Crash analysis with BitBlaze

Charlie Miller
Independent Security Evaluators
http://securityevaluators.com

Juan Caballero (CMU and UC Berkeley)
Noah M. Johnson (UC Berkeley)
Min Gyung Kang (CMU and UC Berkeley)
Stephen McCamant (UC Berkeley)
Pongsin Poosankam (CMU and UC Berkeley)
Dawn Song (UC Berkeley)

July 26, 2010

Introduction

The underlying problem behind many security vulnerabilities of today is memory corruption bugs. There are a variety of different techniques available to try to find such vulnerabilities. One of the most common techniques is fuzzing. Of the different approaches to fuzzing, one of the simplest and also most successful is mutation based fuzzing, which amounts to making random changes to an input and testing it within the application. One of the major limitations with this type of approach is not that it doesn't find problems, but rather it can find too many problems. The limiting factor in finding and exploiting (or fixing) security vulnerabilities in this case is not finding the vulnerabilities, but rather prioritizing the ones found and determining their root cause. This is a problem for two sets of people. One group is security researchers and exploit developers who don't know which crashes to look at and have the time consuming task of trying to determine the root cause of the problem from an invalid input and a crash. The other group are developers who are also interested in the underlying cause of the crash, but instead of trying to exploit it, they are trying to fix it. Understanding the cause of the vulnerability is important to fixing it correctly, otherwise the fix may correct one aspect of the vulnerability or code path to the bug, but not all code paths. For developers, even with source code in hand, this is not always an easy task.

This whitepaper attempts to alleviate this problem by introducing a solution using BitBlaze, a binary analysis tool. This toolset can help to quickly determine whether a particular crash, found by mutation based fuzzing, is exploitable and also to help determine the underlying cause of the crash.

One of the first researcher's to point out the problem of finding *too many* crashes was Ben Nagy in his Syscan 2009 talk, "Finding Microsoft Vulnerabilities by Fuzzing Binary Files with Ruby - A New Fuzzing Framework" [Nagy]. He fuzzed Microsoft Word using a variety of different fuzzing techniques. He found approximately 200,000 crashes which he categorized into 61 distinct bins. Later, he would find that of these 61 bins of crashes, 4 represented critical security vulnerabilities. However, he did not have the resources to examine all 61 distinct crashes, as this would take too much time. He had to give the crashes to Microsoft to help him analyze them.

One of us (Charlie) found a similar problem and discussed it in his CanSecWest 2010 talk, "Babysitting an Army of Monkeys - An analysis of fuzzing 4 products with 5 lines of Python". When fuzzing the PDF format in Preview, the default viewer for Mac OS X, he found crashes at 1373 different instruction pointers. Of these, somewhere between 220-280 were unique crashes and automatic tools from Apple identified over 60 as exploitable. If you consider many complicated vulnerabilities can take at least a day and sometimes a week to diagnose, it is a daunting task to try to analyze that many crashes. Likewise, he found around 30-40 unique crashes when fuzzing Adobe Reader. Some of these crashes are highlighted later in this document as examples of how to use BitBlaze to evaluate crashes. This paper also uses some OpenOffice crashes he found and documented in that presentation. In OpenOffice, he found somewhere around 70 distinct crashes.

Currently there are a variety of tools and utilities to help sort and prioritize crashes. These include tools such as crash.exe [filefuzz], !exploitable [!exploitable], and crashwrangler [crashwrangler]. While these tools seem to do a fairly good job sorting crashes into bins, their ability to evaluate which crashes are exploitable is limited. For example, in the crashes mentioned in the previous paragraph, !exploitable rated more than half as “unknown” with regards to exploitability. In other words, more than half the time the tools provides no additional information.

As for actually taking a crash and determining the root cause of the underlying vulnerability, researchers have a variety of tools such as debuggers, disassemblers, memory dumpers, etc. BlackHat USA this year presents some other work in this area by other researchers as well. Still, with existing tools, this can be a long and frustrating process.

In the next section, we'll introduce BitBlaze, a generic binary analysis platform which can be applied to this problem. As you'll see in the examples later in the paper, it can handle real applications and real bugs found with actual fuzzing runs, including all their inherent complexities. BitBlaze offers a variety of functionalities to a researcher. For example, it can take a taint-enhanced trace of an execution leading to a program crash. From that taint-enhanced execution trace, a researcher can perform offline data flow analysis using the tools BitBlaze provides. For data flowing in the forward direction in time, the researcher can visualize the taint information in the execution trace. For backwards in time, the researcher can slice data to see where it originated. By comparison, !exploitable only slices the current basic block and assumes all data is tainted. Using the tools that BitBlaze provides, a researcher can make a more informed decision about the crash. The drawback is that while !exploitable runs almost instantly, collecting an execution trace with BitBlaze can take much longer. However, once an execution trace has been captured, a variety of analyses can be performed in a repeatable way, without worrying about non-deterministic behavior such as different memory addresses being used or different thread interleaving altering the execution. For example, a researcher could use BitBlaze to also take an execution trace of a good run and compare the good execution with the one that caused the crash, using the trace alignment tool that BitBlaze provides, to identify points when execution differs. Such points are of interest as they may contribute to the cause of the crash.

The rest of this paper is outlined as follows. First, the BitBlaze platform will be introduced at length. Next, some general results of its use with regards to the Adobe Reader crashes from CanSecWest 2010 will be discussed. After that, detailed examples of using BitBlaze for particular crashes from Adobe Reader and Open Office will be given. These examples will show exactly how to use BitBlaze to perform meaningful analysis on actual crashes. Finally, some conclusions will be presented.

BitBlaze

The BitBlaze Binary Analysis Platform is a flexible infrastructure for analyzing off-the-shelf binaries[1][2]. Binary analysis is critical for both defensive and offensive security applications: we must often analyze either malicious software or commercial vulnerable software that lack source code. However, the complexity of binary analysis has limited the development of tools in this area. The BitBlaze platform provides a toolbox of components that can be used on their own or as building blocks in new analysis systems. In this section we explain the design of BitBlaze and its key components, and give examples of the variety of security applications it enables. We start with an overview of the BitBlaze architecture (Section 2.2), then discuss the components for static analysis (Section 2.3), dynamic analysis (Section 2.4), and symbolic exploration (Section 2.5). We then describe the tools for trace-based vulnerability analysis that are the used in this paper (Section 2.6), and some other applications of the BitBlaze framework (Section 2.7). The remainder of this paper, beginning with section 3, will show how BitBlaze can be applied to the problem of crash analysis.

Design and Architecture

In this section, we first describe the challenges of binary analysis for security applications, then the desired properties of a binary analysis platform catering to security applications, and finally outline the architecture of the BitBlaze Binary Analysis Platform.

Challenges

There are several main challenges for binary code analysis, some of which are specific to security applications.

Complexity. The first major challenge for binary analysis is that binary code is complex. Binary analysis needs to model this complexity accurately in order for the analysis itself to be accurate. However, the sheer number and complexity of instructions in modern architectures makes accurate modeling a significant challenge. Popular modern architectures typically have hundreds of different instructions, with new ones added at each processor revision. Further, each instruction can have complex semantics, such as single instruction loops, instructions which behave differently based upon their operand values, and implicit side effects such as setting processor flags. For example, the IA-32 manuals describing the semantics of x86 weigh over 11 pounds.

As an example, consider the problem of determining the control flow in the following x86 assembly program:

---

[1] This section includes some material from a previous paper [SBY+08].

[2] The BitBlaze project is led by Prof. Dawn Song. Other current and former members of the BitBlaze team who have contributed to the software described here include David Brumley, Juan Caballero, Cody Hartwig, Ivan Jager, Noah Johnson, Min Gyung Kang, Zhenkai Liang, Stephen McCamant, James Newsome, Pongsin Poosankam, Prateek Saxena, Heng Yin, and Jiang Zheng.

```
// instruction dst, src
add  a, b     // a = a+b
shl  a, x     // a << x
jz   target   // jump if zero to address target
```

The first instruction, add a,b, computes a := a+b. The second instruction, shl a,x, computes a := a << x. The last instruction, jz a, jumps to address a if the processor zero flag is set.

One problem is that both the add and shl instruction have implicit side effects. Both instructions calculate up to six other bits of information that are stored as processor status flags. In particular, they calculate whether the result is zero, the parity of the result, whether there is an unsigned or BCD carry, whether the result is signed, and whether an overflow has occurred.

Conditional control flow, such as the jz instruction, is determined by the implicitly calculated processor flags. Thus, either the add instruction calculates the zero flag, or the shl will. However, which instruction, add or shl, determines whether the branch is taken? Answering this question is not straight-forward. The shl instruction behaves *differently* depending upon the operands: it only updates the zero flag if x is not zero.

Lack of Higher-Level Semantics. The second major challenge is that binary code is different than source code, and in particular, lacks higher-level semantics present in source code.

- **No Functions**. The function abstraction does not exist at the binary level. Instead, control flow in a binary program is performed by jumps. For example, the x86 instruction call x is just shorthand for storing the current instruction pointer (eip) at the address named by the register esp, decrementing esp by the word size, then loading the eip with number x. Indeed, it is perfectly valid in assembly, and sometimes happens in practice, that code may call into the middle of a “function”, or have a single “function” separated into non-contiguous pieces.

- **Memory vs. Buffers**. Binary code does not have buffers, it has memory. While the OS may determine a particular memory page is not valid, memory does not have the semantics of a user-specified type and size. One implication of the difference between buffers and memory is that in binary code there is no inherent concept of a buffer overflow. While we may say a particular store violates a higher-level semantics given by the source code, such inferences require assumptions beyond the binary code itself.

- **No Types**. The only types available in machine language are those provided by the hardware: registers and memory. Even register types are not necessarily informative, since it is common to store values from one register type (e.g., 32-bit register) and read them as another (e.g., 8-bit register).

For these reasons, the prospect of performing analysis directly on machine instructions is daunting. Handling all of the possible instructions in a complex modern architecture is tedious and error-prone, and verifying that such an analysis is correct would be even more difficult. Further, an assembly-level approach is specific to a single architecture. All analysis would have to be ported each time we want to consider a new architecture. Thus, analysis could not take advantage of the common semantics across many different assembly languages.

Whole-System View. Many security applications require the ability to analyze operations in the operating system kernel and interactions between multiple processes. This requires a whole-system view, presenting greater challenges than in traditional single-program analysis.

Code Obfuscation. Some security applications require analyzing malicious code. Malicious code may employ anti-analysis techniques such as code packing, encryption, and obfuscation to make program analysis difficult, posing greater challenges than analyzing benign programs.

Design Rationale

The goal of the BitBlaze Binary Analysis Platform is to design and develop techniques and the core utilities that cater the common needs of security applications and enable others to build upon and develop new solutions to security problems more easily and effectively. Given the aforementioned challenges, we have a few design guidelines motivating the architecture of the BitBlaze Binary Analysis Platform:

Accuracy. We would like to enable accurate analysis, motivating us to build precise, formal models of instructions that allow the tool to accurately model the program execution behavior symbolically.

Extensibility. Given the complexity of binary analysis, we would like to develop core utilities which can then be re-used and easily extended to enable other more sophisticated analysis on binaries, or easily re-targeted to different architectures.

Fusion of Static and Dynamic Analysis. Static and dynamic analysis both have advantages and disadvantages. Static analysis can give more complete results as it covers different execution paths, however, it may be difficult due to the complexity of pointer aliasing, the prevalence of indirect jumps, and the lack of types and other higher-level abstractions in binaries. Even telling what is code and what is data statically is an undecidable problem in general. Moreover, it is particularly challenging for static analysis to deal with dynamically generated code and other anti-static-analysis techniques employed in malicious code. Furthermore, certain instructions such as kernel and floating point instructions may be extremely challenging to accurately model. On the other hand, dynamic analysis naturally avoids many of the difficulties that static analysis faces, at the cost of analyzing one path at a time. Thus, we would like to combine static and dynamic analysis whenever possible to have the benefits of both.

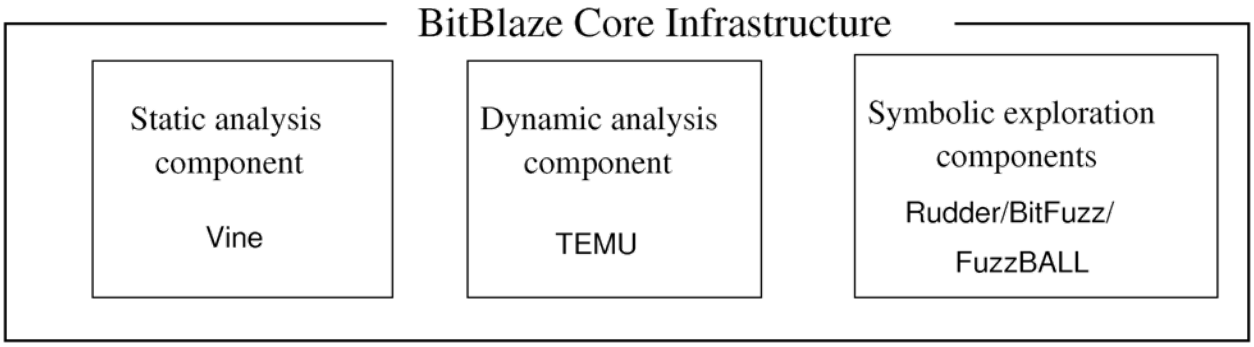

Figure 1: The BitBlaze Binary Analysis Platform Overview

Architecture

Motivated by the aforementioned challenges and design rationale, the BitBlaze Binary Analysis Platform is based on three core infrastructure components: Vine, the static analysis component, TEMU, the dynamic analysis component, and Rudder, BitFuzz, and FuzzBALL, tools for symbolic exploration that combine dynamic and static analysis, as shown in Figure 1.

Vine translates assembly instructions from a binary or trace into a simple, formally specified intermediate language (IL) and provides a set of core utilities for common analysis on the IL, such as control flow, data flow, optimization, symbolic execution, and weakest precondition calculation.

TEMU performs whole-system dynamic analysis, enabling whole-system fine-grained monitoring and dynamic binary instrumentation. It provides a set of core utilities for extracting OS-level semantics, user-defined dynamic taint analysis, and a clean plug-in interface for user-defined activities.

Rudder, BitFuzz, and FuzzBALL uses the core functionalities provided by Vine and TEMU to enable dynamic symbolic execution at the binary level. For a given program execution path, they identify the symbolic path predicates that symbolic inputs need to satisfy to follow the program path. By querying a decision procedure, they can determine whether the path is feasible and what inputs could lead the program execution to follow the given path. Thus, they can automatically generate inputs leading program execution down different paths, exploring different parts of the program execution space. The tools provide a set of core utilities and interfaces enabling users to control the exploration state and provide new path selection policies.

Vine: the Static Analysis Component

In this section, we give an overview of Vine, the static analysis component of the BitBlaze Binary Analysis Platform, describing its intermediate language (IL), its front end and back end components, and implementation.

Vine Overview

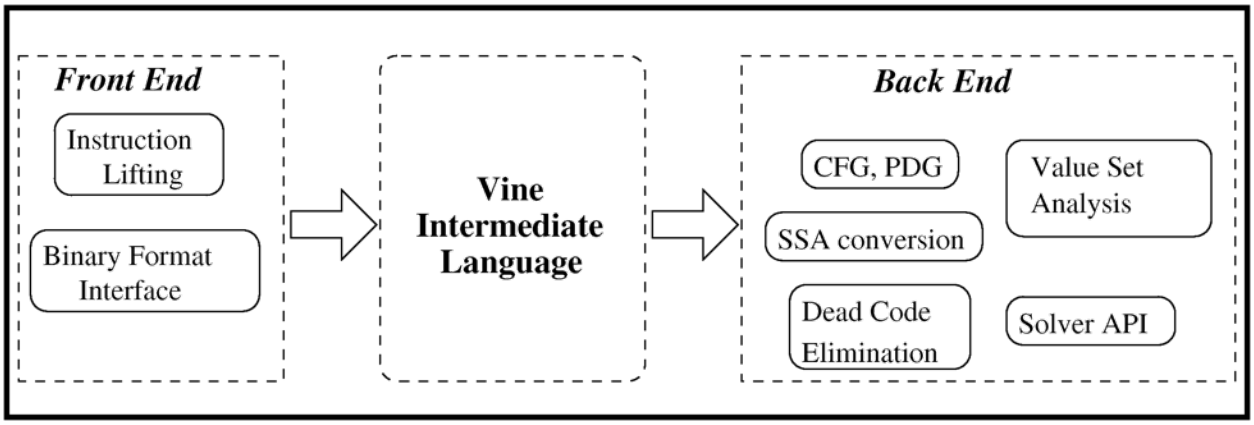

Figure 2: Vine Overview

Figure 2 shows a high-level picture of Vine. The Vine static analysis component is divided into a platform-specific front-end and a platform-independent back-end. At the core of Vine is a platform-independent intermediate language (IL) for assembly. The IL is designed as a small and formally specified language that faithfully represents the assembly languages. Assembly instructions in the underlying architecture are translated to the Vine IL via the Vine front-end. All back-end analyses are performed on the platform-independent IL. Thus, program analyses can be written in an architecture-independent fashion and do not need to directly deal with the complexity of an instruction set such as x86. This design also provides extensibility—users can easily write their own analysis on the IL by building on top of the core utilities provided in Vine. The Vine front-end currently supports translating x86 [Int08] and ARMv4 [ARM05] to the IL. It uses a set of third-party libraries to parse different binary formats and perform disassembly. The parsed instruction semantics are then translated into the Vine IL in a syntax-directed manner.

The Vine back-end supports a variety of core program analysis utilities. The back-end has utilities for creating a variety of different graphs, such as control flow and program dependence graphs. The back-end also provides an optimization framework. The optimization framework is usually used to simplify a specific set of instructions. We also provide program verification capabilities such as symbolic execution, calculating weakest preconditions, and interfacing with decision procedures.

To combine static and dynamic analysis, we also provide an interface for Vine to read an execution trace generated by a dynamic analysis component such as TEMU. The execution trace can be lifted to the IL for various further analysis.

The Vine Intermediate Language

$$\begin{array}{ll} \textit{program} & :: \\ & = \textit{decl* stmt*} \\ \textit{decl} & :: \\ & = \texttt{var}\ \textit{var;} \end{array}$$

$$
\begin{array}{lcl}
stmt & ::= & lval = exp\texttt{;} \mid \texttt{jmp(}exp\texttt{);} \mid \texttt{cjmp(}exp\texttt{, }exp\texttt{, } \\
 & & exp\texttt{);} \mid \texttt{halt(}exp\texttt{);} \mid \texttt{assert(}exp\texttt{);} \\
 & & \mid \texttt{label } label\texttt{:} \mid \texttt{special } string\texttt{;} \mid \texttt{\{ } decl^{*} \\
 & & stmt^{*}\texttt{\}} \\
label & ::= & identifier \\
lval & ::= & var \mid var\texttt{[}exp\texttt{]} \\
exp & ::= & \texttt{(} exp \texttt{)} \mid lval \mid \texttt{name(}label\texttt{)} \mid exp \diamond_b exp \mid \\
 & & \diamond_u exp \mid const \\
 & & \mid \texttt{let } lval = exp \texttt{ in } exp \mid \texttt{cast(}exp\texttt{)} \\
 & & cast\_kind\texttt{:}\tau_{\mathrm{reg}} \\
cast\_kind & ::= & \texttt{Unsigned} \mid \texttt{U} \mid \texttt{Signed} \mid \texttt{S} \mid \texttt{High} \mid \texttt{H} \mid \\
 & & \texttt{Low} \mid \texttt{L} \\
var & ::= & identifier\texttt{:}\tau \\
\diamond_b & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{/\$} \mid \texttt{\%} \mid \texttt{\%\$} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{@>>} \mid \texttt{\&} \mid \\
 & & \texttt{\^{}} \mid \texttt{||} \\
 & & \mid \texttt{==} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<\$} \mid \texttt{<=\$} \mid \texttt{>\$} \mid \\
 & & \texttt{>=\$} \\
\diamond_u & ::= & \texttt{-} \mid \texttt{!} \\
const & ::= & integer\texttt{:}\tau_{\mathrm{reg}} \\
\tau & ::= & \tau_{reg} \mid \tau_{mem} \\
\tau_{reg} & ::= & \texttt{reg1\_t} \mid \texttt{reg8\_t} \mid \texttt{reg16\_t} \mid \texttt{reg32\_t} \mid \\
 & & \texttt{reg64\_t} \\
\tau_{mem} & ::= & \texttt{mem32l\_t} \mid \texttt{mem64l\_t} \mid \tau_{reg}\texttt{[}const\texttt{]}
\end{array}
$$

Table 1: The grammar of the Vine Intermediate Language (IL).

The Vine IL is the target language during translation, as well as the analysis language for back-end program analysis. The semantics of the IL are designed to be faithful to assembly languages. Table 1 shows the syntax of Vine IL. The lexical syntax of identifiers, strings and comments are as in C. Integers may be specified in decimal, or in hexadecimal with a prefix of 0x.

The base types in the Vine IL are 1, 8, 16, 32, and 64-bit-wide bit vectors, also called registers. 1-bit registers are used as booleans; false and true are allowed as syntactic sugar for 0:reg1_t and 1:reg1_t respectively. There are also two kinds of aggregate types, which we call arrays and memories. Both are usually used to represent the memory of a machine, but at different abstraction levels. An array consists of distinct elements of a fixed register type, accessed at consecutive indices ranging from 0 up to one less than their declared size. By contrast, memory indices are always byte offsets,

but memories may be read or written with any type between 8 and 64 bits. Accesses larger than a byte use a sequence of consecutive bytes, so accesses at nearby addresses might partially overlap, and it is observable whether the memory is little-endian (storing the least significant byte at the lowest address) or big-endian (storing the most significant byte at the lowest address). Generally, memories more concisely represent the semantics of instructions, but arrays are easier to analyze, so Vine analyses will convert memories into arrays, a process called *normalization* that we discuss in more detail below.

Expressions in Vine are side-effect free. Variables and constants must be labeled with their type (separated with a colon) whenever they appear. The binary and unary operators are similar to those of C, with the following differences:

- Not-equal-to is <>, rather than !=.
- The division, modulus, right shift, and ordered comparison operators are explicitly marked for signedness: the unadorned versions are always unsigned, while the signed variants are suffixed with a $ (for “signed”), or in the case of right shift prefixed with an @ (for “arithmetic”).
- There is no distinction between logical and bitwise operators, so & also serves for &&, | also serves for ||, and ! also serves for ~.

There is no implicit conversion between types of different widths; instead, all conversions are through an explicit cast operator that specifies the target type. Widening casts are either Unsigned (zero-extending) or Signed (sign-extending), while narrowing casts can select either the High or Low portion of the larger value. (For brevity, these are usually abbreviated by their first letters.) A let expression, as in functional languages, allows the introduction of a temporary variable.

A program in Vine is a sequence of variable declarations, followed by a sequence of statements; block structure is supported with curly braces. (In fact, the parser allows declarations to be intermixed with statements, but the effect is as if the declarations had all appeared first.) We sometimes refer to statements as “instructions,” but note that more complex machine instructions translate into several Vine statements. The most frequent kind of statement is an assignment to a variable or to a location in an array or memory variable. Control flow is unstructured, as in assembly language: program locations are specified with labels, and there are unconditional (jmp) and conditional (cjmp) jumps. The argument to jmp and the second and third arguments to cjmp may be either labels (introduced by name), or a register expression to represent a computed jump. The first argument to cjmp is a reg1_t that selects the second (for 1) or third (for 0) argument as the target.

A program can halt normally at any time by issuing the halt statement. We also provide assert, which acts similar to a C assert: the asserted expression must be true, else the machine halts. A special in Vine corresponds to a call to an externally defined procedure or function. The argument of a special indexes what kind of special, e.g., what system call. The semantics of special is up to the analysis; its operational semantics are not

defined. We include special as an instruction type to explicitly distinguish when such calls may occur that alter the soundness of an analysis. A typical approach to dealing with special is to replace special with an analysis-specific summary written in the Vine IL that is appropriate for the analysis.

