Sienna Locomotive, Phase 1

Final Report, Trail of Bits

Introduction

It is possible to formally prove software to be free from flaws, crashes and undefined behavior. However, this would be unduly burdensome and expensive, so most software is not formally verified. This leads program execution paths to incorrect logic, undefined behavior, and crashes.

Not all crashes are created equal. Some halt the execution of the program, others cause undefined behavior. Crashes resulting in undefined behavior such as memory corruption, Use-After-Free vulnerabilities or buffer overflows can result in a vulnerability and potentially be used to take control of the stream of execution via carefully created malicious input.

Traditionally, separating benign from malicious inputs has been a largely manual process, performed by skilled analysts. Some tools have attempted to automate this process, such as Microsoft's "!exploitable" WinDBG extension. !exploitable relies on heuristic-based rules to determine the exploitability of a crashing program state. Although this provides an estimate, it is prone to false positives and false negatives.

In the optimal scenario, an automated tool will not only determine if a crash is exploitable with certainty, but also produce an optimally-minimized exemplar of such an input. This report surveys the current state of the art in crash analysis and current program analysis tools that can further it. We then present the inception of a prototype of such a tool.

Crash Triage

Crashing conditions are the intermediate results of fuzzing software. Execution paths are explored via methods such as fuzzing or concolic execution until a crash is discovered. For sufficiently-complex software, analysis can (and does) result in an glut of crashing conditions. Not all of these crashes are exploitable however, and some of the crashing conditions will represent the same underlying defect or root cause. Collecting individual crashing conditions into groups and determining the "exploitability" of each group becomes a primary goal of a bug discovery campaign.

Crashes are abundant

When targeting significant software projects such as web browsers and network clients, it is not unheard of to generate hundreds of thousands of crashes during a single fuzzing campaign. Most of these crashes will likely not be unique flaws.

For example, when targeting an image parser with a grammar-based fuzzer, a crash resulting from the incorrect handling of image dimensions will always crash on malformed dimensions, even if the color palette implementation is currently being fuzzed.

Crashes are reproducible

For this project, we are focusing on deterministic programs. That is, for a given set of inputs and environment, the program will exhibit identical behavior across a number of unique executions. This provides us with the ability to decouple the program analysis necessary to determine root cause, with the main source of malformed input, such as a fuzzer instance.

The idealized flow of a crash triage system can be separated into three distinct phases: Discovery of crashing inputs, grouping of related crashes, and determination of crash severity. The first phase requires minimal instrumentation to the target application, and can be performed without extra overhead. The last phase can be as computationally expensive as required for an accurate analysis. The three phases should serve as a “triage funnel”, culling irrelevant inputs.

Security relevant crashes

Once groups of inputs that cause an application to crash are identified, we must decide if crashing exemplars lead to vulnerabilities. In other words, we need to quantify the amount of influence an input has on a program’s behavior. This can range from a vague qualification of “it might be possible for software input to influence the program state at crash time” to a precise formula describing the relation of input bits to the control flow and resultant state of the program under analysis.

Existing Tools and Approaches

The remainder of this document discusses the current state of the art in crash analysis and triage, and describes the prototype of a crash severity rating system Trail of Bits began designing.

Stack Hashing

To help deal with the large collection of crash cases, individual cases must be automatically grouped. Stack hashing is a common technique for approximating whether a crashing condition is unique.

Stack hashing seeks to identify and distinguish points in the program where faults occur. It does this by combining the location that a fault occurs with a summarization of the calling context that reaches this location. This calling context is frequently described as a call stack due to the use

of a stack in many programs to store the history of called procedures, their parameters, and local variables. For example, if a procedure A calls a procedure B which calls a procedure C, the call stack would be represented as a sequence of A, B, C.

This mechanism is used when the system detects a crash. To summarize the crash, the system retrieves the call stack from the crashing location in the program. The system then “hashes” this call stack to produce a unique number. Then, a unique set of integers can be used to represent all of the distinct crashes in an application. The goal behind stack hashing is that if there are multiple inputs, each of which will cause a crash, but each crash is attributed to the same bug, that bug is identified by the calling context and the location in the program with the crash.

This heuristic works sometimes, but not others. For example, consider a set of crashes where the program has crashed within the “memcpy” function. This function is very general and used in many locations throughout programs. Calling context is needed to differentiate the conditions, and when it’s provided, a call from A to memcpy that crashes is distinct from a call from B to memcpy that crashes. However, in the case of use-after-free vulnerabilities, crashes that can be morally attributed to the same bug in the program, i.e. some failure to manage the lifecycle of a buffer correctly, can produce multiple crashes that are distinct even when considering calling context.

