rc) {
    printf(USAGE); exit(rc);
}
void exit_error(const char *error) {
    printf("%s\n", error); exit(1);
}
#define DEBUG 1
void debug_printf(const char *fmt, ...)
{
    if (DEBUG) {
        va_list args;
        va_start(args, fmt);
        vfprintf(stderr, fmt, args);
        va_end(args);
    }

}

typedef enum {
    RAW_BYTES = 0x1eefbeef,
    PRINTER = 0x13371337
} obj_type ;

typedef struct printer {
    long id;
    void (*read_disclosure)(long *);
    long *data_ptr;
    obj_type type;
} printer;

typedef struct chunk {
    char *pointer;
    size_t size;
} chunk;

// arbitrary read 8 bytes from pointer
// stored at data
void word_printer(long *data) {
    printf("here is your word: %lx\n", *data);
}

long global_data = 1;
long printer_counter = 0;
chunk chunk_list[100];


int main_loop(FILE *fp) {
  char tmp[16];                 // 4 or 2 words
  char *line = NULL;            // 1 word
  size_t linecap = 0;           // 1 word
  ssize_t linelen;              // 1 word
  int id1, id2, ret, i, offset; // 5 words or 2.5 words
  ssize_t byte_stream_size;     // 1 word
  printer *printer_tmp;         // 1 word

  while ((linelen = getline(&line, &linecap, fp)) > 0) {
    debug_printf("Line of len %zd\n", linelen);
    if (linelen <= 1) {
      debug_printf("Empty line. Skipping . . .\n");
      continue;
```

```
    }
    line[linelen-1] = '\0';
    switch (line[0]) {
        // create_printer_object -> id
        case 'c':
            debug_printf("c [printer_object]\n");

            for (i = 0; i < 100; i++) {
                // chunk_list is initialized to 0 at the beginning of the
                // program
                if (chunk_list[i].pointer == NULL) {
                    printer_tmp = (printer *) malloc(sizeof(printer));
                    printer_tmp->id = printer_counter++;
                    printer_tmp->read_disclosure = word_printer;
                    printer_tmp->data_ptr = &global_data;
                    printer_tmp->type = PRINTER;
                    chunk_list[i].pointer = (char *) printer_tmp;
                    chunk_list[i].size = sizeof(printer);
                    debug_printf("id %d\n", i);
                    break;
                }
            }
            break;
        // create_raw_byte_array len raw_bytes -> id
        case 'b':
            ret = sscanf(&line[1], " %zd %n", &byte_stream_size, &offset);
            if (ret != 1) {
                debug_printf("Unsuccessful parse of [create_raw_byte_array]\n");
                continue;
            }
            if (byte_stream_size <= 0) {
                debug_printf("byte_stream_size [%zd] is <=0\n", byte_stream_size);
                continue;
            }
            if (byte_stream_size > (linelen - offset)) {
                debug_printf("byte_stream_size [%zd] is <=0\n", byte_stream_size);
                continue;
            }

            for (i = 0; i < 100; i++) {
                if (chunk_list[i].pointer == NULL) {
                    chunk_list[i].pointer = (char *) malloc(byte_stream_size);
                    memcpy(chunk_list[i].pointer, &line[1+offset], byte_stream_size);
                    chunk_list[i].size = byte_stream_size;

                    debug_printf("b [byte_array] of {");
                    for (i = 0; i < byte_stream_size; i++) {
                        debug_printf("%02x ", line[1+offset+i]);
                    }
                    debug_printf("}\n");
                    debug_printf("id %d\n", i);
                    break;
                }
            }
            break;
        // release id
        // VULN: leads to a UAF
        case 'r':
            ret = sscanf(&line[1], "%d", &id1);
            if (ret != 1) {
                debug_printf("Unsuccessful parse of [release]\n");
                continue;
            }
            if (id1 < 0 || id1 > 100) {
                continue;
            }
            debug_printf("r [release %d]\n", id1);

            if (chunk_list[id1].pointer != NULL) {
                free(chunk_list[id1].pointer);
            }
            break;
        // print id
        case 'p':
            ret = sscanf(&line[1], "%d", &id1);
            if (ret != 1) {
                debug_printf("Unsuccessful parse of [print]\n");
                continue;
            }
            if (id1 < 0 || id1 > 100) {
                continue;
            }
            debug_printf("p [print %d]\n", id1);

            if (chunk_list[id1].pointer != NULL) {
                printer_tmp = (printer *) chunk_list[id1].pointer;
                if (chunk_list[id1].size >= sizeof(printer)) {
                    if (printer_tmp->type == PRINTER) {
                        printer_tmp->read_disclosure(printer_tmp->data_ptr);
                    }
                }
            }
            break;
        // move dst_id src_id
        // VULN: leads to a heap buffer overflow
```

```
        case 'm':
            ret = sscanf(&line[1], "%d %d", &id1, &id2);
            if (ret != 2) {
                debug_printf("Unsuccessful parse of [move]\n");
                continue;
            }
            if (id1 < 0 || id1 > 100 || id2 < 0 || id2 > 100) {
                continue;
            }
            debug_printf("m [move %d %d]\n", id1, id2);

            memcpy(chunk_list[id1].pointer, chunk_list[id2].pointer, chunk_list[id2].size);
            break;
        default:
            debug_printf("default case\n");
    }
  }
  return 0;
}

int main(int argc, char *argv[]) {
  char *fname;
  if (argc == 1) {
    /*fname = (char *) "tmp";*/
    exit_usage(0);
  } else if (argc == 2) {
    fname = argv[1];
  } else {
    exit_usage(0);
  }
  FILE *fp = fopen(fname, "rb");
  if (fp == NULL)
      exit_usage(1);

  return main_loop(fp);
}
```