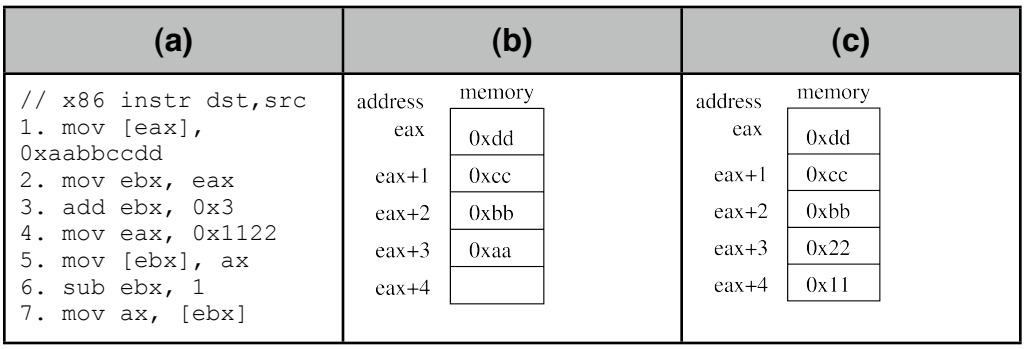

Figure 3: An example of little-endian stores as found in x86 that partially overlap. (b) shows memory after executing line 1, and (c) shows memory after executing line 5. Line 7 will load the value 0x22bb.

```
1. mem4 = let mem1 = store(mem0,eax, 0xdd, reg8_t) in
          let mem2 = store(mem1, eax+1, 0xcc, reg8_t) in
          let mem3 = store(mem2, eax+2, 0xbb, reg8_t) in
              store(mem3, eax+3, 0xcc, reg8_t);
...
5. mem6 = let mem5 = store(mem4, ebx, 0x22, reg8_t) in
            store(mem5, ebx+1, 0x22, reg8_t)
...
7. value = let b1 = load(mem6, ebx, reg8_t) in
           let b2 = load(mem6, ebx+1, reg8_t) in
           let b1' = cast(unsigned, b1, reg16_t) in
           let b2' = cast(unsigned, b2, reg16_t) in
              (b2' << 8) | b1';
```

Figure 4: Vine normalized version of the store and load from Figure 3(a).

Normalized Memory

The endianness of a machine is usually specified by the byte-ordering of the hardware. A little endian architecture puts the low-order byte first, and a big-endian architecture puts the high-order byte first. x86 is an example of a little endian architecture, and PowerPC is an example of a big endian architecture.

We must take endianness into account when analyzing memory accesses. Consider the assembly in Figure 3(a). The mov operation on line 2 writes 4 bytes to memory in little endian order (since x86 is little endian). After executing line 2, the address given by eax contains byte 0xdd, eax+1 contains byte 0xcc, and so on, as shown in Figure 3(b). Lines 2 and 3 set ebx = eax+2. Line 4 and 5 write the 16-bit value 0x1122 to ebx. An analysis of these few lines of code needs to consider that the write on line 4 overwrites the last byte written on line 1, as shown in Figure 3(c). Considering such cases requires

additional logic in each analysis. For example, the value loaded on line 7 will contain one byte from each of the two stores.

We say a memory is normalized for a b-byte addressable memory if all loads and stores are exactly b-bytes and b-byte aligned. For example, in x86 memory is byte addressable, so a normalized memory for x86 has all loads and stores at the byte level. The normalized form for the write on Line 1 of Figure 3(a) in Vine is shown in Figure 4. Note the subsequent load on line 7 are with respect to the current memory mem6. Normalized memory makes writing program analyses involving memory easier. Analyses are easier because normalized memory syntactically exposes memory updates that are otherwise implicitly defined by the endianness. The Vine back-end provides utilities for normalizing all memory operations.

The Vine Front-End

The Vine front-end is responsible for translating binary code to the Vine IL. In addition, the front-end interfaces with libraries such as the GNU Binary File Descriptor (libbfd) library for parsing the low-level details of binary files.

Translating binary code to the IL consists of three steps:

- **Step 1**. First the binary file is disassembled. Vine currently interfaces with two disassemblers: IDA Pro [Dat], a commercial disassembler, and our own linear-sweep disassembler built on top of GNU libopcodes. Interfacing with other disassemblers is straightforward.
- **Step 2**. The disassembly is passed to VEX, a third-party library which turns assembly instructions into the VEX intermediate language. The VEX IL is part of the Valgrind dynamic instrumentation tool [NS07]. The VEX IL is also similar to a RISC-based language. As a result, the lifted IL has only a few instruction types, similar to Vine. However, the VEX IL itself is inconvenient for performing program analysis because its information about side effects of instructions such as what EFLAGS are set by x86 instructions is implicit. This step is mainly performed in order to simplify the development of Vine: we let the existing tool take care of the task of reducing assembly instructions to a basic IL, then in step 3 expose all side-effects so that the analysis is faithful.
- **Step 3**. We translate the VEX IL to Vine. The resulting Vine IL is intended to be faithful to the semantics of the disassembled assembly instructions.

Translated assembly instructions have all side-effects explicitly exposed as Vine statements. As a result, a single typical assembly instruction will be translated as a sequence of Vine statements. For example, the add eax,0x2 x86 instruction is translated as the following Vine IL:

```
tmp1 = EAX;
EAX = EAX + 2;
//eflags calculation
CF = (EAX<tmp1);
tmp2 = cast(low, EAX, reg8_t);
PF = (!cast(low,
```

```
              ((((tmp2>>7)^(tmp2>>6))^((tmp2>>5)^(tmp2>>4)))^
              (((tmp2>>3)^(tmp2>>2))^((tmp2>>1)^tmp2)))), reg1_t);
AF = (16==(16&(EAX^(tmp1^2))));
ZF = (EAX==0);
SF = (1&(EAX>>31));
OF = (1&(((tmp1^(2^0xFFFFFFFF))&(tmp1^EAX))>>31));
```

The translation exposes all the side-effects of the add instruction, including all 6 eflags that are updated by the operation. As another example, an instruction with the rep prefix is translated as a sequence of statements that form a loop.

In addition to binary files, Vine can also translate an instruction trace to the IL. Conditional branches in a trace are lifted as assert statements to check that the executed branch is followed. This is done to prevent branching outside the trace to an unknown instruction. Vine and TEMU are co-designed so that TEMU generates traces in a trace format that Vine can read.

The Vine Back-End

In the Vine back-end, new program analyses are written over the Vine IL. Vine provides a library of common analyses and utilities which serve as building blocks for more advanced analyses. Below we provide an overview of some of the utilities and analyses provided in the Vine back-end.

Evaluator. Vine has an evaluator which implements the operational semantics of the Vine IL. The evaluator allows us to execute programs without recompiling the IL back down to assembly. For example, we can test a raised Vine IL for an instruction trace produced by an input by evaluating the IL on that input and verifying we end in the same state.

Graphs. Vine provides routines for building and manipulating control flow graphs (CFG), including a pretty-printer for the graphviz DOT graph language [gra]. Vine also provides utilities for building data dependence and program dependence graphs [Muc97].

One issue when constructing a CFG of an assembly program is determining the successors of jumps to computed values, called indirect jumps. Resolving indirect jumps usually requires program analyses that require a CFG, e.g., Value Set Analysis (VSA) [Bal07]. Thus, there is a potential circular dependency. Note that an indirect jump may potentially go anywhere, including the heap or code that has not been previously disassembled.

Our solution is to designate a special node as a successor of unresolved indirect jump targets in the CFG. We provide this so an analysis that depends on a correct CFG can recognize that we do not know the subsequent state. For example, a data-flow analysis could widen all facts to the lattice bottom. Most normal analyses will first run an indirect jump resolution analysis in order to build a more precise CFG that resolves indirect jumps to a list of possible jump targets. Vine provides one such analysis based on VSA [Bal07].

Single Static Assignment. Vine supports conversion to and from single static assignment (SSA) form [Muc97]. SSA form makes writing analysis easier because every variable is defined statically only once. We convert both memory and scalars to SSA form. We convert memories because then one can syntactically distinguish between memories before and after a write operation instead of requiring the analysis itself to maintain similar bookkeeping. For example, in the memory normalization example in Figure 3.2.1, an analysis can syntactically distinguish between the memory state before the write on line 1, the write on line 5, and the read on line 7.

Chopping. Given a source and sink node, a program chop [JR94] is a graph showing the statements that cause definitions of the source to affect uses of the sink. For example, chopping can be used to restrict subsequent analysis to only a portion of code relevant to a given source and sink instead of the whole program.

Data-flow and Optimizations. Vine provides a generic data-flow engine that works on user-defined lattices. Vine also implements several data-flow analysis. Vine currently implements Simpson's global value numbering [Sim96], constant propagation and folding [Muc97], dead-code elimination [Muc97], live-variable analysis [Muc97], integer range analysis, and Value set analysis (VSA) [Bal07]. VSA is a data-flow analysis that over-approximates the values for each variable at each program point. Value-set analysis can be used to help resolve indirect jumps. It can also be used as an alias analysis. Two memory accesses are potentially aliased if the intersection of their address value sets is non-empty.

Optimizations are useful for simplifying or speeding up subsequent analysis. For example, we have found that the time for the decision procedure STP to return a satisfying answer for a query can be cut in half by using program optimization to simplify the query first [BHL+08].

Program Verification Analyses. Vine currently supports formal program verification in two ways. First, Vine can convert the IL into Dijkstra's Guarded Command Language (GCL), and calculate the weakest precondition with respect to GCL programs [Dij76]. The weakest precondition for a program with respect to a predicate q is the most general condition such that any input satisfying the condition is guaranteed to terminate (normally) in a state satisfying q. Currently we only support acyclic programs, i.e., we do not support GCL while.

Vine also interfaces with decision procedures. Vine can write out expressions (including formulas such as weakest preconditions) in CVC Lite [cvc] or SMT-LIB syntax, which are supported by many decision procedures. In addition, Vine interfaces directly with the STP [GD07] decision procedure through calls from Vine to the STP library.

Implementation of Vine

The Vine infrastructure is implemented in C++ and OCaml. The front-end lifting is implemented primarily in C++, and consists of about 17,200 lines of code. The back-end

is implemented in OCaml, and consists of about 40,000 lines of code. We interface the C++ front-end with the OCaml back-end using OCaml via IDL-generated stubs. The front-end interfaces with Valgrind's VEX [Net04] to help lift instructions, GNU BFD for parsing executable objects, and GNU libopcodes for pretty-printing the disassembly. The implemented Vine IL has several constructors in addition to the instructions in Figure 1:

- The Vine IL has a constructor for comments. We use the comment constructor to pretty-print each disassembled instruction before the IL, as well as a place-holder for user-defined comments.
- The Vine IL supports variable scoping via blocks. Vine provides routines to de-scope Vine programs via α-varying as needed.
- The Vine IL has constructs for qualifying statements and types with user-defined attributes. This is added to help facilitate certain kinds of analysis such as taint-based analysis.

TEMU: the Dynamic Analysis Component

In this section, we give an overview of TEMU, the dynamic analysis component of BitBlaze Binary Analysis Platform, describing its components for extracting OS-level semantics, performing whole-system dynamic taint analysis, its plugins and implementation.

TEMU Overview

TEMU is a whole-system dynamic binary analysis platform that we developed as an extension of a whole-system emulator, QEMU [QEM]. We run an entire system, including the operating system and applications in this emulator, and observe in a fine-grained manner how the binary code of interest is executed. The whole-system approach in TEMU is motivated by several considerations:

- Many analyses require fine-grained instrumentation (i.e., at instruction level) on binary code. By dynamically translating the emulated code, the whole-system emulator enables fine-grained instrumentation.
- A whole-system emulator presents us a whole-system view. The whole-system view enables us to analyze the operating system kernel and interactions between multiple processes. In contrast, many other binary analysis tools (e.g., Valgrind, DynamoRIO, Pin) only provide a local view (i.e., a view of a single user-mode process). This is particularly important for analyzing malicious code, because many attacks involve multiple processes and kernel attacks such as rootkits have become increasingly popular.
- A whole-system emulator provides an excellent isolation between the analysis components and the code under analysis, to prevent the code under analysis from interfering with analysis results. This is particularly important if the analyzed code might be malicious.

The design of TEMU is motivated by several challenges and considerations:

- The whole-system emulator only provides us only the hardware-level view of the emulated system, whereas we need a software-level view to get meaningful analysis results. Therefore, we need a mechanism that can extract the OS-level semantics from the emulated system. For example, we need to know what process is currently running and what module an instruction comes from.
- In addition, many analyses require reasoning about how specific data depends on its data sources and how it propagates throughout the system. We enable this using whole-system dynamic taint analysis.
- We need to provide a well-designed programming interface (i.e., API) for users to implement their own plugins on TEMU to perform their customized analysis. Such an interface can hide unnecessary details from users and allow reuse of common functionality.

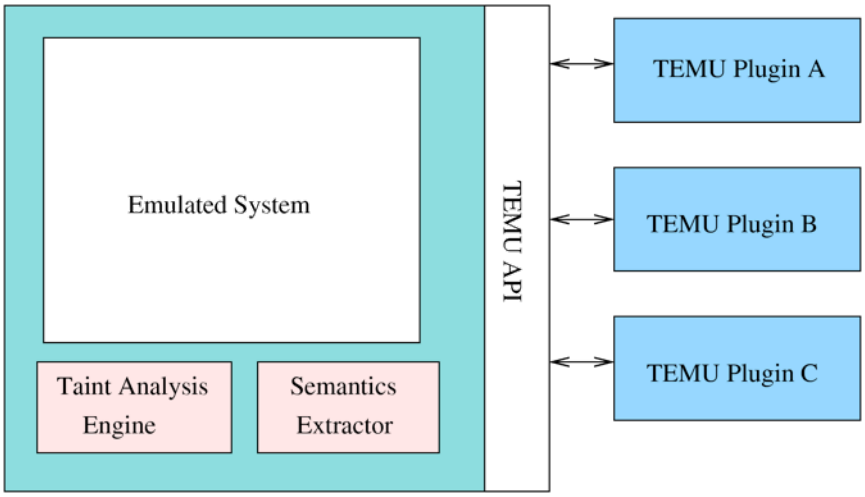

Figure 5: TEMU Overview

With these considerations in mind, we have designed the architecture of TEMU, as shown in Figure 5. We build the *semantics extractor* to extract OS-level semantics information from the emulated system. We build the *taint analysis engine* to perform dynamic taint analysis. We define and implement an interface (the TEMU API) for users to easily implement their own analysis modules (TEMU plugins). These modules can be loaded and unloaded at runtime to perform designated analyses. We implemented TEMU in Linux, and at the time of writing, TEMU can be used to analyze binary code in Windows 2000, Windows XP, and Linux systems. Below we describe these three components respectively.

Semantics Extractor
The semantics extractor is responsible for extracting OS-level semantics information of the emulated system, including process, module, thread, and symbol information.

Process and Module Information. For the current execution instruction, we need to know which process, thread and module this instruction comes from. In some cases, instructions may be dynamically generated and executed on the heap. Maintaining a mapping between addresses in memory and modules requires information from the

guest operating system. We use two different approaches to extract process and module information for Windows and Linux.

For Windows, we have developed a kernel module called module notifier. We load this module into the guest operating system to collect the updated memory map information. The module notifier registers two callback routines. The first callback routine is invoked whenever a process is created or deleted. The second callback routine is called whenever a new module is loaded and gathers the address range in the virtual memory that the new module occupies. In addition, the module notifier obtains the value of the CR3 register for each process. As the CR3 register contains the physical address of the page table of the current process, it is different (and unique) for each process. All the information described above is passed on to TEMU through a designated I/O port.
For Linux, we can directly read process and module information from outside, because we know the relevant kernel data structures, and the addresses of relevant symbols are also exported in the System.map file. In order to maintain the process and module information during execution, we hook several kernel functions, such as do_fork and do_exec.

Thread Information. For Windows, we also obtain the current thread information to support analysis of multi-threaded applications and the OS kernel. It is fairly straightforward, because the data structure of the current thread is mapped into a well-known virtual address in Windows.

Symbol Information. For PE (Windows) binaries, we also parse their PE headers and extract the exported symbol names and offsets. After we determine the locations of all modules, we can determine the absolute address of each symbol by adding the base address of the module and its offset. This feature is very useful, because all windows APIs and kernel APIs are exported by their hosting modules. The symbol information conveys important semantics information, because from a function name, we are able to determine what purpose this function is used for, what input arguments it takes, and what output arguments and return value it generates. Moreover, the symbol information makes it more convenient to hook a function—instead of giving the actual address of a function, we can specify its module name and function name. Then TEMU will automatically map the actual address of the function for the user.

Taint Analysis Engine
Our dynamic taint analysis is similar in spirit to a number of previous systems [CPG+04, NS05, CC04, SLZD04, CCC+05]. However, since our goal is to support a broad spectrum of different applications, our design and implementation is the most complete. For example, previous approaches either operate on a single process only [CCC+05, NS05, SLZD04], or they cannot deal with memory swapping and disks [CPG+04, CC04].

Shadow Memory. We use a shadow memory to store the taint status of each byte of the physical memory, CPU registers, the hard disk and the network interface buffer. Each tainted byte is associated with a small data structure storing the original source of

the taint and some other book keeping information that a TEMU plugin wants to maintain. The shadow memory is organized in a page-table-like structure to ensure efficient memory usage. By using shadow memory for the hard disks, the system can continue to track the tainted data that has been swapped out, and also track the tainted data that has been saved to a file and is then read back in.

Taint Sources. A TEMU plugin is responsible for introducing taint sources into the system. TEMU supports taint input from hardware, such as the keyboard, network interface, and hard disk. TEMU also supports tainting a high-level abstract data object (e.g. the output of a function call, or a data structure in a specific application or the OS kernel).

Taint Propagation. After a data source is tainted, the taint analysis engine monitors each CPU instruction and DMA operation that manipulates this data in order to determine how the taint propagates. The taint analysis engine propagates taint through data movement instructions, DMA operations, arithmetic operations, and table lookups. Considering that some instructions (e.g., xor eax, eax) always produce the same results, independent of the values of their operands, the taint analysis engine does not propagate taint in these instructions.

Note that TEMU plugins may employ very different taint policies, according to their application requirements. For example, for some applications, we do not need to propagate taint through table lookups. For some applications, we want to propagate taint through an immediate operand, if the code region occupied by it is tainted. Therefore, during taint propagation, the taint analysis engine lets TEMU plugins determine how they want to propagate taint into the destination.

This design provides valuable flexibility to TEMU plugins. They can specify different taint sources, maintain an arbitrary record for each tainted byte, keep track of multiple taint sources, and employ various taint policies.

TEMU API & Plugins

In order for users to make use of the functionalities provided by TEMU, we define a set of functions and callbacks. By using this interface, users can implement their own plugins and load them into TEMU at runtime to perform analysis. Currently, TEMU provides the following functionalities:

- Query and set the value of a memory cell or a CPU register.
- Query and set the taint information of memory or registers.
- Register a hook to a function at its entry and exit, and remove a hook. TEMU plugins can use this interface to monitor both user and kernel functions.
- Query OS-level semantics information, such as the current process, module, and thread.
- Save and load the emulated system state. This interface helps to switch between different machine states for more efficient analysis. For example, this interface can be used for multiple path exploration, because we can save a state for a specific branch

point and explore one path, and then load this state to explore the other path without restarting program execution from the beginning.

TEMU defines callbacks for various events, including (1) the entry and exit of a basic block; (2) the entry and exit of an instruction; (3) when taint is propagating; (4) when a memory location is read or written; (5) when a register is read or written; (6) hardware events such as network and disk inputs and outputs.
Quite a few TEMU plugins have been implemented using these functions and callbacks. These plugins include:

- Panorama [YSM+07]: a plugin that performs OS-aware whole-system taint analysis to detect and analyze malicious code's information processing behavior.
- HookFinder [YLS08]: a plugin that performs fine-grained impact analysis (a variant of taint analysis) to detect and analyze malware's hooking behavior.
- Renovo [KPY07]: a plugin that extracts unpacked code from packed executables.
- Polyglot [CYLS07a]: a plugin that make use of dynamic taint analysis to extract protocol message format.
- Tracecap: a plugin that records detailed logs about the program execution for offline analysis. The execution logs produced by Tracecap include an execution trace with intruction-level information, the state of a process at a some point in the execution, and information about the heap allocations requested by a process.
- MineSweeper [BHL+07]: a plugin that identifies and uncovers trigger-based behaviors in malware by performing online symbolic execution.
- BitScope: a more generic plugin that make use of symbolic execution to perform in-depth analysis of malware.
- HookScout: a plugin that infers kernel data structures.

Implementation of TEMU
The TEMU infrastructure is implemented in C and C++. In general, performance-critical code is implemented in C due to efficiency of C, whereas analysis-oriented code is written in C++ to leverage the abstract data types in the STL and stronger type checking in C++. For example, the taint analysis engine inserts code snippets into QEMU micro operations to check and propagate taint information. Since taint analysis is performance critical, we implemented it in C. On the other hand, we implemented the semantics extractor in C++ using string, list, map and other abstract data types in STL, to maintain a mapping between OS-level view and hardware view. The TEMU API is defined in C. This gives flexibility to users to implement their plugin in either C, C++, or both. The TEMU core consists of about 37,000 lines of code, excluding the code originally from QEMU (about 306,000 lines of code). The TEMU plugins we have built so far consist of about 134,000 lines of code, though a simple plugin can be implemented in only about 600 lines.

BitFuzz and FuzzBALL: Symbolic Exploration Components
Symbolic execution generalizes a single execution of a program by representing inputs as variables and performing operations on values dependent on them symbolically. This technique enables automated tools to reason about properties of all the program

executions that follow the same control flow path, and has been successfully applied to a wide range of applications in software engineering and security. In particular, we often use symbolic execution to explore different possible program executions, which we refer to as symbolic exploration for short.

One of the most important security applications of symbolic exploration is in fuzz testing, finding inputs that cause unusual behavior, potentially including security bugs. Many techniques are available for fuzz testing: the term was originally for supplying pure random bits as program input, and it is also common in practice to generate random legal inputs from a grammar, and/or make small random changes (mutations) to a legal input. These methods can be effective via a combination of brute force (automatically trying large numbers of inputs to look for an easily-detectable failure like a crash) and manual guidance (choosing a grammar or a suitable starting input); crashes found using these techniques are the main subject of this paper.

Symbolic execution can be used to make automatic fuzz testing smarter by making a more informed choice of inputs. Specifically, symbolic execution uses a program itself to determine which variations of an input would be interesting to explore: the basic intuition is that an input variation would be interesting to explore if it would cause the program to execute a new control-flow path. Symbolic execution can effectively discover inputs that trigger new control-flow paths by examining the branch conditions on an existing path, and using a decision procedure to find an input that would reverse a branch condition from true to false or vice-versa. The conditions on a path are large enough that it would usually be impractical to examine them by hand, but because they are only a small subset of the behavior of the entire program, they are still susceptible to efficient automated reasoning.

The most recent versions of the BitBlaze platform include two systems for automatic state-space exploration based on symbolic execution, called BitFuzz and FuzzBALL. (An older system, named Rudder, is described in previous papers [SBY+08].) Though they perform similar basic tasks, the two tools have complementary strengths and are suited for different classes of applications; we will describe them in turn.

BitFuzz: Trace-based Dynamic Symbolic Execution

BitFuzz builds on the two basic BitBlaze components, TEMU and Vine. TEMU is used to execute the program under test in its expected operating system environment (such as an unmodified version of Windows XP). The relevant inputs to the program are marked using a dynamic tainting analysis: they can come from the virtual keyboard, from a disk file, from a network message, or any specially designated API routine. A TEMU plugin observes the instructions that operate on the inputs, and saves them to a trace file that records the inputs and their arguments. Next, this trace file is parsed using the Vine toolkit and converted into an intermediate representation that captures the precise semantics of the original instructions using a small set of more general operations (so that the remaining processing can be independent of the complexities of a particular instruction set architecture). Finally, this representation of the trace is analyzed to extract the conditions leading to a particular branch as a logical formula, which can be

automatically solved using a decision procedure for satisfiability in a theory of arrays and bit vectors (precisely representing fixed-sized machine integers).