And finally, and perhaps most pertinently, this technique assumes valid stack frames. In the case of a classic buffer overflow, the stack itself will not represent the functions called to get to a program point.

!exploitable

Microsoft’s Security Engineering Center (MSEC) released !exploitable (pronounced “bang-exploitable”) in 2009 as a free extension to Microsoft’s WinDbg software debugger. It was the first publicly-released project to help developers and analysts determine the security severity of software crashes. !exploitable only supports binaries on the Microsoft Windows operating system, although some of its rules have been ported to the GNU debugger, in various states of completion.

The project implements a variant of stack hashing (including hashing APIs, export tables and other data) to uniquely identify crashes. It then queries and aggregates a list of heuristic-based rules to determine if the crash is exploitable, probably exploitable, probably not exploitable, or produces an inconclusive result.

Although the design of !exploitable can lend itself to automation and scripting, its primary use case is manual invocation from WinDbg, as the tool’s primary users are software developers and not security analysts.

Use

Using exploitable for an analyst is straight-forward: it is implemented as a single command to be executed in a WinDbg debugging session. The output produces a summary of the result data and an English description of the results.

The following present sample output from the extension, analyzing a sample binary from the Trail of Bits' corpus:

```
0:000:x86> !exploitable

!exploitable 1.6.0.0
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - Exception Handler Chain Corrupted
starting at crash01!set_element+0x0000000000000054
(Hash=0xe51c247d.0x5f6b30f4)

Corruption of the exception handler chain is considered exploitable
```

Internals

!exploitable is implemented as a list of rules, which is consulted in a top-down manner. The first rule to match the faulting condition is the conclusion that is provided to the WinDbg user.

The below table summarizes all rules that are implemented by !exploitable. Of note, rule 12 below was used to determine that the above example is exploitable.

	Rule	**Result**	**Source Line(s)**
1	If executing from the stack	**Exploitable**	188
2	If executing from user space, but in kernel mode	**Exploitable**	204
3	If executed an illegal instruction	**Exploitable**	220
4	If a privileged instruction exception is hit	**Exploitable**	236
5	If a guard page violation is hit	**Exploitable**	252
6	If the stack is overrun	**Exploitable**	268
7	If heap corruption is detected	**Exploitable**	284
8	If a DEP violation is hit in kernel mode	**Exploitable**	300
9	If a DEP violation is hit in user mode (if not near NULL)	**Exploitable**	316
10	If an access violation is hit at the in kernel mode	**Exploitable**	331

11	If an access violation is hit in user mode (if not near NULL)	**Exploitable**	348
12	If the exception chain is corrupted	**Exploitable**	369
13	If write violation in user space memory and not near null	**Exploitable**	457
14	If write violation in kernel memory from kernel code	**Exploitable**	475
15	If second chance write violation in kernel code	**Exploitable**	490
16	If access violation on a control flow instruction	**Exploitable**	506
17	If access violation on a control flow instruction in user mode (if not near NULL)	**Exploitable**	522
18	DEP violation in user mode (if not near NULL)	**Likely Exploitable**	384
19	Try disassembling the instruction, if can't	**Likely Exploitable**	425
20	If access violation on a control flow instruction in user mode	**Likely Exploitable**	538
21	If read access violation on a block data move	**Likely Exploitable**	555,570,586
22	If locally "tainted" data is used to control branch target	**Likely Exploitable**	616
23	If locally "tainted" data is used to control write	**Likely Exploitable**	630
24	If read access violation (if not near NULL)	**Likely Unexploitable**	678
25	If first chance exception for access violation, of kernel code to user memory	**Likely Unexploitable**	704,720
26	Divide by zero	**Likely Unexploitable**	736, 752
27	Stack exhaustion	**Likely Unexploitable**	768
28	If application verifier stops	**Unknown**	654
29	All user mode write access violations (if near NULL)	**Unknown**	788
30	Breakpoints are probably not exploitable	**Unknown**	804
31	Bug checks (BSOD)	**Unknown**	820
32	If the stack contains unknown functions	**Unknown**	853
33	If access violation (read) in kernel mode	**Unknown**	868
34	Other influence from locally-tainted data	**Unknown**	884,900,916 ,932,949

Source lines refer to the location of the rule in exploitable_rules.h source code file

Although some rules refer to data tainting, !exploitable only propagates data taints within a single basic block.

SAGE

The SAGE project was a Microsoft effort to combine fuzzing and symbolic execution to identify errors in Microsoft products such as Windows and Office. SAGE combines static analysis and white box testing with fuzzing and black box testing techniques. The system uses fuzzing with trace recording to record what branches have been taken, then synthesizes constraints on the

input required to take the other direction of the branch and feeds that system of constraints to an SMT solver. If a satisfiable model is produced from the SMT solver, it is projected back onto the input to the application and a trace is collected of the application taking the other branch.

The goal of the system is to increase code coverage reached via fuzzing, and to reach new states in the application. As these two metrics are increased, the system also discovers inputs that cause the application to crash. These inputs are saved off to the side and investigated later by the application developers.

A benefit of SAGE as compared to prior software assurance tools is independence from input format specification. SAGE has only one primary goal - find crashes, and a secondary goal that supports the primary - increase code coverage. As SAGE executes, it gathers constraints that keep concrete execution from reaching new code, and it uses taint information to infer the relationship between the input and these unreached locations. It takes the constraints gathered and solves to discover new inputs that would reach previously unreached locations in the program. This allows SAGE to produce new inputs, without any explicit (i.e. grammar based) knowledge of the input format. The ideas behind SAGE have been the basis for successive concolic bug finding tools like FuzzGrind, Triton, Fuzzball / BitBlaze, and Mayhem.

SAGE is a Microsoft-internal tool and not generally available. however it has been published about extensively in academic literature. An implementation of the core SAGE algorithm is available in Triton.

Angr

Angr is a binary analysis framework implemented mostly in the Python programming language. Angr allows an analyst write scripts to automatically explore possible code paths and brings together a number of static analysis tools such as a machine code lifter, a simulation engine, and a collection of loaders. Angr was born from the Shellphish CTF team from UCSB to assist in program analysis for the DARPA Cyber Grand Challenge program and has been used to solve a number of CTF challenges.

Angr works by first statically loading a target binary (and its dependencies) translating instructions to Valgrind’s intermediate format, VEX. It is then possible to use Angr’s static analysis tools to perform tasks such as program path exploration, VEX emulation and solve for path constraints by modeling VEX semantics as SMT statements. The entire process is implemented statically -- that is, no native instruction is executed throughout analysis.

Use

The standard use case of Angr involves defining the symbolic inputs, adding constraints to the evaluation of the program, and then letting Angr discover execution paths that satisfy those constraints.

As an example, we take a basic challenge binary that expects to get a password from the command line[1]. The password is verified by an unknown, but of low complexity, decryption algorithm.

```
user@host:~$ ./r100
Enter the password: password
Incorrect password!
```

To solve the challenge, we simply write a script to tell Angr where in the execution we want to arrive at (after the password is correctly verified) and have it solve the path the program can take to arrive there.

```
import angr

def main():
    p = angr.Project("r100", load_options={'auto_load_libs': False})
    ex = p.surveyors.Explorer(find=(0x400844, ), avoid=(0x400855,))
    ex.run()
    return ex.found[0].state.posix.dumps(0)

if __name__ == '__main__':
    print main()
```

Here, `0x400844` marks the basic block that is executed when the password is correct (this was verified independently) and `0x400855` marks the basic block that produces a “password incorrect” error message. With the above basic setup, we let Angr solve the the input that is required for the execution to arrive at the resultant basic block:

```
user@host:~$ python solve.py
Code_Talkers

user@host:~$ ./r100
Enter the password: Code_Talkers
```

[1] https://github.com/angr/angr-doc/tree/master/examples

```
Nice!
```

Angr works exceptionally well for solving for simple execution paths, those commonly found in CTF challenges. Since Angr evaluates all paths symbolically, it struggles to scale to even medium-scale real world problems.

While Angr is poor at determining exploitability of real world crashes, it can serve as a valuable tool to add verify exploitability and the vulnerability call site.

BitBlaze

BitBlaze[2] is a binary analysis platform developed at UC Berkeley. It is composed of three primary tools: Vine, TEMU and Rudder. Vine is the static analysis component. It defines an intermediate representation for executables, contains implementations of some static analysis algorithms and connects the intermediate representation with theorem provers. Rudder is a dynamic analysis platform that can explore multiple execution paths at run time and build equivalency formulas for dynamic program state. TEMU is a modified QEMU full-system emulator that is used for hosting the dynamic execution of target binaries.

Vine

Prior to performing static analysis, Vine needs to convert the native representation of a program's code to an intermediate representation that encompasses the semantics of target hardware. Vine defines its own intermediate language (VineIL) that encodes the semantics of instructions and their side-effects.