Beyond integrating these existing tools, the first new functionality that the BitFuzz platform provides might be called “closing the loop:” providing a further layer of automation to take candidate inputs produced by Vine and the decision procedure and present them as new inputs to the program under test. Rather than being limited to a simple back-and-forth alternation, BitFuzz maintains pools of candidate inputs and traces that have been collected but not yet processed. One input generates one trace, but one trace can generate many new candidate inputs, since it might be possible to reverse the direction of many branches. Thus any single starting input can lead to a large search tree of possible related program executions, and BitFuzz can explore from several starting points in parallel. The key technical challenge is how to prioritize the processing of traces and inputs to most effectively discover interesting program behaviors quickly. As an example of one heuristic used, BitFuzz maintains a record of which parts of a program have already been explored, and prefers trying to reverse branches that would lead to previously unexplored regions.

The other key functionality that BitFuzz provides is the ability to perform fuzzing in a distributed architecture by running instances of TEMU and Vine on many machines in a network and coordinating their operation. Again because the search space of possible program executions is so large, distributed operation is valuable in helping fuzzing to scale to larger applications. Distributing fuzzing is also a natural match because the structure of the fuzzing problem is easily parallelizable at the level of individual executions, and the analysis of individual executions is sufficiently independent that they can be performed with little intercommunication. However, it is important to choose an architecture so that the centralized coordination around the trace and input pools does not become a bottleneck.

BitFuzz is targeted at security-sensitive Windows applications, both malware and commercial-off-the-self (COTS) programs, and for scalability to large applications. A key advantage of its trace-based approach is that the program can run on a faithful emulator and interact with an unmodified operating system: no modifications to the execution semantics are required, ensuring that the program’s behavior is accurate. Further, separating trace collection from the symbolic execution and solving process provides a separation of concerns, and allows the two processes to easily proceed in parallel on distributed machines.

FuzzBALL: Online Symbolic Execution

Like BitFuzz, FuzzBALL also performs symbolic execution, but it takes a different approach. Rather than basing the symbolic analysis on an execution trace, FuzzBALL performs symbolic execution as an integral part of the execution of the program being analyzed. In essence FuzzBALL implements a symbolic interpreter that operates on a symbolic value in place of a concrete one. Registers and memory locations can all contain symbolic values, and the interpreter can perform operations on symbolic values without knowing a concrete value for them. Thus rather than basing its exploration of

the program's execution space on a seed concrete input, FuzzBALL can choose an arbitrary execution path (subject to feasibility). This is particularly valuable for executing code without access to concrete state: for instance, FuzzBALL can explore the execution of a single function in isolation, treating all of its memory accesses as symbolic.

FuzzBALL is based on Vine, but does not use TEMU, since its symbolic interpreter completely replaces the concrete execution of a program. In comparison to BitFuzz, it is particularly suited for exploration of API and incomplete program fragments, as well as standalone programs such as operating system kernels. It also includes a simulation of the Linux system call interface for running Linux/x86 executables.

Trace-based Vulnerability Analysis

In this section, we discuss the tools that we will use most heavily in the case studies of this paper, those for analyzing execution traces of vulnerable applications. We use a TEMU plugin, called Tracecap, to collect execution traces that include tainting information; then the tools of this section can be used to read and analyze those traces. In particular, we will cover tools for reading traces, for backward slicing of traces, for aligning pairs of traces, for integrating taint information with crash dumps, for tracing the allocations a program performs, and for measuring the quantitative influence and value sets for tainted values. An overview of how these components work together is shown in Figure 6.

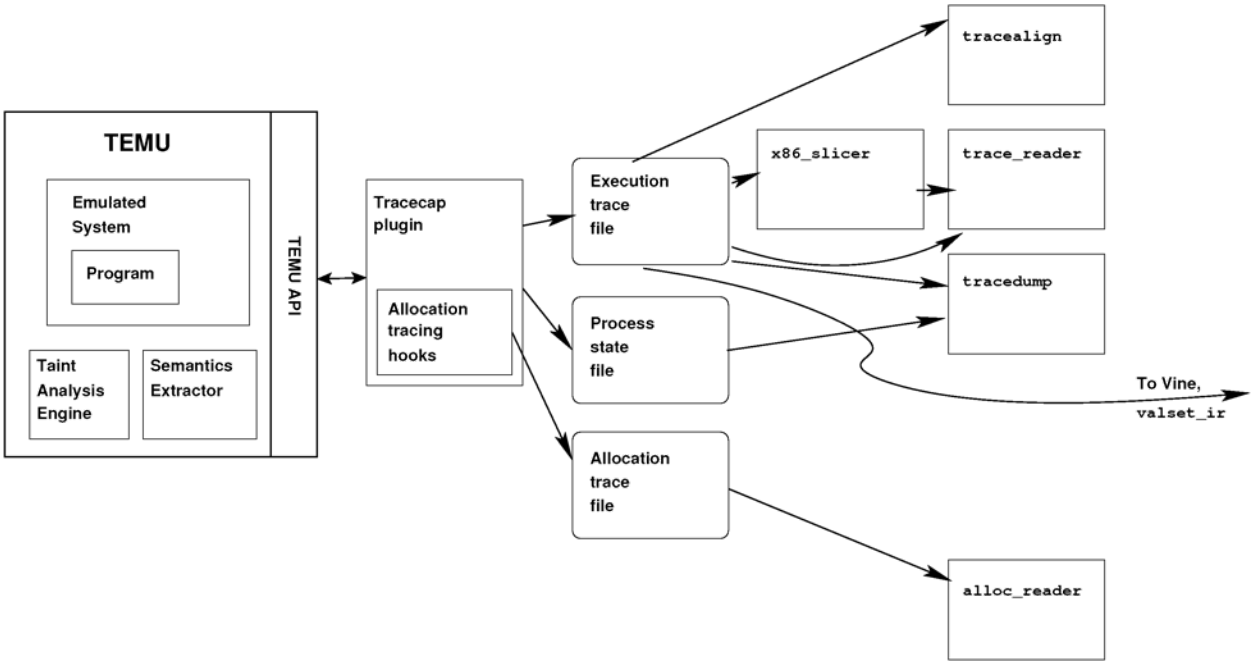

Figure 6: Trace-Related Tools Overview

Trace Reading: trace_reader

The traces collected by TEMU's Tracecap plugin contain detailed information about each executed instruction in a program, starting when the relevant input is first read and

going up to the point of a crash. The traces are stored in a specialized binary format, but the tool trace_reader will parse the trace and print the instruction in a text format. Figure 7 shows an example instruction that demonstrates almost all of the possible fields.

```
(00993850)7814507a: rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x0267ae7c
[0xed012800][4](CR) T1 {12 ()()(1111, 5) (1111, 5) } R@ecx[0x000007f3][4](RCW)
       T0    M@0x0267f0e0[0x0cc00b2f][4](CW) T1 {15 (1111, 5) (1111, 5) (1111, 5)
(1111, 5) } ESP: NUM_OP: 5 TID: 1756 TP: TPSrc EFLAGS: 0x00000202 CC_OP: 0x00000010
DF: 0x00000001 RAW: 0xf3a5 MEMREGS:     R@edi[0x0267f0e0][4](R)    T0     R@esi
[0x0267ae7c][4](R) T0
```

(00993850)	Index of instruction in trace
7814507a	Instruction address
rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]	Instruction disassembly
M@0x0267ae7c[0xed012800][4](CR)	Four-byte memory operand
T1 {12 ()()(1111, 5) (1111, 5) }	…high two bytes are tainted
R@ecx[0x000007f3][4](RCW)	Four-byte register operand
T0	…untainted
M@0x0267f0e0[0x0cc00b2f][4](CW)	Four-byte memory operand
T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111, 5) }	…all four bytes tainted
ESP:	ESP not used
NUM_OP: 5	Total number of operands
TID: 1756	Thread ID
TP: TPSrc	Instruction read tainted value
EFLAGS: 0x00000202	Condition code flags
CC_OP: 0x00000010	Last flags: from 32-bit subtract
DF: 0x00000001	String direction: increment
RAW: 0xf3a5	Instruction bytes: F3 A5
R@edi[0x0267f0e0][4](R) T0	Untainted address operand EDI
R@esi[0x0267ae7c][4](R) T0	Untainted address operand ESI

Figure 7: An example of trace_reader's output, with a description of the fields.

Of particular note in the output are operands and taint information. We call every value read or written by an instruction an operand. In the output of trace_reader, the operands are indicated with fields that look like M@0x0267ae7c[0xed012800][4](CR). Here "M" indicates a memory operand, the value after the @ is the address, the first set of square brackets enclose the value of the memory location, [4] indicates that the value is 4 bytes long, and (CR) indicates that the operand is conditionally read (in this case, if ECX is not zero).

For each byte in memory or a register, taint information records whether that byte is based on a tainted input value. For vulnerability analysis, we taint those inputs that could be under the control of an attacker, so tainted internal values of the program are also potentially under the attacker's control. For each tainted byte, Tracecap records a record, represented here as a pair of integers such as (1111, 5), indicating the source of

the tainted data. The first integer (here 1111), called the taint origin, represents a general source of information such as a tainted file or network stream. The second integer (here 5), called the taint offset, indicates a particular byte within the taint source. (In general, a value might depend on a number of different taint sources, but for efficiency we only record one.)

Dynamic Slicing: x86_slicer

In general, a slice from a value in a program is the set of statements that might influence that value. In understanding a vulnerability, slicing plays a complementary role to tainting: while tainting follows data flow forward from an attacker-controlled input to see what it affects, we can use slicing the follow data flow backwards from a crash to understand its causes. Often, some of the most relevant parts of an execution will be in the intersection of the set of tainted instructions and a backward slice from a crash.

Specifically, we use a BitBlaze tool named x86_slicer that performs backwards dynamic slicing on an execution trace (instructions play the role of statements in a source-level slice). Dynamic slicing refers to the fact that this slice includes only instructions executions that eventually affected the target value on the particular program execution we observed in the trace: instructions that might influence a value on other executions are not included. Since we operate on the binary level, the output of the slicing tool is a subset of the instructions from the original trace (stored in the same trace file format).

Another choice faced in choosing a slicing algorithm is what kinds of dependencies between instructions to include. In order to reduce the size of the slices produced, x86_slicer follows only data dependencies: instructions that read a value written by another instruction. Another possibility, not used by x86_slicer, is to follow control dependencies: an execution of an instruction is control dependent on an execution of a branch if it executed only because the branch was taken. Including control dependencies generally leads to slices that are much larger, containing some additional relevant instructions but also many additional irrelevant ones. Because this tradeoff is often unfavorable, x86_slicer does not currently include control dependencies in its slices.

Trace Alignment: tracealign

An alignment between two sequences is a matching between elements of one sequence and elements of another sequence that respects the ordering of each sequence. An alignment is helpful for understanding which parts of two related sequences are the same and which parts are different. For instance, the Unix diff command computes an alignment between the lines of two files. An alignment between two execution traces is similarly useful for understanding two related executions of a program; the BitBlaze tracealign tool computes such alignments.

Computing an alignment between two execution traces is a computationally easier task than aligning general text files as diff does or genetic sequences because execution traces already have a hierarchical structure. Specifically, tracealign uses the technique of execution indexing [XSZ08]. Execution indexing associates each point in program

execution (an instruction execution, for a binary) with a value in a totally ordered set, such that for any possible execution, the index values of the instructions are strictly increasing. An execution index has a form similar to a call stack and is lexicographically ordered, where locations within a single function are ordered compatibly with control-flow dominance.

In order to compute an alignment, tracealign performs two major passes: first, it constructs a (subset) of a control flow graph for each executed function, covering all the instructions belonging to the function that appear in the execution traces. Then, it uses these control-flow graphs to assign an index to each instruction in either trace, and processes the two traces in index order to find pairs that match (similar to merging two sorted lists).

A useful additional concept related to an alignment is a divergence point. Given an alignment, a divergence point is an instruction execution that is aligned in the two traces, but for which the following instruction in the respective traces are not aligned. Because of how the alignment works, it follows that a divergence point is a branch whose targets in the two traces were different. Divergence points are important in comparing traces because they are often related to the cause of the difference between two traces.

Alignment can be performed independently of tainting, but there are also at least two ways that alignment and tainting can be used together. First, tainting can be used as an additional filter to find interesting divergence points: a divergence caused by tainted data is more likely to be relevant that an untainted divergence. Second, alignment can be used to improve the results of tainting by removing excess tainting in a process we call *differential tainting*. Suppose that we align two runs in which the inputs are the same except for some tainted bytes. Then values later in execution should also only be tainted if they are different between the two runs: if a value is the same in both runs, but tainted, it is probably does not need to be tainted. We can use the alignment to clear the taint of variables that have the same values in both traces, and so make the tainting more precise.

Taint-enhanced Dumps: tracedump

Execution traces as viewed directly by trace_reader contain the complete information about the taint status of each tainted byte, because they contain an entry for every instruction that processed tainted data. However, because the information about each instruction appears only at its point of execution, they are not the most convenient form for checking the taint status of memory. For that purpose, Tracecap can produce taint-enhanced dumps, which contain the contents and taint information of a process' memory, at a given point in the execution. For example, Tracecap can output a crash dump when a program terminates unexpectedly. We provide a separate tool tracedump that takes as input a crash dump and the corresponding execution trace and prints information about the state of a program at the crash point. The printed information contains a stack backtrace, the contents of the general-purpose registers, and for each register that holds a pointer value, the contents of the memory bytes at and near that

location. For each byte in the dump, tracedump shows its taint origin and offset, if it is tainted.

Heap Allocation Tracing: alloc_reader

Since many vulnerabilities are related to improper handling of dynamically allocated memory, another useful piece of information in vulnerability understanding is a trace of the heap allocations and deallocations performed by a program. For this purpose, BitBlaze includes a hook plugin that can be used with TEMU to record the dynamic allocations a program performs, and a tool named alloc_reader to parse and query allocation traces. The allocation trace is correlated with the main execution trace, so that for any point in the execution trace, alloc_reader can report which allocation contains a given address. Or, if no allocation contains an address, it can report the closest allocations (useful in diagnosing overflows). alloc_reader can also produce a single report giving all the allocations that ever contained a particular address, and it can detect some other common allocation errors such as double frees.

Measuring Influence and Value Sets: valset_ir

Tainting gives useful information about which values in an execution might be under the control of an attacker, but one limitation of tainting is that it is just a binary attribute: a particular byte is either tainted or it is not. Often we would like to have more detailed information about what influence an attacker-controlled input exerts over a value. For this purpose, BitBlaze also supports measuring quantitative influence and the value set of a variable [NMS09]. For a variable such as a register or memory location at a particular point in execution, the value set of that variable is the set of all the different values that variable could take on, if the program were supplied different tainted (attacker-controlled) inputs. For instance for a 32-bit variable, the two extremes are that the value set might be a singleton, indicating that the attacker has no control over a variable, or it could be the full interval [0,232–1], indicating that the attacker has complete control.

It is also convenient to summarize the size of the value set with a single number that is the base-two logarithm of its number of elements, what we call the quantitative influence (measured in bits). For the same 32-bit variable example, the influence can range from 0 bits to 32 bits. A variable with 0 bits of influence corresponds to one that is not tainted, but for a tainted value, the range of possible influence values between 1 and 32 bits gives a more fine-grained measure of the attacker's control. If the influence is high, the attacker has a high degree of control that often means an exploit is possible. A low influence does not in general guarantee that no exploit is possible, but some kinds of un-exploitable control lead to low influence; in other cases low influence corresponds to a vulnerability that requires a more complex exploit.

The BitBlaze tool for measuring value sets and influence is called Valset (the executable is valset_ir), and it is based on Vine. Given an IL file generated from an execution trace, Valset converts the trace into a formula representing the relation between the input variables and a target output variable selected by the user. It then uses a series of queries to a decision procedure to estimate the size and contents of the value set. For

small value sets, Valset can compute their size and contents exactly; for larger value sets, it can approximate their size to a user-selected degree of precision.

Other Security Applications

In this section, we give an overview of the different security applications that we have enabled using the BitBlaze Binary Analysis Platform, ranging from automatic vulnerability detection, diagnosis, and defense, to automatic malware analysis and defense, to automatic model extraction and reverse engineering. For each security application, we give a new formulation of the problem based on the root cause in the relevant program. We then demonstrate that this problem formulation leads us to new approaches to address the security application based on the root cause. The results demonstrate the utility and effectiveness of the BitBlaze Binary Analysis Platform and its vision—it was relatively easy to build different applications on top of the BitBlaze Binary Analysis Platform and we could obtain effective results that previous approaches could not.

Vulnerability Detection, Diagnosis, and Defense

Sting: An Automatic Defense System against Zero-Day Attacks. Worms such as CodeRed and SQL Slammer exploit software vulnerabilities to self-propagate. They can compromise millions of hosts within hours or even minutes and have caused billions of dollars in estimated damage. How can we design and develop effective defense mechanisms against such fast, large scale worm attacks?

We have designed and developed Sting [TNL+07, NBS06a], a new end-to-end automatic defense system that aims to be effective against even zero-day exploits and protect vulnerable hosts and networks against fast worm attacks. Sting uses dynamic taint analysis to detect exploits to previously unknown vulnerabilities [NS05] and can automatically generate filters for dynamic instrumentation to protect vulnerable hosts [NBS06b].

Automatic Generation of Vulnerability Signatures. Input-based filters (a.k.a. signatures) provide important defenses against exploits before the vulnerable hosts can be patched. Thus, to automatically generate effective input-based filters in a timely fashion is an important task. We have designed and developed novel techniques to automatically generate input filters based on information about the vulnerability instead of the exploits, and thus generating filters that have zero false positives and can be effective against different exploit variants [BNS+06, BWJS07].

Automatic Patch-based Exploit Generation. Security patches do not only fix security vulnerabilities, they also contain sensitive information about the vulnerability that could enable an attacker to exploit the original vulnerable program and lead to severe consequences. We have demonstrated that this observation is correct—we have developed new techniques showing that given the patched version and the original vulnerable program, we can automatically generate exploits in our experiments with real world patches (often in minutes) [BPSZ08]. This opens the research direction of how to design and develop a secure patch dissemination scheme where attacker cannot use

information in the patch to attack vulnerable hosts before they have a chance to download and apply the patch.

Loop-extended Symbolic Execution: Buffer Overflow Diagnosis and Discovery. Loop-extended symbolic execution (or LESE) is a new technique that generalizes previous dynamic symbolic execution techniques to have a richer treatment of the behavior of loops [SPMS09]. LESE is a key enabler for powerful automated discovery of security vulnerabilities, especially buffer-overflows, which is highly inefficient with pure single-path dynamic symbolic execution. It also enables deeper diagnosis of known vulnerabilities, which allows automated signature generation tools to reason about variable-length input or repeated elements in the input.

Malware Analysis and Defense

Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. A myriad of malware such as keyloggers, Browser-helper Objects (BHO) based spyware, rootkits, and backdoors accesses and leaks users' sensitive information and breaches users' privacy. Can we have a unified approach to identify such privacy-breaching malware despite their widely-varied appearance? We have designed and developed Panorama [YSM+07] as a unified approach to detect privacy-breaching malware using whole-system dynamic taint analysis.

Renovo: Hidden Code Extraction from Packed Executables. Code packing is one technique commonly used to hinder malware code analysis through reverse engineering. Even though this problem has been previously researched, the existing solutions are either unable to handle novel packers, or vulnerable to various evasion techniques. We have designed and developed Renovo [KPY07], as a fully dynamic approach for hidden code extraction, capturing an intrinsic characteristic of hidden code execution.

HookFinder and HookScout: Identfying and Understanding Malware Hooking Behavior. One important malware attacking vector is its hooking mechanism. Malicious programs implant hooks for many different purposes. Spyware may implant hooks to be notified of the arrival of new sensitive data. Rootkits may implant hooks to intercept and tamper with critical system information to conceal their presence in the system. A stealth backdoor may also place hooks on the network stack to establish a stealthy communication channel with remote attackers. We have designed and developed HookFinder [YLS08] and its successor system HookScout [YPHS10] to automatically detect and analyze malware's hooking behaviors, by performing fine-grained impact analysis. Since this technique captures the intrinsic nature of hooking behaviors, it is well suited for identifying new hooking mechanisms.

BitScope: Automatically Dissecting Malware. The ability to automatically dissect a malicious binary and extract information from it is an important cornerstone for system forensic analysis and system defense. Malicious binaries, also called malware, include denial of service attack tools, spamming systems, worms, and botnets. New malware

samples are uncovered daily through widely deployed honeypots/honeyfarms, forensic analysis of compromised systems, and through underground channels. As a result of the break-neck speed of malware development and recovery, automated analysis of malicious programs has become necessary in order to create effective defenses. We have designed and developed BitScope, an architecture for systematically uncovering potentially hidden functionality of malicious software [BHK+07]. BitScope takes as input a malicious binary, and outputs information about its execution paths. This information can then be used by supplemental analysis designed to answer specific questions, such as what behavior the malware exhibits, what inputs activate interesting behavior, and the dependencies between its inputs and outputs.

Emulating Emulation-Resistant Malware. The authors of malware attempt to frustrate reverse engineering and analysis by creating programs that crash or otherwise behave differently when executed on an emulated platform than when executed on real hardware. To defeat such techniques, we have proposed an automated technique to dynamically modify the execution of a whole-system emulator to fool a malware sample's anti-emulation checks [KYH+09]. The technique uses a scalable trace matching algorithm to locate the point where emulated execution diverges, and then compares the states of the reference system and the emulator to create a dynamic state modification that repairs the difference.

Binary Code Reuse. Binary code reuse is the process of automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from an executable program, so that it is self-contained and can be reused by external code. Binary code reuse is useful for a number of security applications, including reusing the proprietary cryptographic or unpacking functions from a malware sample and for rewriting a network dialog. Using BitBlaze we have implemented a tool that uses a combination of dynamic and static analysis to automatically identify the prototype and extract the instructions of an assembly function into a form that can be reused by other C code [CJMS10].

Finding Malware Bugs with Decomposition and Re-Stitching. Attackers often take advantage of vulnerabilities in benign software, but there has been little research on the converse question of whether defenders can turn the tables by finding vulnerabilities in malware. We have provided an affirmative answer to that question by introducing a new technique, stitched dynamic symbolic execution, that makes it possible to use powerful exploration techniques based on symbolic execution in the presence of functionalities that are common in malware and otherwise hard to analyze, such as decryption and checksums [CPM+10]. The technique is based on decomposing the constraints induced by a program, solving only a subset, and then re-stitching the constraint solution into a complete input, and is implemented as part of BitFuzz. Applying our technique to 4 prevalent families of bots and other malware, we have found 6 bugs that could be exploited by a network attacker to terminate or subvert a broad range of malware client versions.

Automatic Model Extraction and Analysis

Polyglot and Dispatcher: Automatic Extraction of Protocol Message Format. Protocol reverse-engineering techniques extract the specification of unknown or undocumented network protocols and file formats. Protocol reverse engineering is important for many network security applications. Currently, protocol reverse engineering is mostly manual. For example, it took the open source Samba project over the course of 10 years to reverse engineer SMB, the protocol Microsoft Windows uses for sharing files and printers [Tri03].

We have proposed a new approach for automatic protocol reverse-engineering, which leverages the availability of a program that implements the protocol [CYLS07b]. Our approach uses dynamic program binary analysis techniques and is based on the intuition that monitoring how the program parses and constructs protocol messages reveals a wealth of information about the message structure and its semantics. Dispatcher [CPKS09] is a successor system that builds on Polyglot by implementing buffer deconstruction, a novel technique to extract the format of messages being sent by the application implementing the protocol, when Polyglot only extracted the format of messages received by the application. It can also infer the semantics of fields and reverse-engineer encrypted protocols by identifying the memory buffers that hold the unencrypted data.

Automatic Deviation Detection. Many network protocols and services have several different implementations. Due to coding errors and protocol specification ambiguities, these implementations often contain deviations, i.e., differences in how they check and process some of their inputs. Automatically finding deviations enables the automatic detection of potential implementation errors and the automatic generation of fingerprints that can be used to distinguish among implementations of the same network service. Automatic deviation detection (without requiring access to source code) is a challenging task— deviations usually happen in corner cases, and discovering deviations is often like finding needles in a haystack. We have developed deviation detection techniques for taking two binary implementations of the same protocol and automatically finding inputs which when sent to both implementations drive them to different output states [BCL+07]. By comparing two implementations we don't need access to a manually written model of the specification.

Replayer: Sound Replay of Application Dialogue. The ability to accurately replay application protocol dialogs is useful in many security-oriented applications, such as replaying an exploit for forensic analysis or demonstrating an exploit to a third party. A central challenge in application dialog replay is that the dialog intended for the original host will likely not be accepted by another without modification. For example, the dialog may include or rely on state specific to the original host such as its host name or a known cookie. In such cases, a straight-forward byte-by-byte replay to a different host with a different state (e.g., different host name) than the original dialog participant will likely fail. These state-dependent protocol fields must be updated to reflect the different state of the different host for replay to succeed. We have proposed the first approach for

soundly replaying application dialog where replay succeeds whenever the analysis yields an answer [NBFS06].

String-enhanced White-Box Exploration for Web Browsers. Content-sniffing attacks can occur when a web browser incorrectly infers the type of a served object, upgrading it to a type that can contain active content and so allowing cross-site scripting (XSS). As part of work discovering and fixing such vulnerabilities [BCS09], we have used BitBlaze to extract models of the content-sniffing functions in proprietary web browsers such as Internet Explorer 7 [CMBS09]. Our approach is based on string-enhanced white-box exploration, which improves the effectiveness of current white-box exploration techniques on programs that use strings, by reasoning directly about string operations, rather than about the individual byte-level operations that comprise them.

Availability
Some of the core parts of the BitBlaze platform, include Vine and TEMU (though not all of the particular applications that have been built using the platform), is available via our web site http://BitBlaze.cs.berkeley.edu/ or by request. Also, a virtual-machine image containing a binary release of the trace analysis tools described in Section 6 is available at http://BitBlaze.cs.berkeley.edu/release/blackhat-2010.html.

Case study: Adobe Reader

Now that you have a feel for the range of problems that BitBlaze can help solve, we turn back to our original problem, crash dump analysis. To demonstrate this, we used the BitBlaze tools on a variety of crashes found for Adobe Reader 9.2.0 for Windows. These were crashes found by mutation-based fuzzing, that is to say, we had no prior knowledge about the cause of the crash before analysis.

These crash inducing files originated from a 3 week fuzzing run conducted in November of 2009 and presented at CanSecWest [monkeys]. For Adobe Reader, there were 2582 crashes at 100 unique instruction pointers. Depending on the tool used, there were anywhere from 30-40 unique crashes in this bunch. BitBlaze could possibly be used to help determine the uniqueness of crashes, since it would have access to the full trace of execution. At the time of crash, full information about the program and memory contents could be accessed to help determine if two crashes really represented the same underlying vulnerability. We did not perform this analysis. The major problem with trying to do this this is that taking a trace for a large program like Reader can take several hours. It is not ideal to run such a tool on 2500 crashes, although it certainly is feasible for the 33 “unique” crashes categorized by !exploitable.

For these 33 files, !exploitable categorized their exploitability in the following way:

- 4 Exploitable
- 8 Probably exploitable
- 17 Unknown
- 4 Probably not exploitable

As you can see, !exploitable could not make a determination on around half the crashes. BitBlaze can help determine the fate of many of these Unknown crashes. BitBlaze has two pieces of information available that !exploitable does not. Namely, it has full taint information and it can slice data values to their origin. !exploitable can only examine the basic block in which the crash occurs. Therefore, !exploitable must be conservative in its analysis. For example, if something looks like a Null-pointer dereference, but !exploitable can not determine for sure where the Null value came from, it is forced to conclude it is an Unknown type crash. However, BitBlaze can determine whether the Null is always Null, or the result of something like a wild read, uninitialized value, or more importantly, some value from our input. Therefore, on these types of crashes, BitBlaze can move many of these crashes from the Unknown to the Not Exploitable bin. In the Adobe Reader example, it can very quickly remove 6 of the 17 files from consideration, see Section 4.3 for an example of the analysis of these types of crashes.

Also, BitBlaze can supply crash reports with taint information. Such reports provide opportunities for improved heuristics in determining exploitability as well as providing human analysts more information to base prioritization decisions. To use these crash reports with taint information, you must first make sure to taint the entire file. Then with the final state file, you can generate crash reports with taint. A couple of examples are given below.

First is a crash of Adobe Reader that appears to be a non-exploitable Null pointer dereference. !exploitable characterized it as Unknown.

```
Dump at EIP=20a2fd72,      mov    (%eax),%eax
Process: AcroRd32.exe      PID: 1908    TID: 1192    CTR: 132684549

Backtrace:
 (132684549)  unknown              0x20a2fd72  sub_20A2FD6E(?) +4
 (132684547)  unknown              0x20889825  sub_2088980F(?) +22
 (132684534)  unknown              0x2088b1d5  sub_2088B1BC(?) +25
 (132684519)  unknown              0x2088c3cf  sub_2088C3C8(?) +7
 (132684515)  unknown              0x208f4535  sub_208F4525(?) +16
 (132684506)  unknown              0x208f5a5d  sub_208F5A42(?) +27
 (132684478)  unknown              0x208f7087  sub_208F6FD4(?) +179
 (132668130)  unknown              0x208ca772  sub_208CA547(?) +555
 (96842557)  unknown              0x208774bb  sub_208773B3(?) +264
 (96833560)  AcroRd32.dll         0x009f3480  sub_9F3414(?) +108
 (96729701)  AcroRd32.dll         0x009c8e58  sub_9C8D15(?) +323
 (88528862)  AcroRd32.dll         0x009c8d0b  sub_9C8CF3(?) +24
 (88528851)  AcroRd32.dll         0x009c8c6e  sub_9C8C11(?) +93
 (88528789)  AcroRd32.dll         0x009c8bc3  sub_9C8AED(?) +214
 (88528728)  AcroRd32.dll         0x009c89ec  sub_9C8988(?) +100
 (88528656)  AcroRd32.dll         0x009c8630  sub_9C8542(?) +238
 (88521044)  unknown              0x221102d8  sub_22110272(?) +102
 (88520882)  unknown              0x22110449  sub_22110433(?) +22
 (88520876)  AcroRd32.dll         0x009c82c2  sub_9C8258(?) +106
 (88502113)  AcroRd32.dll         0x009c76b4  sub_9C754F(?) +357
 (88406838)  AcroRd32.dll         0x009c74ac  sub_9C72E0(?) +460
 (88391284)  AcroRd32.dll         0x009c72d8  sub_9C72CA(?) +14
```

```
EAX: 00000000
     -> N/A
EBX: 00000001
     -> N/A
ECX: 0268e354
     ->
 -12 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -08 02085ed4   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 -04 00000001   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +00 7439574f   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +04 02057238   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +08 00000003   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +12 32526963   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +16 76455349   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)

EDX: 01e3c0c0   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     ->
 -12 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -08 00e9008d   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -04 00e90031   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +00 001620c8   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +04 ffffffff   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +08 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +12 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +16 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)

ESI: 0269a8b0   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
     ->
 -12 020571c8   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 -08 02057238   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 -04 0269a850   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +00 02057190   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +04 0202c4e8   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +08 00000001   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +12 0269a1dc   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +16 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)

EDI: 0012e500   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     ->
 -12 ffffffff   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -08 0012e53c   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -04 208f708c   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +00 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +04 f5246116   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +08 000000d1   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +12 c0000000   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +16 c0000000   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)

EBP: 0012e4bc   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     ->
 -12 0012e4e8   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -08 010ea9e8   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -04 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +00 0012e4f8   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +04 208f5a62   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +08 0012e4dc   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +12 0012e500   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +16 f52460d2   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)

ESP: 0012e170   (NO TAINT)     (NO TAINT)
     ->
 -12 014c7608   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -08 02054b10   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 -04 026c4f6c   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
```

```
 +00 0012e178   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +04 0012e194   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +08 c0000005   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +12 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +16 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
```

eax is Null and not tainted. Some of the other registers are tainted or point to values that are addresses which contain or are near tainted values, but the actual register value being dereferenced is not tainted. You still can't know for sure, but with some (more) certainty you can conclude it is not exploitable. For comparison, !exploitable only says:

```
(c8.d28): Access violation - code c0000005 (!!! second chance !!!)
eax=00000000 ebx=00000001 ecx=02a30bcc edx=020acf78 esi=02a3d8c4 edi=0012e500
eip=20a2fd72 esp=0012e460 ebp=0012e4bc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000 efl=00000246
AcroForm!DllUnregisterServer+0x219bef:
20a2fd72 8b00            mov     eax,dword ptr [eax]
ds:0023:00000000=????????
0:000> !exploitable
Exploitability Classification: UNKNOWN
Recommended Bug Title: Data from Faulting Address may be used as a
return value starting at AcroForm!DllUnregisterServer+0x0000000000219bef
(Hash=0x687b6b23.0x366d6739)

The data from the faulting address may later be used as a return value
from this function.
```

More detail allows some more certainty about the origin of eax. Determining the root cause can also be performed, see later sections.

The following is another crash of Adobe Reader that is characterized by !exploitable as "Unknown".

```
Dump at EIP=08036f88,      mov    0x8(%edx),%ah
Process: AcroRd32.exe      PID: 316     TID: 324     CTR: 358264925

EAX: 00000000
     -> N/A
EBX: 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     -> N/A
ECX: 00000827   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     -> N/A
EDX: 0aaf90ef   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     -> N/A
ESI: 0aaf90fb
     -> N/A
EDI: 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     -> N/A
EBP: 0012d158   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
     ->
 -12 0012d798   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -08 08181bc8   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 -04 00000002   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +00 0012d158   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +04 0005760b   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +08 02823aa4   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +12 0012d824   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
```

```
 +16 02823bd4   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)

ESP: 0012ce04
     ->
 -12 397875ab   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 -08 ac8b0000   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 -04 20140000   ( 1111,43991)  ( 1111,43991)  ( 1111,43991)  ( 1111,43991)
 +00 0012ce0c   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +04 0012ce28   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +08 c0000005   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +12 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
 +16 00000000   (NO TAINT)     (NO TAINT)     (NO TAINT)     (NO TAINT)
```

In this crash, it is a read from an invalid address. The address itself is not tainted. It could still be based on an uninitialized value or some random wild read, but it does not come from data from the input file.

This last type of crash is one that BitBlaze is not very good at providing help analyzing, namely, BitBlaze does not typically provide much help for crashes involving invalid reads that are not near null. The problem is that invalid reads are not exploitable by themselves. However, they may lead to invalid writes or other problems that affect code flow later in execution. Or, just as likely, they may *not* lead to an invalid write or other control flow affecting behavior. Since the BitBlaze tools we use are fundamentally execution tracing tools, Bitblaze can not infer what will happen after the crash since those instructions were not executed. It *can* provide information about taint at the time of the crash. For example, it can tell the user if the address of the read was attacker controlled, as was seen from the previous crash dump. However, it is easy to imagine cases where an attacker controlled read does not represent a security problem for the program and cases where a read not controlled by the attacker is exploitable due to something that happens later. In either case, to be sure, more analysis needs to be conducted.

Root cause analysis examples

The last section is the heart of this paper and illustrated some uses of BitBlaze for determining the cause of a crash. Crash analysis is a laborious process that can normally take anywhere from a few hours to a week per crash. BitBlaze cannot do this automatically but does supply a variety of tools which hopefully reduce the time needed to conclude what caused a particular crash. BitBlaze provides tools for both data and execution analysis. These examples assume a good file which does not crash an application and a bad file which causes a crash. The bad file and the good file differ by only a few bytes, in most cases a single byte. We use BitBlaze to take traces of the execution of the bad file (x.trace) and the good file (y.trace). We taint the few bytes where they differ to observe how these bytes propagate through the program, since presumably they are what cause the problems. We then make a final trace (z.trace) using differential tainting techniques. Here, we again taint the few differing bytes but remove taint from any bytes that are the same between the good and bad file run. This serves as a way to remove unnecessary over-tainting situations. For most cases practical examples, we need to only look at the z (bad) and y (good) traces.

Each of these traces can be examined and any tainted bytes will be caused by the differences between the good and bad file. Note that other bytes from the input file will appear untainted, even though the attacker controls them. It is also possible to taint all bytes, as we did in the last section regarding when doing crash dumps.

The two traces, for the good and the bad file, can be aligned and examined to see where they execute differently. The cause of these unaligned regions can be examined by looking at the traces and it can be determined by looking at tainted values if it was caused by the differing bytes. Furthermore, registers and memory locations can be sliced to see where they originated. While debuggers generally provide facilities to trace bytes moving forward, i.e. hardware breakpoints, it is often a tedious process to back track where the data came from. The only drawback is while some slices are finished in seconds, some can take up to a couple of hours to complete.

One of the other advantages of working with a trace rather than a live program is "repeatability". Sometimes rerunning the same program with the same input will result in different memory addresses being used, different timing, and other complications that can make analysis more difficult. With odd calling conventions, even getting a back trace at a crash isn't always easy. When working with a trace, you get a view of the program that is always the same and stack backtraces are trivial to produce. Looking at a trace essentially give you the ability to step backwards in the program as well as forward, although no changes can be made to the execution.

With these tools in hand, we present their usage against several real crashes originating from mutation-based fuzzing against real programs. These examples are meant to not only illustrate the underlying vulnerability, but to also demonstrate how to use BitBlaze to find out information about the underlying vulnerability. Some of these steps could probably be conducted simply using a debugger or disassembler, but we use BitBlaze as much as possible to illustrate it's use. As for the vulnerabilities, there are times more analysis is probably needed, but this would lead too far away from the point of the paper which is to illustrate why BitBlaze is useful and how to use it.

An exploitable Adobe Reader crash

The following is an analysis using BitBlaze of a crash of Adobe Reader found by mutation based fuzzing. This vulnerability was found in Adobe Reader 9.2.0 and is currently fixed. We begin with the assumption that we have a trace of execution of a good file (y.trace) and a trace with differential tainting of a bad file (z.trace). We also assume we have an alignment file, y_z.aligned.txt.

With that, let's take a look at the crash. The first thing we need to do is see what instruction counter corresponds to the crash (the last one) and then we can view that instruction with trace_reader.

```
$ trace_reader -trace z.trace -header | grep Number
Number of instructions: 1816519
$ trace_reader -intel -trace z.trace -count -first 1816519
```

```
(01816519)603e598:  call   edx    R@edx[0xe1140d83][4](R)    T0     M@0x02ecf68c
[0x02699024][4](W)  T0
```

Crashes that crash at a control changing instruction are nice. In the good file, the same instruction is called, but edx has a legit value. We can see what instruction counter in the good trace corresponds to the crashing instruction counter in the bad trace using aligned.pl.

```
$ aligned.pl y_z.aligned.txt T1:01816518
<T0:2616698> ~ T1:01816518
(T0:02615515-02616698 ~ T1:01815335-01816518)
$ trace_reader -intel -trace y.trace -count -first 2616698 -last 2616698
(02616698)603e598:  call   edx    R@edx[0x08046e5d][4](R)    T0     M@0x02e5f68c
[0x0269900c][4](W)  T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111, 5) }
```

Ok, let's slice on the bad value in edx to see why this value is invalid.

```
$ x86_slicer -in-trace z.trace -ctr 1816519 -regloc edx > /dev/null
$ trace_reader -intel -trace EDX_0.trace
603e58f:     mov    edx,DWORD PTR [esi+0x48]   M@0x06003ef7[0xe1140d83][4](R)
       T0     R@edx[0x0269904c][4](W)    T0
```

Hmm... it came from memory address 0x06003ef7 from which the tool could not continue slicing. This means the data at this address was either set before we started tracing or is uninitialized. A careful reader will notice that this address is actually not from the heap or stack, but rather from the DLL we're investigating. Let's see where this value comes from, i.e. where esi originated. In order to do that, we have to see what instruction counter the instruction in the slice corresponds to. We do this by grep'ing for it in the execution trace.

```
$ trace_reader -intel -trace z.trace -count | grep 603e58f: | tail -1
(01816515)603e58f:  mov    edx,DWORD PTR [esi+0x48]   M@0x06003ef7[0xe1140d83][4]
(R)    T0     R@edx[0x0269904c][4](W)    T0
```

Ok, we are now ready to slice esi.

```
$ x86_slicer -in-trace z.trace -ctr 01816515 -regloc esi > /dev/null
$ trace_reader -intel -trace ESI_0.trace
6003eaa:     call   0x0000000006066d7e  J@0x00000000[0x00062ed4][4](R)    T0
       M@0x02ecf738[0x0652e260][4](W)    T0
6066d88:     pop    ecx    M@0x02ecf738[0x06003eaf][4](R)    T0     R@ecx[0x02ecf7e0]
[4](W) T0
6066d90:     push   ecx    R@ecx[0x06003eaf][4](R)    T0     M@0x02ecf75c[0x02ecf7ec]
[4](W) T0
6064bab:     mov    eax,DWORD PTR [edi+0x4]    M@0x02ecf75c[0x06003eaf][4](R)
       T0     R@eax[0x02ecf7b4][4](W)    T0
6064bae:     mov    DWORD PTR [esi+0x4],eax    R@eax[0x06003eaf][4](R)    T0
       M@0x02ecf7b8[0x00000000][4](W)    T0
603e583:     mov    esi,DWORD PTR [esi+0x4]    M@0x02ecf7b8[0x06003eaf][4](R)
       T0     R@esi[0x02ecf7b4][4](W)    T0
```

So esi comes from a value on the stack that was pushed on to it during a call, i.e. it is a return address. Therefore, esi came from an uninitialized stack variable. In general, this is exploitable, and as an attacker, we would begin to figure out how to exploit it at

this point. However, for the sake of demonstrating the tool, let's look a bit at what causes this situation to arise. Why is the variable in question uninitialized? To do this, let us examine the slice of esi for the good file to see where it was *supposed* to be initialized.

```
$ x86_slicer -in-trace y.trace -ctr 02616695 -regloc esi > /dev/null
$ trace_reader -intel -trace ESI_0.trace
60028a5:      mov    esi,DWORD PTR [edi] M@0x0652e234[0x02696428][4](R)    T0     R@esi
[0x02e5f750][4](W)  T0
60028ee:      mov    eax,DWORD PTR [esi+0x18]   M@0x02696440[0x00000001][4](R)
       T0     R@eax[0x01e64af8][4](W)    T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111,
5) }
60028e0:      mov    al,BYTE PTR [esi+0x1c]     M@0x02696444[0x00000001][1](R)
       T0     R@al[0x00000001][1](W)     T0
60028e3:      add    esi,0x20     I@0x00000000[0x00000020][1](R)    T0     R@esi
[0x02696428][4](RW) T0
60028e6:      neg    al     R@al[0x00000001][1](RW)    T0
60028e8:      sbb    eax,eax      R@eax[0x000000ff][4](R)    T0     R@eax[0x000000ff]
[4](RW)       T0
60028ea:      and    esi,eax      R@eax[0xffffffff][4](R)    T0     R@esi[0x02696448]
[4](RW)       T0
6002916:      mov    eax,esi      R@esi[0x02696448][4](R)    T0     R@eax[0x00000000]
[4](W) T0
6003e7a:      mov    DWORD PTR [esi+0x4],eax    R@eax[0x02696448][4](R)    T0
       M@0x02e5f754[0x0633f53e][4](W)    T0
6064bab:      mov    eax,DWORD PTR [edi+0x4]    M@0x02e5f754[0x02696448][4](R)
       T0     R@eax[0x02e5f7b4][4](W)    T0
6064bae:      mov    DWORD PTR [esi+0x4],eax    R@eax[0x02696448][4](R)    T0
       M@0x02e5f7b8[0x02e5f84c][4](W)    T0
603e583:      mov    esi,DWORD PTR [esi+0x4]    M@0x02e5f7b8[0x02696448][4](R)
       T0     R@esi[0x02e5f7b4][4](W)    T0
```

This slice agrees with the slice for crash for the final 3 lines, but before that differs as you move up. It looks like the value of esi comes from instructions in the range 0x60028a5-0x60028ea and is finally set at 0x6003e7a. Lets look at the alignment.

```
$ tail -20 y_z.aligned.txt
ALIGNED    @ T0:#02611291-02611391 (00000101 insts) ~ T1:#01814264-01814364 (00000101 insts)
DISALIGNED @ T0:#02611392-02614234 (00002843 insts) ~
ALIGNED    @ T0:#02614235-02614777 (00000543 insts) ~ T1:#01814365-01814907 (00000543 insts)
DISALIGNED @ T0:#02614778-02614787 (00000010 insts) ~ T1:#01814908-01814908 (00000001 insts)
ALIGNED    @ T0:#02614788-02614844 (00000057 insts) ~ T1:#01814909-01814965 (00000057 insts)
DISALIGNED @ T0:#02614845-02615113 (00000269 insts) ~ T1:#01814966-01814975 (00000010 insts)
ALIGNED    @ T0:#02615114-02615147 (00000034 insts) ~ T1:#01814976-01815009 (00000034 insts)
DISALIGNED @ T0:#02615148-02615161 (00000014 insts) ~ T1:#01815010-01815011 (00000002 insts)
ALIGNED    @ T0:#02615162-02615169 (00000008 insts) ~ T1:#01815012-01815019 (00000008 insts)
DISALIGNED @                                        ~ T1:#01815020-01815022 (00000003 insts)
ALIGNED    @ T0:#02615170-02615183 (00000014 insts) ~ T1:#01815023-01815036 (00000014 insts)
DISALIGNED @                                        ~ T1:#01815037-01815064 (00000028 insts)
ALIGNED    @ T0:#02615184-02615186 (00000003 insts) ~ T1:#01815065-01815067 (00000003 insts)
DISALIGNED @ T0:#02615187-02615238 (00000052 insts) ~
ALIGNED    @ T0:#02615239-02615492 (00000254 insts) ~ T1:#01815068-01815321 (00000254 insts)
DISALIGNED @ T0:#02615493-02615506 (00000014 insts) ~ T1:#01815322-01815323 (00000002 insts)
ALIGNED    @ T0:#02615507-02615514 (00000008 insts) ~ T1:#01815324-01815331 (00000008 insts)
DISALIGNED @                                        ~ T1:#01815332-01815334 (00000003 insts)
ALIGNED    @ T0:#02615515-02616698 (00001184 insts) ~ T1:#01815335-01816518 (00001184 insts)
DISALIGNED @ T0:#02616699-03654426 (01037728 insts) ~ T1:#01816519-01816519 (00000001 insts)
```

It turns out the instructions that are supposed to be setting the stack variable occur in the region where the good file executes 269 instructions and the bad file executes only 10. This can be seen a couple of ways. Either you could figure out which instruction

counters correspond to the instruction in the slice and look at the alignment, or you could just check each unaligned area and look for the instructions within it. We'll do the latter and read the trace for that particular section where the traces are not aligned.

```
$ trace_reader -intel -trace y.trace -eip -first 02614845 -last 02615113 | grep 60028
0600287a
0600287c
06002881
06002886
06002888
0600288b
06002890
06002891
06002894
06002895
06002898
0600289b
060028a1
060028a5
060028a7
060028b4
060028b6
060028a9
060028ac
060028af
060028ee
060028f1
060028f7
060028f9
060028e0
060028e3
060028e6
060028e8
060028ea
060028ec
```

So the good file initializes the variable when the traces are unaligned, the bad file never does. The reason why one initializes the variable and the other doesn't can be seen by looking at the comparison that takes place before the traces become unaligned.

```
$ trace_reader -intel -trace z.trace -first 01814964 -last 01814964
600f57b:     cmp    DWORD PTR [ebp-0x14],0x0   I@0x00000000[0x00000000][1](R)
       T0     M@0x02ecf760[0x00000000][4](R)    T0
$ trace_reader -intel -trace y.trace -first 2614843 -last 2614843
600f57b:     cmp    DWORD PTR [ebp-0x14],0x0   I@0x00000000[0x00000000][1](R)
       T0     M@0x02e5f760[0x026a6700][4](R)    T1 {15 (1111, 5) (1111, 5) (1111, 5)
(1111, 5) }
```

They differ because in the bad case, a NULL is found at a particular memory address while a valid pointer is found in the good case. We have reduced the problem of why one trace has an uninitialized stack value and the other doesn't to why this memory address has a non-NULL value. The same techniques we've already used can be used to trace this further back to try to answer this question. To begin, let's slice that address in the bad case.

```
$ x86_slicer -in-trace z.trace -ctr 01814964 -bs 0x02ecf760 -bo 0 -bl 4
$ trace_reader -intel -trace 0x02ecf760.trace
```

```
600f55c:      and    DWORD PTR [ebp-0x14],0x0   I@0x00000000[0x00000000][1](R)
       T0     M@0x02ecf760[0x060041e6][4](RW)   T0
```

In the bad file, this address is set to 0 and never changes from that value. In the good file, things are different.

```
$ x86_slicer -in-trace y.trace -ctr 2614843 -bs 0x02e5f760 -bo 0 -bl 4 > /dev/null
$ trace_reader -intel -trace 0x02e5f760.trace  | tail -1
8046fad:      mov    DWORD PTR [ecx],eax R@eax[0x026a6700][4](R)    T1 {15 (1111, 5)
(1111, 5) (1111, 5) (1111, 5) }  M@0x02e5f760[0x00000000][4](W)    T0
```

We can see this corresponds to counter 02614785 using grep. Looking for this instruction counter in the alignment, we see this instruction which set the contents at 0x02e5f760 occurred in another unaligned area:

```
DISALIGNED @ T0:#02614778-02614787 (00000010 insts) ~ T1:#01814908-01814908 (00000001 insts)
```

Again, we can see why the two traces became unaligned.

```
$ trace_reader -intel -trace z.trace -first 01814906 -last 01814907
8046f88:      cmp    ecx,DWORD PTR [eax+0xc]    R@ecx[0x00000000][4](R)    T0
       M@0x0269fb18[0x00000000][4](R)    T0
8046f8b:      jb     0x0000000008046f98  J@0x00000000[0x0000000d][4](R)    T0
$ trace_reader -intel -trace y.trace -first 2614776  -last 2614776
8046f88:      cmp    ecx,DWORD PTR [eax+0xc]    R@ecx[0x00000000][4](R)    T0
       M@0x026987b8[0x00000001][4](R)    T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111,
5) }
```

So, earlier in the trace, things started down the path toward an uninitalized variable when a particular memory location contained 0 in the bad case, but 1 in the good case. Slicing on this shows that in the bad case, it was always 0. I've removed push/pop pairs used for saving/restoring register values during function calls for clarity.

```
$ x86_slicer -in-trace z.trace -ctr 01814906 -bs 0x0269fb18 -bo 0 -bl 4 > /dev/null
$ trace_reader -intel -trace 0x0269fb18.trace
8151684:      xor    edi,edi       R@edi[0x00000000][4](R)    T0     R@edi[0x00000000]
[4](RW)       T0
81516a8:      mov    DWORD PTR [eax+0xc],edi    R@edi[0x00000000][4](R)    T0
       M@0x0269fb18[0x00000001][4](W)    T0
```

In the good file case, it is incremented.

```
$ trace_reader -intel -trace 0x026987b8.trace
8151684:      xor    edi,edi       R@edi[0x00000000][4](R)    T0     R@edi[0x00000000]
[4](RW)       T0
81516a8:      mov    DWORD PTR [eax+0xc],edi    R@edi[0x00000000][4](R)    T0
       M@0x026987b8[0x00000000][4](W)    T0
803adda:      inc    DWORD PTR [ecx+0x4] M@0x026987b8[0x00000000][4](RW)   T0
```

The instruction that does the incrementing corresponds to counter: 02613985. This instruction occurs in the huge unaligned area,

```
DISALIGNED @ T0:#02611392-02614234 (00002843 insts) ~
```

This process could be continued to see why our data eventually caused this problem.

Besides using the command line tools discussed thus far, the problem could have been approached using the IDA Pro visualization tool which is part of the BitBlaze platform. Figure 8 shows the trace from this crash loaded into the visualization tool. You can see some of the shortcuts which can be used to navigate through the information from the trace.

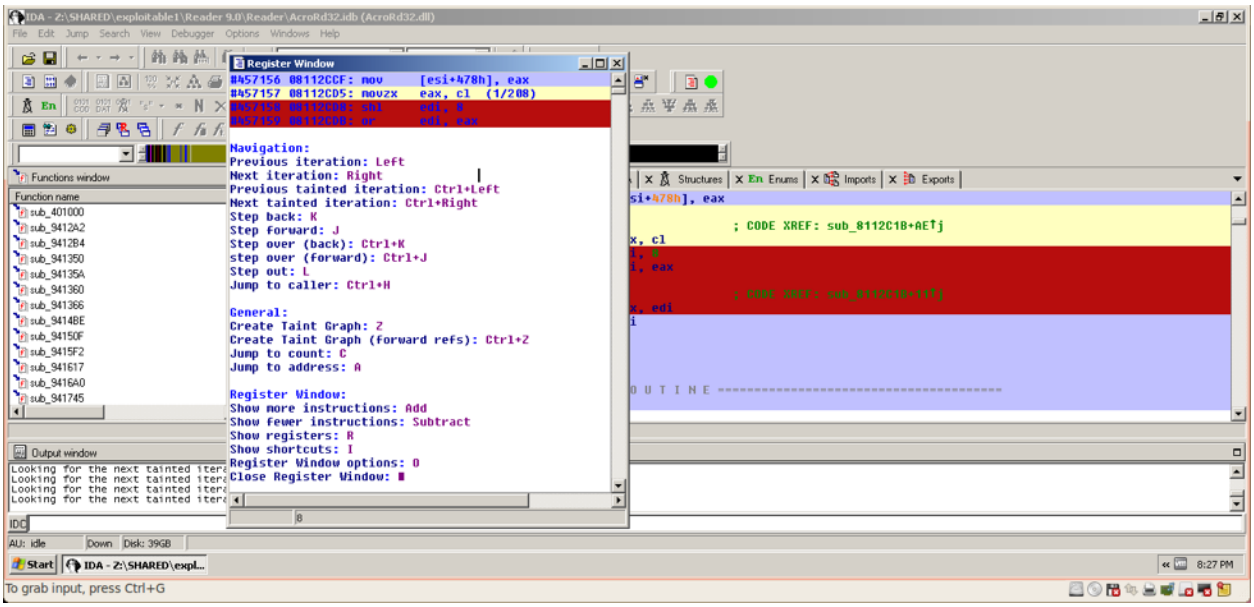

Figure 8: BitBlaze trace visualization tool.

One of the main benefits of the visualization tool is that the taint information can seen along with the disassembly, see Figure 9. Also, information about when registers are read or written can be used to navigate through the disassembly.

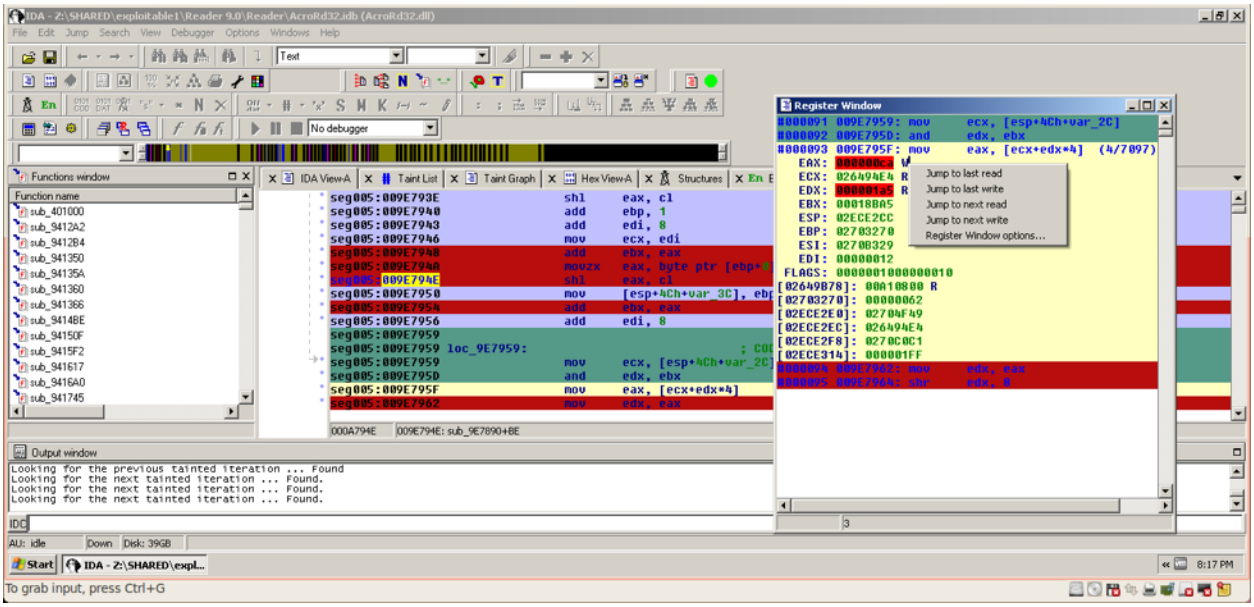

Figure 9: The register window in the visualization tool now contains taint information

Finally, the visualization tool provides new windows which include a list of all tainted instructions as well as the ability to generate graphs which show the propagation of tainted data, see Figure 10.

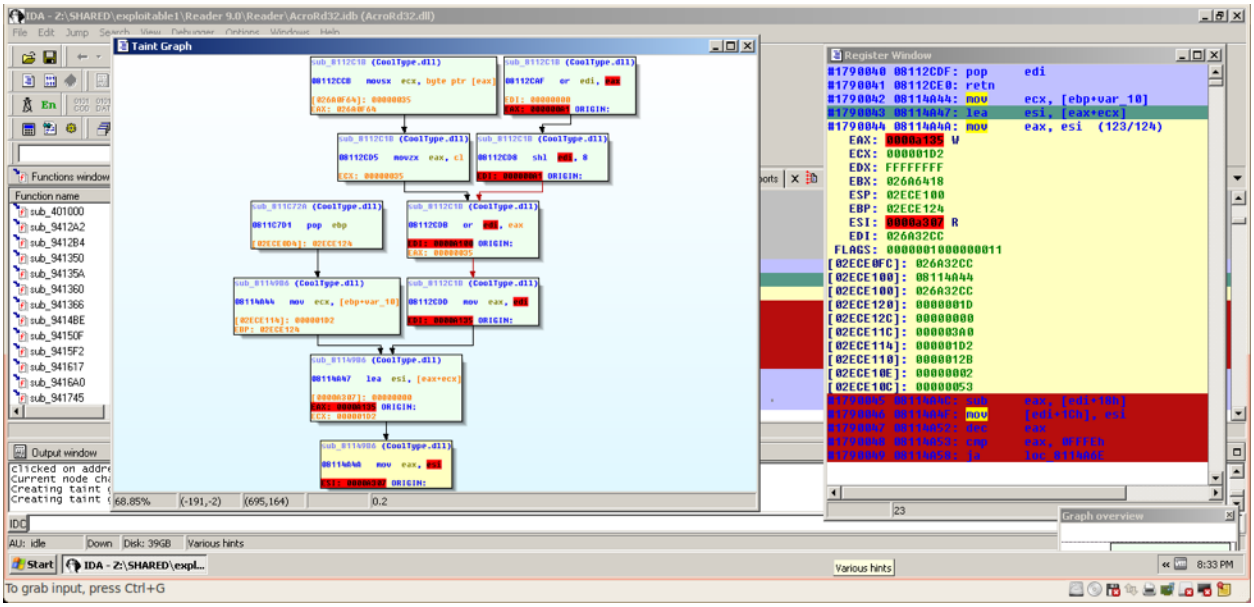

Figure 10: The taint graph

For more information on the visualization plugin, please consult the BitBlaze project page.

A second exploitable Adobe Reader crash

This is another exploitable Adobe Reader crash originally found for Adobe Reader 9.2.0 and recently fixed by Adobe. This crash was found by mutation based fuzzing. Let's see the crash. In this case we look at the x.trace file instead of the z.trace file for a reason that will become clear later.

```
$ trace_reader -trace x.trace -header | grep Number
Number of instructions: 1173735
$ trace_reader -intel -trace x.trace -first 1173735 -count -v
(01173735)7001400:  lock dec DWORD PTR [eax]   M@0xf5fc081d[0x00000000][4](RW)   T0
ESP:  NUM_OP: 2 TID: 1756 TP: TPSrc EFLAGS: 0x00000083 CC_OP: 0x00000018 DF:
0x00000001 RAW: 0xf0ff08 MEMREGS:       R@eax[0xf5fc081d][4](R)    T1 {15 (1111, 5)
(1111, 5) (1111, 5) (1111, 5) }
```

It crashes trying to decrement an invalid address contained in eax. eax is tainted, that is a good sign. If we controlled eax, this would definitely be exploitable, see [SMS].

Let's look at the alignment to get an idea of what goes wrong in the bad case.

```
$ tail -2 x_y.aligned.txt
ALIGNED    @ T0:#01152003-01173723 (00021721 insts) ~ T1:#01151778-01173498 (00021721
insts)
DISALIGNED @ T0:#01173724-01173735 (00000012 insts) ~ T1:#01173499-01477799 (00304301
insts)
```

The crash is in an unaligned area. Let's look at the traces near the divergence to see why they diverged. Here is the trace of the bad file near this spot:

```
$ trace_reader -intel -trace x.trace -count -first 01173718 -last 01173723
(01173718)8002046:  test   eax,eax      R@eax[0xf5fc0801][4](R)    T1 {15 (1111, 5)
(1111, 5) (1111, 5) (1111, 5) }  R@eax[0xf5fc0801][4](R)    T1 {15 (1111, 5) (1111, 5)
(1111, 5) (1111, 5) }
(01173719)8002048:  mov    DWORD PTR [esi],ecx R@ecx[0x01dd98f8][4](R)    T0
      M@0x026a0d04[0xf5fc0801][4](W)    T0
(01173720)800204a:  mov    ecx,DWORD PTR [ebp-0x4]    M@0x0012cab8[0x01dd991c][4]
(R)    T0     R@ecx[0x01dd98f8][4](W)    T0
(01173721)800204d:  mov    DWORD PTR [ebp-0x8],eax    R@eax[0xf5fc0801][4](R)    T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }      M@0x0012cab4[0x01dd98f8][4](W)    T0
(01173722)8002050:  mov    DWORD PTR [esi+0x4],ecx    R@ecx[0x01dd991c][4](R)
      T0     M@0x026a0d08[0x00000000][4](W)    T0
(01173723)8002053:  je     0x000000000800205d  J@0x00000000[0x0000000a][4](R)    T0
```

It looks like the test instruction at counter 01173718 is the cause of the divergence. Let's check out what happens at that instruction in the good file.

```
$ aligned.pl x_y.aligned.txt 01173718
T0:01173718 ~ <T1:1173493>
(T0:01152003-01173723 ~ T1:01151778-01173498)

$ trace_reader -intel -trace y.trace -count -first 1173493 -last 1173493
(01173493)8002046:  test   eax,eax      R@eax[0x00000000][4](R)    T1 {15 (1111, 5)
(1111, 5) (1111, 5) (1111, 5) }  R@eax[0x00000000][4](R)    T1 {15 (1111, 5) (1111, 5)
(1111, 5) (1111, 5) }
```

They diverge at 0x8002046 because in the bad file, eax is not 0, but rather 0xf5fc0801. Notice that this value of eax is almost the memory address that eventually tries to be decremented during the crash.

Let's slice on the value of eax at the time of the crash and see where it comes from in order to know if we control it.

```
$ x86_slicer -in-trace x.trace -ctr 01173735 -include-memregs true -regloc eax > /dev/
null
$ trace_reader -intel -trace EAX_0.trace -v | tail -11
808857d:      lea     ecx,[ecx+eax*8]      A@0x026a0d04[0x00000000][4](R)    T0     R@ecx
[0x026a0504][4](W)  T0 ESP:  NUM_OP: 5 TID: 1756 TP: TPSrc EFLAGS: 0x00000213 CC_OP:
0x00000010 DF: 0xFFFFFFFF RAW: 0x8d0cc1 MEMREGS:      R@ecx[0x026a0504][4](R)
       T0     R@eax[0x00000100][4](R)    T1 {3 (1111, 5) (1111, 5) ()()}
8002037:      mov     esi,ecx      R@ecx[0x026a0d04][4](R)    T1 {15 (1111, 5) (1111, 5)
(1111, 5) (1111, 5) }      R@esi[0x0267d178][4](W)    T0 ESP:  NUM_OP: 2 TID: 1756 TP:
TPSrc EFLAGS: 0x00000213 CC_OP: 0x00000010 DF: 0xFFFFFFFF RAW: 0x8bf1 MEMREGS:
800127a:      push    esi    R@esi[0x026a0d04][4](R)    T1 {15 (1111, 5) (1111, 5) (1111,
5) (1111, 5) }      M@0x0012caa4[0x0700f603][4](W)    T0 ESP:       R@esp[0x0012caa8]
[4](R) T0 NUM_OP: 3 TID: 1756 TP: TPSrc EFLAGS: 0x00000213 CC_OP: 0x00000010 DF:
0xFFFFFFFF RAW: 0x56 MEMREGS:     R@esp[0x0012caa8][4](R)    T0
8001297:      pop     esi    M@0x0012caa4[0x026a0d04][4](R)    T1 {15 (1111, 5) (1111, 5)
(1111, 5) (1111, 5) }      R@esi[0x0012cab4][4](W)    T0 ESP:       R@esp[0x0012caa4]
[4](R) T0 NUM_OP: 3 TID: 1756 TP: TPSrc EFLAGS: 0x00000202 CC_OP: 0x0000001C DF:
0xFFFFFFFF RAW: 0x5e MEMREGS:     R@esp[0x0012caa4][4](R)    T0
8002044:      mov     eax,DWORD PTR [esi] M@0x026a0d04[0xf5fc0801][4](R)    T0     R@eax
[0x0012cab4][4](W)  T0 ESP:  NUM_OP: 3 TID: 1756 TP: TPMemReadIndex EFLAGS: 0x00000202
CC_OP: 0x0000001C DF: 0xFFFFFFFF RAW: 0x8b06 MEMREGS:       R@esi[0x026a0d04][4]
(R)    T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111, 5) }
8002055:      push    eax    R@eax[0xf5fc0801][4](R)    T1 {15 (1111, 5) (1111, 5) (1111,
5) (1111, 5) }      M@0x0012caac[0x0012cad8][4](W)    T0 ESP:       R@esp[0x0012cab0]
[4](R) T0 NUM_OP: 3 TID: 1756 TP: TPSrc EFLAGS: 0x00000282 CC_OP: 0x00000018 DF:
0xFFFFFFFF RAW: 0x50 MEMREGS:     R@esp[0x0012cab0][4](R)    T0
7005caf:      mov     ecx,DWORD PTR [esp+0x4]    M@0x0012caac[0xf5fc0801][4](R)    T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }      R@ecx[0x01dd991c][4](W)    T0 ESP:
       R@esp[0x0012caa8][4](R)    T0 NUM_OP: 4 TID: 1756 TP: TPSrc EFLAGS: 0x00000282
CC_OP: 0x00000018 DF: 0xFFFFFFFF RAW: 0x8b4c2404 MEMREGS:   R@esp[0x0012caa8][4]
(R)    T0
70013f3:      mov     ebp,esp      R@esp[0x0012caa4][4](R)    T0     R@ebp[0x0012cabc]
[4](W) T0 ESP:       R@esp[0x0012caa4][4](R)    T0 NUM_OP: 2 TID: 1756 TP: TPNone
EFLAGS: 0x00000282 CC_OP: 0x00000018 DF: 0xFFFFFFFF RAW: 0x8bec MEMREGS:
70013f7:      lea     eax,[ecx+0x1c]      A@0xf5fc081d[0x00000000][4](R)    T0     R@eax
[0xf5fc0801][4](W)  T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111, 5) } ESP:  NUM_OP: 4
TID: 1756 TP: TPSrc EFLAGS: 0x00000282 CC_OP: 0x00000018 DF: 0xFFFFFFFF RAW: 0x8d411c
MEMREGS:      R@ecx[0xf5fc0801][4](R)    T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111,
5) }
70013fa:      mov     DWORD PTR [ebp-0x8],eax    R@eax[0xf5fc081d][4](R)    T1 {15 (1111,
5) (1111, 5) (1111, 5) (1111, 5) }      M@0x0012ca9c[0xf5fc0801][4](W)    T1 {15 (1111,
5) (1111, 5) (1111, 5) (1111, 5) } ESP:  NUM_OP: 4 TID: 1756 TP: TPSrc EFLAGS:
0x00000282 CC_OP: 0x00000018 DF: 0xFFFFFFFF RAW: 0x8945f8 MEMREGS:       R@ebp
[0x0012caa4][4](R)  T0
70013fd:      mov     eax,DWORD PTR [ebp-0x8]    M@0x0012ca9c[0xf5fc081d][4](R)    T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }      R@eax[0xf5fc081d][4](W)    T1 {15 (1111,
5) (1111, 5) (1111, 5) (1111, 5) } ESP:  NUM_OP: 4 TID: 1756 TP: TPSrc EFLAGS:
0x00000282 CC_OP: 0x00000018 DF: 0xFFFFFFFF RAW: 0x8b45f8 MEMREGS:       R@ebp
[0x0012caa4][4](R)  T0
```

Tracing back the value of eax, we see that at instruction 0x8002044, it came from a memory read from the address contained in esi which originated from ecx. The

instruction at 0x808857d computes this memory address where eax is read from by the formula ecx = ecx + 8*eax. Observe that eax = 0x100 and is tainted.

Things are different in the good file:

```
$ aligned.pl x_y.aligned.txt 01173680
T0:01173680 ~ <T1:1173455>
(T0:01152003-01173723 ~ T1:01151778-01173498)
$ trace_reader -intel -trace y.trace -v -count -first 1173455 -last 1173455
(01173455)808857d:  lea    ecx,[ecx+eax*8]     A@0x026a4cc0[0x00000000][4](R)
      T0     R@ecx[0x026a4aa0][4](W)    T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111,
5) } ESP:  NUM_OP: 5 TID: 1448 TP: TPSrc EFLAGS: 0x00000213 CC_OP: 0x00000010 DF:
0x00000001 RAW: 0x8d0cc1 MEMREGS:       R@ecx[0x026a4aa0][4](R)    T1 {15 (1111, 5)
(1111, 5) (1111, 5) (1111, 5) }  R@eax[0x00000044][4](R)    T1 {3 (1111, 5) (1111, 5)
()()}
```

so in the non-crashing case, the offset into the buffer that is read is not as far into the buffer as in the crashing case. Namely, it is 0x44 bytes into it rather than 0x100. This buffer offset is tainted, but exactly how is it controlled? Let's slice it to find out where it comes from.

```
$ x86_slicer -in-trace x.trace -ctr 01173680 -regloc eax > /dev/null
$ trace_reader -intel -trace EAX_0.trace | tail -28
7814507a:     rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x0267ae7c[0xed012800]
[4](CR)       T1 {12 ()()(1111, 5) (1111, 5) } R@ecx[0x000007f3][4](RCW)  T0
      M@0x0267f0e0[0x0cc00b2f][4](CW)  T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111,
5) }
8114da3:      movsx  ecx,BYTE PTR [eax]  M@0x0267f0e3[0x000000ed][1](R)    T1 {1 (1111,
5) ()()()}    R@ecx[0x00000029][4](W)    T0
8114dad:      movzx  ax,cl R@cl[0x000000ed][1](R)     T1 {1 (1111, 5) ()()()}    R@ax
[0x0000f0e4][2](W)  T0
8114db1:      movzx  eax,ax R@ax[0x000000ed][2](R)     T1 {1 (1111, 5) ()()()}    R@eax
[0x026700ed][4](W)  T1 {1 (1111, 5) ()()()}
8114db4:      mov    DWORD PTR [ebp-0x10],eax  R@eax[0x000000ed][4](R)    T1 {1 (1111,
5) ()()()}    M@0x0012ce38[0x0000002a][4](W)    T0
8114e1f:      inc    DWORD PTR [ebp-0x10]      M@0x0012ce38[0x000000ed][4](RW)  T1 {1
(1111, 5) ()()()}
8114e1f:      inc    DWORD PTR [ebp-0x10]      M@0x0012ce38[0x000000ee][4](RW)  T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }
8114e1f:      inc    DWORD PTR [ebp-0x10]      M@0x0012ce38[0x000000ef][4](RW)  T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }
...
8114e1f:      inc    DWORD PTR [ebp-0x10]      M@0x0012ce38[0x000000fd][4](RW)  T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }
8114e1f:      inc    DWORD PTR [ebp-0x10]      M@0x0012ce38[0x000000fe][4](RW)  T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }
8114e1f:      inc    DWORD PTR [ebp-0x10]      M@0x0012ce38[0x000000ff][4](RW)  T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }
8114e0e:      push   DWORD PTR [ebp-0x10]      M@0x0012ce38[0x00000100][4](R)   T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }     M@0x0012ce24[0x000000ff][4](W)   T1 {15
(1111, 5) (1111, 5) (1111, 5) (1111, 5) }
8114c50:      movzx  eax,WORD PTR [esp+0x8]     M@0x0012ce24[0x00000100][2](R)   T1 {3
(1111, 5) (1111, 5) ()()}  R@eax[0x0267a848][4](W)    T0
8114c59:      mov    DWORD PTR [edi+0x10],eax   R@eax[0x00000100][4](R)    T1 {3 (1111,
5) (1111, 5) ()()}  M@0x0267d17c[0xffffffff][4](W)    T0
808856f:      mov    eax,DWORD PTR [esi+0x4]    M@0x0267d17c[0x00000100][4](R)   T1 {3
(1111, 5) (1111, 5) ()()}  R@eax[0x0012cad8][4](W)    T0
```

The value 0x100 comes from the value 0xed after being incremented a bunch of times. This value 0xed comes from the dword 0xed012800 located at 0x0267ab9c. This value is copied at this instruction:

```
$ trace_reader -intel -trace x.trace -count -v -first 993850 -last 993850
(00993850)7814507a: rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x0267ae7c
[0xed012800][4](CR) T1 {12 ()()(1111, 5) (1111, 5) } R@ecx[0x000007f3][4](RCW)
      T0     M@0x0267f0e0[0x0cc00b2f][4](CW)   T1 {15 (1111, 5) (1111, 5) (1111, 5)
(1111, 5) } ESP:  NUM_OP: 5 TID: 1756 TP: TPSrc EFLAGS: 0x00000202 CC_OP: 0x00000010
DF: 0x00000001 RAW: 0xf3a5 MEMREGS:     R@edi[0x0267f0e0][4](R)    T0     R@esi
[0x0267ae7c][4](R)  T0
```

So it is part of a rather large copy (at least 0x7fc). It would make sense that this is a copy of the entire uncompressed stream. In fact this is true, and can be verified by looking for long sequences of tainted buffers originating from a single tainted byte, which is what you would expect a decompressed buffer to look like from a tainted perspective. By the way, this is why we chose to work with x.trace instead of z.trace. In the differential tainting trace, these large sections of tainted bytes in the uncompressed stream would not be tainted since they would agree in the good and bad file cases.

The address that the 0xed originated from (0x0267ae7c) is verified to be in one of these uncompressed streams, which we control as they come from the file. We use a couple of python scripts to find large regions of adjacent tainted bytes in memory.

```
$ count2.py x.trace.disasm > /dev/null

$ buffer.py memaddr.txt 2048
[0x0265eca1] - [0x0265fca2] 4097 bytes: 4098
[0x0267a01c] - [0x0267ee4c] 20016 bytes: 5053
[0x0267ef64] - [0x026810a8] 8516 bytes: 2226
[0x02690564] - [0x02692c98] 10036 bytes: 2510
```

Ok, so we have control over an offset used to read an address from a buffer and we are able to decrement the value at that address. But can we really read past the end of the buffer or are we reading uninitialized values inside a large buffer? Looking at the lea instruction, we see the buffer in question begins at address 0x026a0504. Let's see where this buffer was allocated.

```
$ alloc_reader -alloc x.trace.alloc -addr 0x026a0504
Found 1 buffers
[01151948,end] 0x026a04e0 (65064)
```

So, a very large buffer of size 65064 was allocated at instruction counter 01151948 and our buffer is in the middle of it (36 bytes in, the bad read occurs at 2084 bytes in). This smells like a custom memory allocator to me.

Let's see where this large allocation occurred in the program to try to understand the custom memory allocator. We look at the call stack when the malloc was called (at instruction counter 01151948). We trace the callstack up until it is in the DLL's we're interested in (CoolType.dll is loaded at address 0x08xxxxx).

```
$ trace_reader -intel -trace x.trace -count -first 01151948 -last 1152200 | grep -A1
ret | grep -m1 -B1 ")80"
(01152180)700453a:  ret          M@0x0012cadc[0x0800139e][4](R)    T0
(01152181)800139e:  test   eax,eax      R@eax[0x026a0504][4](R)    T0     R@eax
[0x026a0504][4](R)  T0
```

Whatever function was called at 0x8001394 returned the address of our buffer. It looks like this time we were lucky and our request for a buffer forced the custom allocator to make a call to malloc. Looking closer at the function called at 0x8001394, we see that it takes one argument and returns a buffer to use. It is a wrapper to something that behaves like malloc. Let's see how big of a buffer we requested.

First, lets see when this malloc-wrapper was called for our buffer (0x026a0504) and how big the requested "allocation" was. To do this, we just grep for that relevant instruction counter.

```
$ trace_reader -intel -trace x.trace -count -last 01152181 | grep 8001394: | tail -1
(01150773)8001394:  push   DWORD PTR [esp+0x4] M@0x0012cae8[0x00000800][4](R)
       T0     M@0x0012cae0[0x0819dbb4][4](W)    T0
```

So, the value of eax in the crash comes from a read past the end of a buffer from a custom allocator. We now know this since we read the value 0x100*8 bytes into the buffer and it was only "allocated" to be of size 0x800. By carefully arranging memory allocations, via a heap feng-shui technique[heap], we should be able to control the contents of this next buffer and thus be able to perform arbitrary memory writes.

A non-exploitable Adobe Reader crash characterized as unknown
This is a crash found by mutation based fuzzing Adobe Reader. It is still in the latest version. Let's check out the crash.

```
$ trace_reader -trace z.trace -header | grep Number
Number of instructions: 272998
$ trace_reader -intel -v -trace z.trace -first 272998
208063fb:     mov    edx,DWORD PTR [ecx+0x4]    M@0x00000014[0x00000000][4](R)
       T0     R@edx[0x00000000][4](W)    T0 ESP:  NUM_OP: 4 TID: 1712 TP: TPNone
EFLAGS: 0x00000083 CC_OP: 0x00000008 DF: 0x00000001 RAW: 0x8b5104 MEMREGS:      R@ecx
[0x00000010][4](R)T0
```

It looks like it is probably a Null pointer exception and not exploitable. But maybe that 0x14 came from a wild read, or even worse/better, our input data. For these reasons, ! exploitable won't say it is not exploitable.

The first observation is ecx is not tainted, so that is even further evidence indicating it is not exploitable. Beyond that, lets slice ecx to see where the 0x14 originated. I'm omitting the push/pop pairs for saving ecx during function calls for clarity:

```
$ x86_slicer -in-trace z.trace -ctr 272998 -regloc ecx > /dev/null
$ trace_reader -intel -trace ECX_0.trace
208f46df:     xor    esi,esi      R@esi[0xc0000000][4](R)    T0     R@esi[0xc0000000]
[4](RW)       T0
208f4bbf:     mov    eax,esi      R@esi[0x00000000][4](R)    T0     R@eax[0x00000b12]
[4](W) T0
```

```
208f5c6b:     mov    ebx,eax      R@eax[0x00000000][4](R)    T0     R@ebx[0x025cdf3c]
[4](W) T0
208f5c91:     mov    eax,ebx      R@ebx[0x00000000][4](R)    T0     R@eax[0x00000000]
[4](W) T0
208f63f8:     mov    esi,eax      R@eax[0x00000000][4](R)    T0     R@esi[0xc0000000]
[4](W) T0
208f648f:     mov    ecx,esi      R@esi[0x00000000][4](R)    T0     R@ecx[0x00000000]
[4](W) T0
2088c9a2:     mov    esi,ecx      R@ecx[0x00000000][4](R)    T0     R@esi[0x00000000]
[4](W) T0
2088c9a4:     lea    ebx,[esi+0x10]      A@0x00000010[0x00000000][4](R)    T0     R@ebx
[0xc0000000][4](W)  T0
2088c9a7:     mov    ecx,ebx      R@ebx[0x00000010][4](R)    T0     R@ecx[0x00000000]
[4](W) T0
```

So there are no memory reads, it came straight from an xor through moves and an lea instruction. This is definitely not exploitable, as it is a Null-ptr dereference. As a security researcher, you'd be done at this point. But if you really wanted to find the underlying problem, you could also use BitBlaze for that.

First let's compare the crash location between the good file and the bad file.

```
$ trace_reader -intel -trace z.trace -first 272998 -last 272998
208063fb:     mov    edx,DWORD PTR [ecx+0x4]    M@0x00000014[0x00000000][4](R)
       T0     R@edx[0x00000000][4](W)    T0
$ aligned.pl y_z.aligned.txt T1:272998
<T0:271645> ~ T1:272998
(T0:00271451-00271645 ~ T1:00272804-00272998)
$ trace_reader -intel -trace y.trace -first 271645 -last 271645
208063fb:     mov    edx,DWORD PTR [ecx+0x4]    M@0x0202c5fc[0x025d91f8][4](R)
       T0     R@edx[0x00000000][4](W)    T0
```

In the good file case, ecx has a legitimate value. Let's take a look at the alignment.

```
$ tail -6 y_z.aligned.txt
ALIGNED    @ T0:#00234897-00265964 (00031068 insts) ~ T1:#00241546-00272613 (00031068 insts)
DISALIGNED @ T0:#00265965-00271264 (00005300 insts) ~ T1:#00272614-00272619 (00000006 insts)
ALIGNED    @ T0:#00271265-00271448 (00000184 insts) ~ T1:#00272620-00272803 (00000184 insts)
DISALIGNED @ T0:#00271449-00271450 (00000002 insts) ~
ALIGNED    @ T0:#00271451-00271645 (00000195 insts) ~ T1:#00272804-00272998 (00000195 insts)
DISALIGNED @ T0:#00271646-00354492 (00082847 insts) ~
```

Toward the end of execution, it briefly diverges for 2 instructions and before that for quite a few instructions. Nothing particularly interesting happens in the 2 instructions.

```
$ trace_reader -intel -trace y.trace -first 00271449 -last 00271450 -count
(00271449)208f5c88: cmp    WORD PTR [ebx+0x2c],0x0   I@0x00000000[0x00000000][1]
(R)    T0     M@0x0202c614[0x00000000][2](R)T0
(00271450)208f5c8d: je     0x00000000208f5c91  J@0x00000000[0x00000004][4](R)    T0
```

If you slice on the legit value of ecx in the good file, it agrees with the slice of ecx in the bad file, except toward the end of the slice where it fills in the value of esi with a pointer instead of it just being 0.

```
94433a:       mov    eax,DWORD PTR [ecx+0x4]    M@0x0039b57c[0x0202c5e8][4](R)
       T0     R@eax[0x00000014][4](W)    T0
```

```
944341:      mov    esi,eax      R@eax[0x0202c5e8][4](R)    T0     R@esi[0x0039b2d0]
[4](W) T0
944407:      mov    eax,esi      R@esi[0x0202c5e8][4](R)    T0     R@eax[0x00000000]
[4](W) T0
20803c44:    mov    esi,eax      R@eax[0x0202c5e8][4](R)    T0     R@esi[0x00000000]
[4](W) T0
20803c59:    mov    eax,esi      R@esi[0x0202c5e8][4](R)    T0     R@eax[0x0202c5e8]
[4](W) T0
20804cee:    mov    esi,eax      R@eax[0x0202c5e8][4](R)    T0     R@esi[0x00000000]
[4](W) T0
20804d01:    mov    eax,esi      R@esi[0x0202c5e8][4](R)    T0     R@eax[0x0202c5e8]
[4](W) T0
208f4a48:    mov    ecx,eax      R@eax[0x0202c5e8][4](R)    T0     R@ecx[0x00000001]
[4](W) T0
208f35f7:    mov    esi,ecx      R@ecx[0x0202c5e8][4](R)    T0     R@esi[0x00000000]
[4](W) T0
208f3610:    mov    eax,esi      R@esi[0x0202c5e8][4](R)    T0     R@eax[0x0202c5e8]
[4](W) T0
208f4ad9:    mov    esi,eax      R@eax[0x0202c5e8][4](R)    T0     R@esi[0x00000000]
[4](W) T0
208f4bbf:    mov    eax,esi      R@esi[0x0202c5e8][4](R)    T0     R@eax[0x20d6db64]
[4](W) T0
208f5c6b:    mov    ebx,eax      R@eax[0x0202c5e8][4](R)    T0     R@ebx[0x025cdf0c]
[4](W) T0
208f5c91:    mov    eax,ebx      R@ebx[0x0202c5e8][4](R)    T0     R@eax[0x00000000]
[4](W) T0
208f63f8:    mov    esi,eax      R@eax[0x0202c5e8][4](R)    T0     R@esi[0xc0000000]
[4](W) T0
208f648f:    mov    ecx,esi      R@esi[0x0202c5e8][4](R)    T0     R@ecx[0x00000000]
[4](W) T0
2088c9a2:    mov    esi,ecx      R@ecx[0x0202c5e8][4](R)    T0     R@esi[0x0202c5e8]
[4](W) T0
2088c9a4:    lea    ebx,[esi+0x10]      A@0x0202c5f8[0x00000000][4](R)    T0     R@ebx
[0xc0000000][4](W)  T0
2088c9a7:    mov    ecx,ebx      R@ebx[0x0202c5f8][4](R)    T0     R@ecx[0x0202c5e8]
[4](W) T0
```

It looks like esi is filled in at instruction 0x208f35f7, which happens to be at instruction 00266161. We can verify that this value is set in the large unaligned area near the end of the trace.

```
DISALIGNED @ T0:#00265965-00271264 (00005300 insts) ~ T1:#00272614-00272619 (00000006 insts)
```

So, ecx is initially set to 0. In the good file, execution proceeds down a branch which sets esi to a valid pointer. In the bad file, this value of esi is never set which leads to the null pointer dereference. Looking at the alignment, this divergence occurs at

```
$ trace_reader -intel -trace z.trace -first 00272612 -last 00272613
208f49b7:    cmp    WORD PTR [ebp-0x14],ax    R@ax[0x00000b11][2](R)    T0
      M@0x0012e3f4[0x00000b10][2](R)   T0
208f49bb:    jne    0x00000000208f4a5a  J@0x00000000[0x0000009f][4](R)   T0
$ trace_reader -intel -trace y.trace -first 00265963 -last 00265963
208f49b7:    cmp    WORD PTR [ebp-0x14],ax    R@ax[0x00000b10][2](R)    T0
      M@0x0012e3f4[0x00000b10][2](R)   T0
```

So the difference in execution comes from the value of 0b11 instead of 0b10 in ax. This is probably enough to fix the bug, but the value of ax could be sliced, and it could be continued in this fashion.

An exploitable Adobe Reader bug rated as unknown

This is a bug rated as unknown by !exploitable in the current version of Adobe Reader. It results from a one byte change in a compressed stream. It was found by mutation-based fuzzing.

This particular bug is really hard to reverse engineer with traditional methods because the function where it crashes gets called hundreds of times and if you set conditional breakpoints in the function that calls it, it doesn't crash anymore! This behavior must be based on timing or a race condition. Regardless, let's look at the crash.

```
$ trace_reader -trace z.trace -header | grep Number
Number of instructions: 243897288
$ trace_reader -intel -trace z.trace -first 243897288 -v
8074fc8:     cmp    WORD PTR [ecx+0x4],si     R@si[0x00007fff][2](R)     T0
      M@0x06fb1c70[0x00000000][2](R)    T0 ESP:  NUM_OP: 4 TID: 1660 TP: TPNone
EFLAGS: 0x00000083 CC_OP: 0x00000008 DF: 0x00000001 RAW: 0x66397104 MEMREGS:    R@ecx
[0x06fb1c6c][4](R)  T0
```

Notice that the memory at 0x6fb1c70 is actually unmapped, not 0. Looking at the assembly, four instructions later, there is a write (actually an add):

```
add [ecx], edi
```

So there is some hope of exploitation. Let's take a closer look at what caused this problem.

Let's slice on ecx whose address is wrongly dereferenced.

```
$ x86_slicer -in-trace z.trace -ctr 243897288 -regloc ecx > /dev/null
$ trace_reader -intel -trace ECX_0.trace | tail -2
8074fad:     add    DWORD PTR [eax],ecx R@ecx[0x037d8e2c][4](R)    T0     M@0x037d8e2c
[0x037d8e40][4](RW) T0
8074faf:     mov    ecx,DWORD PTR [eax] M@0x037d8e2c[0x06fb1c6c][4](R)    T0     R@ecx
[0x037d8e2c][4](W)  T0
```

So ecx came from eax which came from the addition of two heap pointers. That's probably not supposed to happen. Exploitation could happen if you imagine using a heap spray to put some interesting data in a region whose address is approximately twice a normal heap address.

Comparing the good vs. the bad file we see the byte that differs.

```
$ diff sheared.pdf.txt 210_a39c1ae51a2a5ea1f9ecccd70a0e9c8b.pdf.txt
2952c2952
< 0000b870  6a 68 05 df 3f c**f** f2 da  6c 57 59 71 ed 9a 21 99  |jh..?...lWYq..!.|
---
> 0000b870  6a 68 05 df 3f c**1** f2 da  6c 57 59 71 ed 9a 21 99  |jh..?...lWYq..!.|
```

The difference is a c1 vs a cf, so they only differ in one "nibble". Looking at the file, you can see that this is in some FlateDecode'd stream.

The crash occurs in CoolType.dll, which isn't surprising if you stare at the pdf, you'll see the flipped byte is in a font.

```
$ trace_reader -trace z.trace -header | grep 0x080
       Module: CoolType.dll @ 0x08000000 Size: 2486272
```

So, lets see when tainted data first enters the CoolType dll.

```
$ trace_reader -trace z.trace -taintedonly -count | grep ")80" | head
(01519331)808b015:  push   ecx    R@ecx[0x01e7546c][4](R)    T0     M@0x0012e950
[0xe49b0000][4](W)  T1 {8 ()()()(1111, 4) }
(01738319)8048170:  push   DWORD PTR [ebp-0x4] M@0x0012d99c[0x08083305][4](R)
      T0     M@0x0012d97c[0xe49b0000][4](W)    T1 {8 ()()()(1111, 4) }
(01754161)8015367:  movzx  eax,BYTE PTR [esi+0xb]     M@0x0012d4ef[0x000000e4][1]
(R)    T1 {1 (1111, 4) ()()()}    R@eax[0x0000009b][4](W)    T0
...
```

The first two are tainted data becoming untainted. The third is the first time it is really being used. Notice, the tainted byte is an e4. This isn't the original tainted cf, but that's not surprising since cf was in a compressed stream.

using alignment, we see in the good case at the same instruction,

```
$ ./aligned.pl y_z.aligned.txt T1:01754161
<T0:1753039> ~ T1:01754161
(T0:01750116-01753248 ~ T1:01751238-01754370)
(01753039)8015367:  movzx  eax,BYTE PTR [esi+0xb]     M@0x0012d4ef[0x000000d4][1]
(R)    T1 {1 (1111, 4) ()()()}    R@eax[0x0000009b][4](W)    T1 {1 (1111, 4) ()()()}
```

So instead of e4, its d4 in the good file.

It'd be nice to know where the e4/d4 came from. We'll have to look at the instructions before it entered CoolType.

```
$ trace_reader -intel -trace z.trace -first 01753161 -last 01754161 -taintedonly -v -
count

(01753389)7814507a: rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x02e21a78
[0xe49b0000][4](CR) T1 {8 ()()()(1111, 4) }    R@ecx[0x00000057][4](RCW)  T0
      M@0x0293860c[0xe49b0000][4](CW)   T1 {8 ()()()(1111, 4) } ESP:  NUM_OP: 5 TID:
1660 TP: TPSrc EFLAGS: 0x00000202 CC_OP: 0x00000010 DF: 0x00000001 RAW: 0xf3a5
MEMREGS:     R@edi[0x0293860c][4](R)    T0     R@esi[0x02e21a78][4](R)    T0
(01754027)7814507a: rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x0293860c
[0xe49b0000][4](CR) T1 {8 ()()()(1111, 4) }    R@ecx[0x0000000e][4](RCW)  T0
      M@0x0012d4ec[0xffc95f62][4](CW)   T0 ESP:  NUM_OP: 5 TID: 1660 TP: TPSrc EFLAGS:
0x00000202 CC_OP: 0x00000010 DF: 0x00000001 RAW: 0xf3a5 MEMREGS:   R@edi[0x0012d4ec]
[4](R) T0     R@esi[0x0293860c][4](R)    T0
(01754161)8015367:  movzx  eax,BYTE PTR [esi+0xb]     M@0x0012d4ef[0x000000e4][1]
(R)    T1 {1 (1111, 4) ()()()}    R@eax[0x0000009b][4](W)    T0 ESP:  NUM_OP: 4 TID:
1660 TP: TPSrc EFLAGS: 0x00000203 CC_OP: 0x00000008 DF: 0x00000001 RAW: 0x0fb6460b
MEMREGS:     R@esi[0x0012d4e4][4](R)    T0
```

So it looks like it was copied from the stack from 0x0293860c at instruction counter 01754027. Looking at that spot in memory the e4 isn't there anymore. Well, it was

copied to 0x0293860c from 0x02e21a78 in instruction counter 01753389. Lets look at that spot in memory:

```
$ state_reader -in-state z.trace.state -in-state-range 0x02e21a60:0x02e21a8f  -out-raw
z.state.dump.raw
$ hexdump -C z.state.dump.raw
00000000  6c 6f 63 61 01 99 84 e2  00 00 79 e8 00 00 1a 2c  |loca......y....,|
00000010  6d 61 78 70 0d 84 01 93  00 00 9b e4 00 00 00 20  |maxp........... |
00000020  6e 61 6d 65 8c 9d 3d 42  00 00 95 f8 00 00 05 dc  |name..=B........|
```

So there is the e4. Googling on the three strings "loca", "maxp", and "name", the first hit is True Type Font. This looks like a decompressed TTF. Looking to see what bytes are tainted, we see that only the e4 is.

```
$ state_reader -in-state z.trace.state -in-state-range 0x02e21a60:0x02e21a8f -out-text
z.state.dump.txt

$ grep -v "Taint: none" z.state.dump.txt
0x02e21a7b (0xe4) Taint: (4444, 1111, 4)
```

So the single nibble change in the compressed stream turned into a single nibble change in the decompressed font. The change is in the maxp header. We could try to start following the e4 as its taint propagates, but it gets pretty complicated. In fact, there are over 8 million instructions in the trace dealing with tainted data in CoolType.dll alone. Going backwards isn't much better.

So, we may as well use what we know at this point by looking at the TTF spec. Its not hard to track down in a font file where things like the above occur. Namely, its a list of table directory entries, see [TTF]. In particular, the e4 is in the table directory broken out as:

```
00000010  6d 61 78 70 0d 84 01 93  00 00 9b e4 00 00 00 20  |maxp........... |
```

Field	Data
Identifier	“maxp”
Checksum	0x0d840193
Offset from beginning of sfnt	0x9b**e4**
Length of table	0x20

So it appears that the location of this maxp table has been changed in the bad file (and that they don't check the checksum). Let’s grab the whole decompressed font.

```
$ state_reader -in-state z.trace.state -in-state-range 0x2e219d4:0x2e2b5ff -out-raw
bad.ttf
```

Doing some pain staking parsing of this font file, we find where the maxp should be and where it is in the bad file:

```
$ hexdump -C bad.ttf | tail
00009ba0  00 74 00 20 00 26 00 20  00 54 00 4d 00 20 00 4f  |.t. .&. .T.M. .O|
00009bb0  00 66 00 66 00 2e 00 20  00 61 00 6e 00 64 00 20  |.f.f... .a.n.d. |
00009bc0  00 65 00 6c 00 73 00 65  00 77 00 68 00 65 00 72  |.e.l.s.e.w.h.e.r|
00009bd0  00 65 00 2e 00 01 00 00  06 8a 00 f2 00 3c 00 6f  |.e...........<.o|
00009be0  00 06 00 02 00 10 00 2f  00 55 00 00 06 4e ff ff  |......./.U...N..|
00009bf0  00 03 00 02 00 01 00 00  00 02 e6 67 f1 c1 1d f9  |...........g....|
00009c00  5f 0f 3c f5 08 19 08 00  00 00 00 00 a2 e3 3c 1d  |_.<...........<.|
00009c10  00 00 00 00 b9 d5 b5 13  fa fa fc fd 10 00 08 15  |................|
00009c20  00 00 00 09 00 01 00 01  00 00 00 00              |............|
```

For the good file (i.e. where a d4 is in the offset), the maxp table starts at 0x9bd4. For the bad file, the maxp table starts at 0x9be4. Looking at this table you get

Field	Good table	Bad table
Version	0x00010000	0x0010002f
numGlyphs	0x6a8a	0x0055
maxPoints	0x00f2	0x0000
maxContours	0x003c	0x06fe
maxComponentPoints	0x006f	**0xffff**
maxComponentContours	0x0006	0x0003
maxZones	0x0002	0x0002
maxTwilightPoints	0x0010	0x0001
maxStorage	0x002f	0x0000
maxFunctionDefs	0x0055	0x0002
maxInstructionDefs	0x0000	0xe667
maxStackElements	0x06fe	0xf1c1
maxSizeOfInstructions	0xffff	0x1df9
maxComponentElements	0x0003	0x5f0f
maxComponentDepth	0x0002	0x3cf5

Its not hard to imagine it is the 0xffff in the MaxComponentPoints field that is causing the problem. You can verify this by changing it to some value smaller than 0xfff0 and

seeing that it doesn't cause a crash. Alternatively, changing any of the other fields doesn't cause a crash. One other interesting thing is if you fuzz the actual maxp table itself in an otherwise clean file, you won't find this crash with single byte mutations, it can only be found by fuzzing words, not bytes.

So, let's create a PDF file with this font and instead of tainting the e4, which is the offset to where the table is found, we taint the MaxPoints field (The 0xffff).

Running it generates more traces. Looking to see where tainted data enters Cooltype.dll, we do the same as before:

```
$ trace_reader -intel -trace x.trace -taintedonly -count | grep ")80" | head -20
(79500371)80112f3:  mov    dh,BYTE PTR [eax+0xa]      M@0x0267ae86[0x000000ff][1]
(R)    T1 {1 (1111, 6) ()()()}    R@dh[0x00000000][1](W)     T0
(79500372)80112f6:  mov    dl,BYTE PTR [eax+0xb]      M@0x0267ae87[0x000000ff][1]
(R)    T1 {1 (1111, 7) ()()()}    R@dl[0x00000000][1](W)     T0
(79500373)80112f9:  mov    WORD PTR [ecx+0xa],dx      R@dx[0x0000ffff][2](R)     T1 {3
(1111, 7) (1111, 6) ()()}  M@0x01e7592e[0x00000000][2](W)    T0
(79500374)80112fd:  xor    edx,edx      R@edx[0x0000ffff][4](R)    T1 {3 (1111, 7)
(1111, 6) ()()}     R@edx[0x0000ffff][4](RW)   T1 {3 (1111, 7) (1111, 6) ()()}
(79501343)80053e7:  movzx  edx,WORD PTR [ebx+0xa]     M@0x01e7592e[0x0000ffff][2]
(R)    T1 {3 (1111, 7) (1111, 6) ()()}  R@edx[0x000027fc][4](W)    T0
(79501344)80053eb:  cmp    ax,dx  R@dx[0x0000ffff][2](R)     T1 {3 (1111, 7) (1111, 6)
()()}  R@ax[0x000000f2][2](R)     T0
(79501347)80053f3:  movzx  eax,dxR@dx[0x0000ffff][2](R)     T1 {3 (1111, 7) (1111, 6)
()()}  R@eax[0x000000f2][4](W)    T0
(79501350)80053fa:  lea    edx,[ebp+0xc]       A@0x0012d618[0x00000000][4](R)
       T0     R@edx[0x0000ffff][4](W)    T1 {3 (1111, 7) (1111, 6) ()()}
(79501355)8005403:  add    eax,0x8      I@0x00000000[0x00000008][1](R)    T0     R@eax
[0x0000ffff][4](RW) T1 {3 (1111, 7) (1111, 6) ()()}
(79501356)8005406:  push   eax    R@eax[0x00010007][4](R)    T1 {15 (1111, 7) (1111, 7)
(1111, 7) (1111, 7) }      M@0x0012d5ec[0x01e759c0][4](W)    T0
(79501360)8004745:  mov    eax,DWORD PTR [ebp+0x14]  M@0x0012d5f8[0x0012d618][4]
(R)    T0     R@eax[0x00010007][4](W)    T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111,
7) }
(79501365)8004751:  movzx  esi,WORD PTR [ebp+0x8]     M@0x0012d5ec[0x00000007][2]
(R)    T1 {3 (1111, 7) (1111, 7) ()()}  R@esi[0x01e75968][4](W)    T0
...
```

So our 0xffff is loaded from memory and 8 is added to it. This new value, 0x10007 is then pushed onto the stack (for a function call). You can see more details if you look at the non tainted instructions around this point.

```
$ trace_reader -intel -trace x.trace -first 79501356 -last 79501359 -count
(79501356)8005406:  push   eax    R@eax[0x00010007][4](R)    T1 {15 (1111, 7) (1111, 7)
(1111, 7) (1111, 7) }      M@0x0012d5ec[0x01e759c0][4](W)    T0
(79501357)8005407:  call   0x0000000008004742  J@0x00000000[0xfffff33b][4](R)
       T0     M@0x0012d5e8[0x000029b8][4](W)T0
(79501358)8004742:  push   ebp    R@ebp[0x0012d60c][4](R)    T0     M@0x0012d5e4
[0x0012d614][4](W)  T0
(79501359)8004743:  mov    ebp,esp      R@esp[0x0012d5e4][4](R)    T0     R@ebp
[0x0012d60c][4](W)  T0
```

Looking at instruction counter 79501365 above, we see this memory location holding 0x10007 is read in as a word, so a (short) integer overflow has occurred since esi now

has the value 7 which is probably smaller than intended. Let's peek ahead and see what problems this causes.

A bit further down the trace, we see the following:

```
(79501385)8004786:  mov    edx,esi      R@esi[0x00000007][4](R)    T1 {3 (1111, 7)
(1111, 7) ()()}     R@edx[0x00000134][4](W)    T1 {15 (1111, 7) (1111, 7) (1111, 7)
(1111, 7) }
(79501386)8004788:  shl    edx,0x2      I@0x00000000[0x00000002][1](R)    T0     R@edx
[0x00000007][4](RW) T1 {3 (1111, 7) (1111, 7) ()()}
```

so that edx = 0x1c. Then a bunch of instructions like:

```
(79501397)80047a3:  add    DWORD PTR [eax],edx R@edx[0x0000001c][4](R)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }      M@0x0012d618[0x0000016c][4](RW)  T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }
(79501398)80047a5:  mov    edi,DWORD PTR [eax] M@0x0012d618[0x00000188][4](R)    T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }      R@edi[0x0000016c][4](W)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }
(79501399)80047a7:  mov    DWORD PTR [ecx+0xc],edi    R@edi[0x00000188][4](R)    T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }      M@0x01e75988[0x00000000][4](W)    T0
```

What's going on here is a some value is being incremented by 0x1c, and being stored into different spots in memory pointed to by ecx (in this case, 0xc bytes into this buffer). We can see what this buffer looks like in memory (it is all tainted).

```
$ state_reader -in-state x.trace.state -in-state-range 0x01e7597c:0x01e75993 -out-
raw /tmp/foo
$ hexdump -C /tmp/foo
00000000  34 01 00 00 50 01 00 00  6c 01 00 00 88 01 00 00  |4...P...l.......|
00000010  a4 01 00 00 c0 01 00 00                           |........|
```

So there are some small values which are incremented by a number that is the result of an integer overflow. Looking ahead in the trace to see where these values are used (for example, seeing the next occurrence of 0x134 anywhere) and you come to a series of instructions like:

```
(79505865)80048ae:  mov    esi,DWORD PTR [ecx+0x14]  M@0x01e7597c[0x00000134][4]
(R)    T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }      R@esi[0x0012d61c][4]
(W)    T0
(79505866)80048b1:  add    esi,edx      R@edx[0x08235ab8][4](R)    T0     R@esi
[0x00000134][4](RW) T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79505867)80048b3:  mov    DWORD PTR [eax],esi R@esi[0x08235bec][4](R)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }      M@0x08235a28[0x00000000][4](W)    T0
(79505868)80048b5:  mov    esi,DWORD PTR [ecx+0x18]  M@0x01e75980[0x00000150][4]
(R)    T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }      R@esi[0x08235bec][4]
(W)    T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79505869)80048b8:  add    esi,edx      R@edx[0x08235ab8][4](R)    T0     R@esi
[0x00000150][4](RW) T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79505870)80048ba:  mov    DWORD PTR [eax+0x4],esi   R@esi[0x08235c08][4](R)    T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }      M@0x08235a2c[0x00000000][4](W)    T0
```

These are not very far into the reduced tainted-only-in-Cooltype trace, still only at about the 125th instruction. These instructions are adding the small numbers from the buffer above to a fixed value in edx, 0x08235ab8, and saving these pointers in a buffer pointed to by eax beginning at 0x08235a28.

Extracting what this eax buffer looks like reveals a series of 11 pointers pointing to the .data section of Cooltype.dll. You could tell there were 11 by seeing the offsets used in the writes in the instructions.

```
$ state_reader -in-state x.trace.state -in-state-range 0x08235a28:0x08235a53 -out-
raw /tmp/foo
$ hexdump -C /tmp/foo
00000000  ec 5b 23 08 08 5c 23 08  24 5c 23 08 40 5c 23 08  |.[#..\#.$\#.@\#.|
00000010  5c 5c 23 08 78 5c 23 08  b8 5a 23 08 c0 5a 23 08  |\\#.x\#..Z#..Z#.|
00000020  38 5b 23 08 94 5c 23 08  b0 5b 23 08              |8[#..\#..[#.|
```

By the way these pointers were constructed, they have the property that they point to buffers of size 0x1c, or more accurately, they differ by 0x1c. This is probably smaller than intended due to the integer overflow. For comparison in the good file which doesn't have the integer overflow, you get:

```
00000000  e0 5c 23 08 c8 60 23 08  b0 64 23 08 98 68 23 08  |.\#..`#..d#..h#.|
00000010  80 6c 23 08 68 70 23 08  b8 5a 23 08 b4 5b 23 08  |.l#.hp#..Z#..[#.|
00000020  2c 5c 23 08 50 74 23 08  a4 5c 23 08              |,\#.Pt#..\#.|
```

so that the buffer sizes here are 0x3e8. It turns out that these pointers are indeed supposed to point to independent buffers, as can be seen by observing the way the good file treats them. Not surprisingly, in the bad file case, the buffers are smaller than expected and the program tends to write over the top of neighboring buffers. It is not hard to find these spots, as almost every buffer access is an overflow:

```
(79737405)800c4ef:  mov    ebx,DWORD PTR [edx] M@0x08235bec[0x00003a79][4](R)    T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }      R@ebx[0x0012d54c][4](W)    T0
(79737406)800c4f1:  sub    edi,0x4      I@0x00000000[0x00000004][1](R)    T0     R@edi
[0x08235c0c][4](RW) T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79737407)800c4f4:  mov    DWORD PTR [ebp-0x14],ebx  R@ebx[0x00003a79][4](R)    T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }      M@0x0012d4cc[0x08235c08][4](W)    T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79737408)800c4f7:  mov    ebx,DWORD PTR [edi] M@0x08235c08[0x00000000][4](R)    T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }      R@ebx[0x00003a79][4](W)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }
(79737409)800c4f9:  mov    DWORD PTR [edx],ebx R@ebx[0x00000000][4](R)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }      M@0x08235bec[0x00003a79][4](W)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }
(79737410)800c4fb:  mov    ebx,DWORD PTR [ebp-0x14]  M@0x0012d4cc[0x00003a79][4]
(R)    T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }      R@ebx[0x00000000][4]
(W)    T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79737411)800c4fe:  add    edx,0x4      I@0x00000000[0x00000004][1](R)    T0     R@edx
[0x08235bec][4](RW) T1 {15 (1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79737412)800c501:  dec    DWORD PTR [ebp-0x8] M@0x0012d4d8[0x00000003][4](RW)  T1 {15
(1111, 7) (1111, 7) (1111, 7) (1111, 7) }
(79737413)800c504:  mov    DWORD PTR [edi],ebx R@ebx[0x00003a79][4](R)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }      M@0x08235c08[0x00000000][4](W)    T1 {15 (1111,
7) (1111, 7) (1111, 7) (1111, 7) }
(79737414)800c506:  jns    0x000000000800c4ef  J@0x00000000[0xfffffff9][4](R)    T0
```

Here, the program is trying to shuffle values around inside the 0x08235bec buffer, but its already writing to 0x08235c08 which should be in the second buffer. But what kind of data do you find in these buffers? Spot checking them reveals that they always seem to

be words or pointers back into this "scratch buffer" in the .data section, but occasionally contain "random" looking data:

```
$ state_reader -in-state x.trace.state -in-state-range 0x8235bec:0x8235dec -out-raw /
tmp/foo
$ hexdump -C /tmp/foo
00000000  64 62 00 00 16 40 00 00  16 40 00 00 4a 2d 00 00  |db...@...@..J-..|
00000010  c7 13 00 00 c7 13 00 00  34 21 00 00 00 00 00 00  |........4!......|
00000020  00 00 00 00 46 81 00 00  00 00 00 00 0a 8b 00 00  |....F...........|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 c7 13 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 02 b6 00 00  |................|
00000050  63 cb ff ff 00 00 00 00  00 00 00 00 02 b6 00 00  |c...............|
00000060  b0 b3 00 00 0a 8b 00 00  00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 c7 13 00 00  00 00 00 00 00 00 00 00  |................|
00000080  00 00 00 00 02 b6 00 00  63 cb ff ff 00 00 00 00  |........c.......|
00000090  00 00 00 00 02 b6 00 00  b0 b3 00 00 73 04 00 00  |............s...|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 a2 00 00 00  |................|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 d3 05 00 00  |................|
000000c0  51 fe ff ff 00 00 00 00  00 00 00 00 d3 05 00 00  |Q...............|
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000e0  ba dd 67 02 00 00 00 00  14 00 a2 00 00 00 26 03  |..g...........&.|
000000f0  c0 05 73 04 82 07 a2 00  13 00 00 00 02 00 00 00  |..s.............|
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00  |................|
```

This explains why we saw the two pointers being added in the crash, it was probably supposed to be a pointer added to a small (word) offset, but instead got two pointers.

As is, by doing a heap spray, we can get arbitrary values in our heap buffers modified (added really). By ensuring this is a vtable or something, we can get control of execution.

An exploitable Open Office bug

This is a crash !exploitable says is exploitable in Open Office Simpress that was found by mutation based fuzzing. The bad file and good file differ by a single byte. We turn on full trace heap checking for this example. The crash is a write inside memcpy:

```
$ trace_reader -trace z.trace -header | grep Number
Number of instructions: 2895297
$ trace_reader -intel -trace z.trace -first 2895297 -v
7855ac5a:     mov     BYTE PTR [edi],al   R@al[0x000000 6d ][1](R)     T0     M@0x1244c000
[0x00000000][1](W)  T0 ESP:  NUM_OP: 3 TID: 1224 TP: TPNone EFLAGS: 0x00000083 CC_OP:
0x00000010 DF: 0x00000001 RAW: 0x8807 MEMREGS:       R@ edi[0x1244c000] [4](R)    T0
```

The first thing we need to do is figure out where the memcpy was called. As before, we do this by seeing the last non-Windows DLL instructions executed before the crash.

```
$ trace_reader -intel -trace z.trace -first 2894097 -count | grep -v ")78" | tail -2
(02895270) 5732c511: call   0x0000000057349d30  J@0x00000000[0x0001d81f][4](R)
      T0     M@0x014aea8c[0x5732c516][4](W)    T0
(02895271)57349d30: jmp    DWORD PTR ds:0x573511d4   M@0x573511d4[ 0x7855aac0 ][4]
(R)    T0
```

The call to memcpy occurred at 0x5732c511, which is inside tlmi.dll.

```
$ trace_reader -trace z.trace -header | grep 0x5730
```

```
        Module: tlmi.dll @ 0x57300000 Size: 528384
```

Currently, the only way we know that the call to 0x7855aac0 is a call to memcpy is by checking with IDA Pro. So, lets see what the arguments were to memcpy.

```
$ trace_reader -intel -trace z.trace -count -first 02895266 -last 02895270
(02895266)5732c50a: push   edi    R@edi[0x00000002][4](R)    T0     M@0x014aea98
[0x00000002][4](W)  T0
(02895267)5732c50b: add    eax,ecx      R@ecx[0x00000070][4](R)    T0     R@eax
[0x12419e18][4](RW) T0
(02895268)5732c50d: push   eax    R@eax[0x12419e88][4](R)    T0     M@0x014aea94
[0x12419e86][4](W)  T0
(02895269)5732c50e: push   DWORD PTR [ebp+0x8] M@0x014aeaac[0x1244c000][4](R)
       T0     M@0x014aea90[0x1244bffe][4](W)    T0
(02895270)5732c511: call   0x0000000057349d30  J@0x00000000[0x0001d81f][4](R)
       T0     M@0x014aea8c[0x5732c516][4](W)    T0
```

So the final call to memcpy which crashes took the form:

```
memcpy(0x1244c000,0x12419e88,2);
```

If we examine the source buffer to this memcpy, we see that we control the source, i.e. it comes from the file.

```
$ state_reader -in-state z.trace.state -in-state-range 0x12419e78:0x12419e9f -out-
raw /tmp/foo
$ hexdump -C /tmp/foo
00000000  5f 00 41 00 75 00 74 00  68 00 6f 00 72 00 45 00  |_.A.u.t.h.o.r.E.|
00000010  6d 00 61 00 69 00 6c 00  00 00 00 00 05 00 00 00  |m.a.i.l.........|
00000020  18 00 00 00 5f 00 41 00                           |...._.A.|
```

Let's look at the destination buffer.

```
$ alloc_reader -alloc z.trace.alloc -addr 0x1244c000 -closest 1 2>/dev/null
Found 0 buffers. Closest are:
  [02890556,end] 0x1244bfb8 (67) Dist: 6
```

The closest buffer began at 0x1244bfb8 and ended at 0x1244bffb. So this write was occurred just beyond a buffer. Probably this is a 2 byte heap overflow. Let's see where the buffer that we are overflowing was allocated. We do this by tracing beyond when the buffer was returned (instruction counter 02890556) paying attention to returns until we end up back in OpenOffice DLLs.

```
$ trace_reader -intel -trace z.trace -count -first 02890556 | grep -A1 ret | head -8
(02890556)7c9101bb: ret    0xc    I@0x00000000[0x0000000c][2](R)    T0     M@0x014aeaa0
[0x78583a58][4](R)  T0
(02890557)78583a58: mov    edi,eax      R@eax[0x1244bfb8][4](R)    T0     R@edi
[0x5732df90][4](W)  T0
--
(02890566)78583ab5: ret          M@0x014aeac0[0x78583b58][4](R)    T0
(02890567)78583b58: pop    ecx    M@0x014aeac4[0x00000043][4](R)    T0     R@ecx
[0x7c9101bb][4](W)  T0
--
(02890571)78583b5e: ret          M@0x014aead8[0x5f38b211][4](R)    T0
(02890572)5f38b211: mov    edi,eax      R@eax[0x1244bfb8][4](R)    T0     R@edi
[0x5732df90][4](W)  T0
```

So the first time outside of Windows dll's that it returns after the allocation is 0x5f38b211. We can quickly find the instruction which made this call and the size of the requested allocation:

```
(02877747)5f38b20b: push   eax    R@eax[0x00000043][4](R)    T0     M@0x014aeadc
[0x5f38b205][4](W)  T0
(02877748)5f38b20c: call   0x000000005f496ddc  J@0x00000000[0x0010bbd0][4](R)
       T0     M@0x014aead8[0x5730ff79][4](W)    T0
```

Looking again in IDA Pro we see this is actually a call to new(0x43). So the destination buffer was allocated at 0x5f38b20c to have size 0x43.

As an aside, let's see where the tainted byte (0x71) is used in the program.

```
$ trace_reader -intel -taintedonly -trace z.trace -count | grep -v ")78" | tail -2
(02814669)5f38b1bb: cmp    DWORD PTR [ebp-0x18],ebx   R@ebx[0x00000000][4](R)
       T0     M@0x014aeb7c[0x00000071][4](R)    T1 {1 (1111, 1) ()()()}
(02877474)5f38b2fc: cmp    eax,DWORD PTR [ebp-0x18]   R@eax[0x0000000b][4](R)
       T0     M@0x014aeb7c[0x00000071][4](R)    T1 {1 (1111, 1) ()()()}
```

These two addresses are in the same function where the function new() is called above so we know we have found the function of interest for this bug.

Let us see where the size of the allocation, 0x43, came from.

```
$ x86_slicer -in-trace z.trace -ctr 02877747 -regloc eax > /dev/null
$ trace_reader -intel -trace EAX_0.trace | tail -6
7855ab1a:     rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x123e9e34[0x00690076]
[4](CR)       T0     R@ecx[0x00000072][4](RCW)  T0M@0x12419e34[0xc0c0c0c0][4](CW) T0
7855ab1a:     rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x123e9e38[0x00770065]
[4](CR)       T0     R@ecx[0x00000071][4](RCW)  T0M@0x12419e38[0xc0c0c0c0][4](CW) T0
7855ab1a:     rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x123e9e3c[0x00790043]
[4](CR)       T0     R@ecx[0x00000070][4](RCW)  T0M@0x12419e3c[0xc0c0c0c0][4](CW) T0
7855ac18:     mov    eax,DWORD PTR [esi+ecx*4-0x4]    M@0x12419e3c[0x00790043][4]
(R)    T0     R@eax[0x12419e40][4](W)    T0
7855ac1c:     mov    DWORD PTR [edi+ecx*4-0x4],eax    R@eax[0x00790043][4](R)
       T0     M@0x014aeb8c[0x0000000a][4](W)    T0
5f38b207:     movzx  eax,WORD PTR [ebp-0x8]    M@0x014aeb8c[0x00000043][2](R)
       T0     R@eax[0x00000028][4](W)    T0
```

(Notice the 0x00000043 in 0x5f38b207 is really still 0x00790043 but that address is being presented as a word pointer since that is how it is used in that instruction.) Looking at this slice, we see that the value 0x43 came from a word from the file (you can see the file being copied in the rep movs instructions: 006900760077006500790043). So we control the word that is used for the allocation as well as the data being copied. Now how much data is copied? We saw that the actual call to memcpy only had length 2, so the question is how much data is really being copied?

Let's look at the instructions executed in the DLL from the time of the allocation until the crash.

```
$ trace_reader -intel -trace z.trace -first 02890572 -count | grep ")5f" | tail
(02895097)5f38b25c: call   DWORD PTR ds:0x5f5032c0   M@0x5f5032c0[0x5732df24][4]
(R)    T0     M@0x014aead8[0x5f38b262][4](W)    T0
(02895213)5f38b262: inc    DWORD PTR [ebp-0xc] M@0x014aeb88[0x00000023][4](RW)   T0
(02895214)5f38b265: movzx  eax,WORD PTR [ebp-0x8]    M@0x014aeb8c[0x00008021][2]
(R)    T0     R@eax[0x014aeaec][4](W)    T0
(02895215)5f38b269: inc    ebx    R@ebx[0x1244bffe][4](RW)   T0
(02895216)5f38b26a: inc    ebx    R@ebx[0x1244bfff][4](RW)   T0
(02895217)5f38b26b: cmp    DWORD PTR [ebp-0xc],eax   R@eax[0x00008021][4](R)
       T0     M@0x014aeb88[0x00000024][4](R)    T0
(02895218)5f38b26e: jb     0x000000005f38b255  J@0x00000000[0xfffffffe7][4](R)    T0
(02895219)5f38b255: push   ebx    R@ebx[0x1244c000][4](R)    T0     M@0x014aeadc
[0x1244bffe][4](W)  T0
(02895220)5f38b256: lea    ecx,[ebp-0xa8]      A@0x014aeaec[0x00000000][4](R)
       T0     R@ecx[0xffff0000][4](W)    T0
(02895221)5f38b25c: call   DWORD PTR ds:0x5f5032c0   M@0x5f5032c0[0x5732df24][4]
(R)    T0     M@0x014aead8[0x5f38b262][4](W)    T0
```

It looks like the execution ends up in a loop calling a function from tlmi.dll each time (where the bad memcpy call occurs). A little bit of analysis shows that each time the function from tlmi.dll is called, 2 bytes are copied into our buffer of size 0x43. To see this, observe each time through the loop, the call to memcpy occurs exactly once and the argument pushed for length of the copy is 2:

```
$ trace_reader -intel -trace z.trace -first 02890739 -count | grep -P "5f38b25c:|
5732c511:|5732c50a:"
(02890739)5f38b25c: call   DWORD PTR ds:0x5f5032c0   M@0x5f5032c0[0x5732df24][4]
(R)    T0     M@0x014aead8[0x5f38b247][4](W)    T0
(02890784)5732c50a: push   edi    R@edi[0x00000002][4](R)    T0     M@0x014aea98
[0x00000043][4](W)  T0
(02890788)5732c511: call   0x0000000057349d30  J@0x00000000[0x0001d81f][4](R)
       T0     M@0x014aea8c[0x014aef84][4](W)    T0
(02890864)5f38b25c: call   DWORD PTR ds:0x5f5032c0   M@0x5f5032c0[0x5732df24][4]
(R)    T0     M@0x014aead8[0x5f38b262][4](W)    T0
(02890909)5732c50a: push   edi    R@edi[0x00000002][4](R)    T0     M@0x014aea98
[0x00000002][4](W)  T0
(02890913)5732c511: call   0x0000000057349d30  J@0x00000000[0x0001d81f][4](R)
       T0     M@0x014aea8c[0x5732c516][4](W)    T0
(02890988)5f38b25c: call   DWORD PTR ds:0x5f5032c0   M@0x5f5032c0[0x5732df24][4]
(R)    T0     M@0x014aead8[0x5f38b262][4](W)    T0
(02891033)5732c50a: push   edi    R@edi[0x00000002][4](R)    T0     M@0x014aea98
[0x00000002][4](W)  T0
(02891037)5732c511: call   0x0000000057349d30  J@0x00000000[0x0001d81f][4](R)
       T0     M@0x014aea8c[0x5732c516][4](W)    T0
(02891113)5f38b25c: call   DWORD PTR ds:0x5f5032c0   M@0x5f5032c0[0x5732df24][4]
(R)    T0     M@0x014aead8[0x5f38b262][4](W)    T0
(02891158)5732c50a: push   edi    R@edi[0x00000002][4](R)    T0     M@0x014aea98
[0x00000002][4](W)  T0
(02891162)5732c511: call   0x0000000057349d30  J@0x00000000[0x0001d81f][4](R)
       T0     M@0x014aea8c[0x5732c516][4](W)    T0
(02891237)5f38b25c: call   DWORD PTR ds:0x5f5032c0   M@0x5f5032c0[0x5732df24][4]
(R)    T0     M@0x014aead8[0x5f38b262][4](W)    T0
(02891282)5732c50a: push   edi    R@edi[0x00000002][4](R)    T0     M@0x014aea98
[0x00000002][4](W)  T0
(02891286)5732c511: call   0x0000000057349d30  J@0x00000000[0x0001d81f][4](R)
       T0     M@0x014aea8c[0x5732c516][4](W)    T0
...
```

The only question that remains is how many times this loop occurs. Looking at cmp that determines this,

```
(02895217)5f38b26b: cmp    DWORD PTR [ebp-0xc],eax    R@eax[0x00008021][4](R)
       T0     M@0x014aeb88[0x00000024][4](R)    T0
```

we see the loop will happen 0x8021 times. That means 0x10042 bytes will be copied into the buffer of size 0x43, way too much! Where did this 0x8021 value come from?

```
$ x86_slicer -in-trace z.trace -ctr 02895217 -regloc eax > /dev/null
$ trace_reader -intel -trace EAX_0.trace | tail -5
7855ab1a:     rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x123e9e3c[0x00790043]
[4](CR)       T0     R@ecx[0x00000070][4](RCW)  T0M@0x12419e3c[0xc0c0c0c0][4](CW) T0
7855ac18:     mov    eax,DWORD PTR [esi+ecx*4-0x4]    M@0x12419e3c[0x00790043][4]
(R)    T0     R@eax[0x12419e40][4](W)    T0
7855ac1c:     mov    DWORD PTR [edi+ecx*4-0x4],eax    R@eax[0x00790043][4](R)
       T0     M@0x014aeb8c[0x0000000a][4](W)    T0
5f38b237:     shr    DWORD PTR [ebp-0x8],1      I@0x00000000[0x00000001][1](R)
       T0     M@0x014aeb8c[0x00790043][4](RW)   T0
5f38b265:     movzx  eax,WORD PTR [ebp-0x8]    M@0x014aeb8c[0x00008021][2](R)
       T0     R@eax[0x014aeaec][4](W)    T0
```

Oh, it came from the dword we supplied divided by 2. However, they treated the allocation as a word, but the number they are dividing by two they treat as a dword. So, in this case, the copy is much larger than the buffer allocated and we control the data being copied. This is a very nice heap overflow.

Another exploitable Open Office bug

This is an exploitable crash open Open Office Simpress found by mutation based fuzzing. The good file and bad file differ in a single byte. The crash occurs as follows:

```
$ trace_reader -trace z.trace -header | grep Number
Number of instructions: 209430179
$ trace_reader -intel -v -trace z.trace -first 209430179
7855ab1a:     rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi] M@0x072602c0[0x00001bcf]
[4](CR)       T0     R@ecx[0x0001fee0][4](RCW)  T1 {15 (1111, 3) (1111, 3) (1111, 3)
(1111, 3) }  M@0x07360000[0x00000000][4](CW)   T0 ESP:  NUM_OP: 5 TID: 1380 TP: TPSrc
EFLAGS: 0x00000083 CC_OP: 0x00000010 DF: 0x00000001 RAW: 0xf3a5 MEMREGS:  R@edi
[0x07360000][4](R)  T1 {15 (1111, 3) (1111, 3) (1111, 3) (1111, 3) }      R@esi
[0x072602c0][4](R)  T0
```

This is in a call to memcpy. As an example of how we can use valset analysis, we can automatically see if the attacker has control over the destination address of the copy.

```
$ x86_slicer -in-trace z.trace -ctr 209430179 -regloc edi > /dev/null
$ cp EDI_0.trace z-sliced.trace
$ dynslicer -concall -nomemconstraints -ir z-sliced.trace
$ valset_ir -ir-in z-sliced.ir.concall -tmp-name R_EDI_3 -get-val-bounds true -sample-
pts 2000
...
Summary:
Lowest and highest possible values: 0x72e0020 to 0x7360018
Influence bounds after establishing value bounds: 3.321928 to 18.999981
Influence bounds after asking for counterexamples: 6.209453 to 18.999981
Probable influence: 16.012888 (252 hits of 2000 samples in population 524207)
```

It seems the attacker has control over 2 bytes of the destination address. As we proceed with analysis, we'll see why this is the case.

The instructions that called this memcpy can be found by looking for the last instructions not inside Windows DLL's.

```
$ trace_reader -intel -trace z.trace -first 209420079 -last 209430179 -count | grep -v
")78" | tail
(209429976)57308071:       mov    eax,DWORD PTR [ebp+0xc]    M@0x014ad988[0x0000ffc4]
[4](R) T1 {3 (1111, 3) (1111, 3) ()()}  R@eax[0x072eff6c][4](W)    T1 {3 (1111, 3)
(1111, 3) ()()}
(209429977)57308074:       shl    eax,0x3      I@0x00000000[0x00000003][1](R)
      T0     R@eax[0x0000ffc4][4](RW)   T1 {3 (1111, 3) (1111, 3) ()()}
(209429978)57308077:       push   eax    R@eax[0x0007fe20][4](R)    T1 {15 (1111, 3)
(1111, 3) (1111, 3) (1111, 3) }  M@0x014ad960[0x00000001][4](W)    T0
(209429979)57308078:       mov    eax,DWORD PTR [esi] M@0x08a34718[0x072e0020][4]
(R)    T0     R@eax[0x0007fe20][4](W)    T1 {15 (1111, 3) (1111, 3) (1111, 3) (1111,
3) }
(209429980)5730807a:       push   DWORD PTR [edi]     M@0x08a348c8[0x07260020][4]
(R)    T0     M@0x014ad95c[0x0000ff6c][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
(209429981)5730807c:       movzx  ebx,bx R@bx[0x0000ffa8][2](R)     T1 {3 (1111, 3)
(1111, 3) ()()}     R@ebx[0x0000ffa8][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
(209429982)5730807f:       lea    eax,[eax+ebx*8]     A@0x0735fd60[0x00000000][4]
(R)    T0     R@eax[0x072e0020][4](W)    T0
(209429983)57308082:       push   eax    R@eax[0x0735fd60][4](R)    T1 {15 (1111, 3)
(1111, 3) (1111, 3) (1111, 3) }  M@0x014ad958[0x57308066][4](W)    T0
(209429984)57308083:       call   0x0000000057349d30  J@0x00000000[0x00041cad][4]
(R)    T0     M@0x014ad954[0x014ad97c][4](W)    T0
(209429985)57349d30:       jmp    DWORD PTR ds:0x573511d4    M@0x573511d4[0x7855aac0]
[4](R) T0
```

From this it seems the call to memcpy looked something like

```
memcpy(0x0735fd60, 0x07260020, 0x0007fe20);
```

Let's see where these buffers came from. The source buffer can be traced back to its allocation point using alloc_reader.

```
$ ulimit -s unlimited
$ alloc_reader -alloc z.trace.alloc -ctr 209430179 -addr 0x07260020 -closest 1
Found 1 buffers
[141406597,end] 0x07260020 (523808)
```

So the source was allocated at instruction counter 141406597 and had a size of 0x7fe20 (523808), which happens to be the same size as the length in the memcpy.

Now we examine the destination buffer.

```
$ alloc_reader -alloc z.trace.alloc -ctr 209429984 -addr 0x0735fd60 -closest 1
Found 0 buffers. Closest are:
  [209160625,end] 0x072e0020 (523104) Dist: 481
```

So the closest buffer begins at 0x072e0020 and ends at 0x735fb80, a couple of hundred bytes before where the memcpy is going to write. If you look at the instruction

before the destination is pushed on the stack (already displayed above), it says how the destination of the memcpy is computed.

```
$ trace_reader -intel -trace z.trace -first 209429982 -last 209429982 -count -v
(209429982)5730807f:       lea    eax,[eax+ebx*8]     A@0x0735fd60[0x00000000][4]
(R)    T0     R@eax[0x072e0020][4](W)    T0 ESP:  NUM_OP: 5 TID: 1380 TP: TPSrc EFLAGS:
0x00000202 CC_OP: 0x00000024 DF: 0x00000001 RAW: 0x8d04d8 MEMREGS:        R@eax
[0x072e0020][4](R)  T0     R@ebx[0x0000ffa8][4](R)    T1 {3 (1111, 3) (1111, 3) ()()}
```

So the destination buffer is computed as 0x072e0020 + 8*0xffa8 = 0x072e0020 + 0x7fd40. But we already saw that the heap buffer 0x072e0020 is only 0x7fb60 bytes long. So the beginning of the copy is already beyond the allocated buffer. This means either the offset into the buffer used (0xffa8) or the allocation size (0x7fb60) is bad, or possibly both.

Let us look at these two values and see where they came from to clear up the problem. First, we'll consider the allocation. alloc_reader tells us the buffer is returned from the call to malloc at instruction counter 209160625. To see the actual call to malloc occurring, we can look in the trace for the last instructions that are not in Windows DLL's before this instruction counter.

```
$ trace_reader -intel -trace z.trace -first 209160025 -last 209160625 -count | grep -v
")7c" | grep -v ")78" | tail -6
(209160387)57307f43:       movzx  edi,ax R@ax[0x0000ff6c][2](R)     T1 {3 (1111, 3)
(1111, 3) ()()}     R@edi[0x0000ffa8][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
(209160388)57307f46:       mov    ebx,edi      R@edi[0x0000ff6c][4](R)    T1 {3 (1111,
3) (1111, 3) ()()}  R@ebx[0x0000ffa8][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
(209160389)57307f48:       shl    ebx,0x3      I@0x00000000[0x00000003][1](R)
       T0     R@ebx[0x0000ff6c][4](RW)   T1 {3 (1111, 3) (1111, 3) ()()}
(209160390)57307f4b:       push   ebx    R@ebx[0x0007fb60][4](R)    T1 {15 (1111, 3)
(1111, 3) (1111, 3) (1111, 3) }  M@0x014ad940[0x0000ffd0][4](W)    T0
(209160391)57307f4c:       call   0x0000000057349d2a  J@0x00000000[0x00041dde][4]
(R)    T0     M@0x014ad93c[0x089a6000][4](W)    T0
(209160392)57349d2a:       jmp    DWORD PTR ds:0x573511d8    M@0x573511d8[0x78583bb3]
[4](R) T0
```

The allocation size 0x7fb60 came from ax (0xff6c) after multiplication by 8. Let us slice this register ax to see where *it* came from.

```
$ x86_slicer -in-trace z.trace -ctr 209160387 -regloc ax > /dev/null
$ trace_reader -intel -trace EAX_0.trace  | tail -6
5730803a:     mov    ecx,DWORD PTR [ebp+0xc]    M@0x014ad988[0x0000ffc4][4](R)    T1 {3
(1111, 3) (1111, 3) ()()}  R@ecx[0x08a34718][4](W)    T0
57308043:     movzx  eax,WORD PTR [esi+0x8]     M@0x08a34720[0x0000ffa8][2](R)    T1 {3
(1111, 3) (1111, 3) ()()}  R@eax[0x0000ffc4][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
57308047:     lea    edx,[eax+ecx*1]     A@0x0001ff6c[0x00000000][4](R)    T0     R@edx
[0x00000000][4](W)  T1 {3 (1111, 3) (1111, 3) ()()}
5730804e:     movzx  edx,dx R@dx[0x0000ff6c][2](R)     T1 {3 (1111, 3) (1111, 3) ()
()}    R@edx[0x0001ff6c][4](W)    T1 {15 (1111, 3) (1111, 3) (1111, 3) (1111, 3) }
5730805b:     push   edx    R@edx[0x0000ff6c][4](R)    T1 {3 (1111, 3) (1111, 3) ()
()}    M@0x014ad95c[0x014ad754][4](W)    T0
57307f2b:     mov    ax,WORD PTR [ebp+0x8]      M@0x014ad95c[0x0000ff6c][2](R)    T1 {3
(1111, 3) (1111, 3) ()()}  R@ax[0x0000ffa8][2](W)     T1 {3 (1111, 3) (1111, 3) ()()}
```

So the register ax, which gets multiplied by 8 to compute the allocation size, comes from the addition of two words (0xffc4 and 0xffa8) which sum up to 0x1ff6c. Then this value is copied as a word into the register ax to become 0xff6c. Here the short that will control the allocation size has overflown. This is problematic.

Even though we've basically found the vulnerability, let us slice to see where that offset that is used for determining the write originates.

```
$ x86_slicer -in-trace z.trace -ctr 209429982 -regloc ebx > /dev/null
$ trace_reader -intel -trace EBX_0.trace | tail -5
57308043:     movzx  eax,WORD PTR [esi+0x8]     M@0x08a34720[0x0000ffa8][2](R)    T1 {3
(1111, 3) (1111, 3) ()()}  R@eax[0x0000ffc4][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
5730805e:     movzx  ebx,ax R@ax[0x0000ffa8][2](R)     T1 {3 (1111, 3) (1111, 3) ()
()}    R@ebx[0x0000ffa8][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
57307f3c:     push   ebx    R@ebx[0x0000ffa8][4](R)    T1 {3 (1111, 3) (1111, 3) ()
()}    M@0x014ad948[0x08a34718][4](W)    T0
5730802a:     pop    ebx    M@0x014ad948[0x0000ffa8][4](R)    T1 {3 (1111, 3) (1111, 3)
()()}  R@ebx[0x089b5fd8][4](W)    T0
5730807c:     movzx  ebx,bx R@bx[0x0000ffa8][2](R)     T1 {3 (1111, 3) (1111, 3) ()
()}    R@ebx[0x0000ffa8][4](W)    T1 {3 (1111, 3) (1111, 3) ()()}
```

Oh, the offset is one of the values (0xffa8) that was added to get the value stored in the register ax.

So this vulnerability arises due to a short integer overflow in an allocation which is followed by a memcpy into this buffer at an offset that was used as one of the summands for the short overflow. In other words, this is a heap overflow.

Conclusions

BitBlaze is a binary analysis toolset which allows for data and execution tracing, amongst other things. It can be helpful for a variety of crash analysis tasks. It can be used to determine which registers and memory addresses come from the attacker controlled input file at the time of the crash. It can be used to quickly rule out exploitability in some cases by slicing on Null pointers to see where they originated. Most importantly, the toolset can be used to speed up analysis of crashes to determine the root cause of the vulnerability and whether it can be exploited. These tools include taint tracing of the single byte difference between a good and bad file, as well as slicing data to see where it comes from. Most of these tasks can be done manually or with a debugger, but using BitBlaze can simplify and speed up the process as well as providing repeatability since the program is not needed to be run multiple times. Being able to speed up the time required to perform crash analysis increases the number of crashes that can be examined...and exploited!

References

[BitBlaze] http://BitBlaze.cs.berkeley.edu/
[monkeys] http://securityevaluators.com/files/slides/cmiller_CSW_2010.ppt
[SMS] http://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-SLIDES.pdf
[heap] http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf
[filefuzz] http://labs.idefense.com/software/fuzzing.php
[!exploitable] http://msecdbg.codeplex.com/
[crashwrangler] https://connect.apple.com/cgi-bin/WebObjects/MemberSite.woa/wa/getSoftware?bundleID=20390
[TTF] http://developer.apple.com/fonts/TTRefMan/RM06/Chap6.html#Dictionary
[Nagy] http://www.coseinc.com/en/index.php?rt=download&act=publication&file=A%20New%20Fuzzing%20Framework.pptx

[ARM05] ARM. ARM Architecture Reference Manual, 2005. Doc No. DDI-0100I.
[Bal07] Gogul Balakrishnan. WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison, August 2007.
[BCL+07] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In Proceedings of the USENIX Security Symposium, Boston, MA, August 2007.
[BCS09] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers or how to stop papers from reviewing themselves. In Proceedings of the 30th IEEE Symposium on Security & Privacy, Oakland, CA, May 2009.
[BHK+07] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Dawn Song. Bitscope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, March 2007.
[BHL+07] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University School of Computer Science, January 2007.
[BHL+08] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In Wenkee Lee, Cliff Wang, and David Dagon, editors, Botnet Detection, volume 36 of Countering the Largest Security Threat Series: Advances in Information Security. Springer-Verlag, 2008.
[BNS+06] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In Proceedings of the 2006 IEEE Symposium on Security and Privacy, pages 2–16, 2006.
[BPSZ08] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy, 2008.

[BWJS07] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest pre-conditions. In Proceedings of Computer Security Foundations Symposium, July 2007.
[CC04] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04), December 2004.
[CCC+05] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In In Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP'05), 2005.
[CJMS10] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In Proceedings of the 17th Annual Network and Distributed System Security Symposium, San Diego, CA, February 2010.
[CMBS09] Juan Caballero, Stephen McCamant, Adam Barth, and Dawn Song. Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley, March 2009.
[CPG+04] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In Proceedings of the 13th USENIX Security Symposium (Security'04), August 2004.
[CPKS09] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In Proceedings of the 16th ACM Conference on Computer and Communication Security, Chicago, IL, November 2009.
[CPM+10] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babić, and Dawn Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In Proceedings of the 17th ACM Conference on Computer and Communication Security, Chicago, IL, October 2010.
[cvc] CVC Lite documentation. http://www.cs.nyu.edu/acsys/cvcl/doc/. Page checked 7/26/2008.
[CYLS07a] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07), October 2007.
[CYLS07b] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In Proceedings of the ACM Conference on Computer and Communications Security, October 2007.
[Dat] DataRescue. IDA Pro. http://www.datarescue.com. Page checked 7/31/2008.
[Dij76] E.W. Dijkstra. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.
[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, Proceedings on the Conference on Computer Aided Verification, volume 4590 of Lecture Notes in Computer Science, pages 524–536, Berlin, Germany, July 2007. Springer-Verlag.

[gra] The DOT language. http://www.graphviz.org/doc/info/lang.html. Page checked 7/26/2008.
[Int08] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 1-5, April 2008.
[JR94] Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical Report CS-94-169, Carnegie Mellon University School of Computer Science, 1994.
[KPY07] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'07), October 2007.
[KYH+09] Min Gyung Kang, Heng Yin, Steve Hanna, Steve McCamant, and Dawn Song. Emulating emulation-resistant malware. In Proceedings of the 2nd Workshop on Virtual Machine Security, Chicago, IL, November 2009.
[Muc97] Steven S. Muchnick. Advanced Compiler Design and Implementation. Academic Press, 1997.
[NBFS06] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In Rebecca Write, Sabrina De Capitani di Vimercati, and Vitaly Shmatikov, editors, Proceedings of the ACM Conference on Computer and Communications Security, pages 311–321, 2006.
[NBS06a] James Newsome, David Brumley, and Dawn Song. Sting: An end-to-end self-healing system for defending against zero-day worm attacks. Technical Report CMU-CS-05-191, Carnegie Mellon University School of Computer Science, 2006.
[NBS06b] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS), 2006.
[Net04] Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. PhD thesis, Trinity College, University of Cambridge, 2004.
[NMS09] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In Proceedings of the Fourth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS), Dublin, Ireland, June 2009. http://BitBlaze.cs.berkeley.edu/papers/influence_plas09.pdf.
[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05), February 2005.
[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In Proceedings of the ACM Conference on Programming Language Design and Implementation, pages 89–101, 2007. http://www.valgrind.org/docs/valgrind2007.pdf.
[QEM] QEMU. http://wiki.qemu.org/Main_Page.
[SBY+08] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper., Hyderabad, India, December 2008. http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf.

[Sim96] Loren Taylor Simpson. Value-Driven Redundancy Elimination. PhD thesis, Rice University Department of Computer Science, 1996.
[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04), October 2004.
[SPMS09] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Chicago, IL, July 2009.
[TNL+07] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In Proceedings of the EuroSys Conference, 2007.
[Tri03] Andrew Tridgell. How samba was written. http://www.samba.org/ftp/tridge/misc/french_cafe.txt, August 2003. URL Checked on 8/21/2008.
[XSZ08] Bin Xin, William N. Summer, and Xiangyu Zhang. Efficent program execution indexing. In Proceedings of the ACM Conference on Programming Language Design and Implementation, pages 238–248, 2008. http://www.cs.purdue.edu/homes/wsumner/research/papers/pldi08.pdf.
[YLS08] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), February 2008.
[YPHS10] Heng Yin, Pongsin Poosankam, Steve Hanna, and Dawn Song. Hookscout: Proactive binary-centric hook detection. In Proceedings of the 7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Bonn, Germany, July 2010.
[YSM+07] Heng Yin, Dawn Song, Egele Manuel, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07), October 2007.