Vine uses Valgrind's VEX format as an intermediate step, receiving assistance from external disassemblers and Valgrind tools for the lifting process. VineIL further ensures that platform-dependent features such as endianness are handled consistently. Individual VineIL instructions are side-effect free and all data casting is explicit. Furthermore, VineIL can also be expressed in SSA form to simplify further analysis.

Once a program is lifted, it can further be transformed and annotated by Vine's passes, to build Control Flow Graphs, to perform program slicing, or optimizations. Vine's backend is then used for analysis passes such as value analysis and interfacing with a solver.

Since static analysis is inherently computationally expensive for non-trivial programs, Vine can be paired with TEMU for concolic analysis.

---

[2] http://bitblaze.cs.berkeley.edu/

TEMU

TEMU is a dynamic binary analysis tool derived from the QEMU[3] project. QEMU is an open source machine emulator; a virtual machine emulator that can both act as a hypervisor and a complete system emulator. TEMU, just as QEMU, emulates the entire operating system, all user processes and has complete insight into data flowing in and out of the system.

TEMU includes a model of the host operating system, including processes and threads. TEMU also adds a process monitoring extension to QEMU that detects activity such as process creation and destruction, and memory map updates.

All taint introduction, propagation and monitoring is also implemented in TEMU via shadow memory. Shadow memory is a separate region of memory that contains the taint information of every physical byte in the target system. Any memory location can then be queried for taint status. TEMU also provides support for different plugins to respond differently to taint propagation and status.

TEMU also includes a plugin, Tracecap, which records execution traces, bundling with it taint transfer data. Recorded traces can then be used to analyze taint propagation and influence post hoc.

Rudder Et Al

While Rudder is documented as a dynamic analysis toolkit that is capable of path exploration, it is not released and thus, was not explored or considered for this report.

BitBlaze has been used for crash analysis in the past, however it is difficult to scale BitBlaze to work on individual applications. The most immediate way to use BitBlaze is to collect a trace of the applications behavior within a TEMU virtual machine. This comes at a cost of tremendous slowdown as TEMU is a whole system virtual machine. The BitBlaze system is also not well maintained, and is implemented in a programming language with limited support for Windows (OCaml). There has not been an update to TEMU or Vine since their initial releases in 2008.This also has poor implications for using BitBlaze as a platform for prototyping and rapid research. For this reason, we didn't explore using BitBlaze.

Pin

Intel's Pin is a dynamic binary instrumentation toolkit for Intel x86 and x64 architectures. Pin permits the creation of programs that introspect or analyze code via a number of callback

[3] http://www.qemu.org

functions. Pin is traditionally used for performance tuning, generic program analysis and instrumentation of parallel code.

To improve performance, code that is instrumented (“client code”) is Just-in-Time translated before when inserting instrumentation. Although this preserves the semantics of executed client code, the memory and instruction addresses are different at runtime.

Client code, or “pintools”, are implemented in the C or C++ programming languages and can be linked at runtime. Pintools provide callbacks for events in the system, such as a function being called, an instruction being executed, or memory being read. Pin also defines a number of standard types representing program atoms (such as instructions, memory cells, etc) for portability.

Use

To provide a basic example, we are going to instrument the Unix “`ls`” utility to count how many instructions were executed in its listing.

```
#include "pin.H"
#include <iostream>
#include <fstream>

static uint64_t kInstructions;
std::ofstream kOut;

void on_instruction(void) {
  kInstructions++;
}

void instrument_ins(INS i, void *arg) {
  INS_InsertCall(i, IPOINT_BEFORE, (AFUNPTR)on_instruction, IARG_END);
}
void on_fini(int32_t code, void *arg) {
  kOut << "Ran " << kInstructions << " instructions.\n";
  kOut.close();
}
int main(int argc, char *argv[])
{
  if( PIN_Init(argc,argv) )
      return 1;

  kOut.open("pin_output");
```

```
  INS_AddInstrumentFunction(instrument_ins, 0);
  PIN_AddFiniFunction(on_fini, 0);

  PIN_StartProgram();

  return 0;
}
```

MyTool.cpp

```
user@ubuntu:~/pin/source/tools/MyPinTool$ pin -t obj-intel64/MyPinTool.so
\
 -- /bin/ls
bak.MyPinTool.cpp  makefile  makefile.rules  MyPinTool.cpp  obj-intel64
pin_output
user@ubuntu:~/pin/source/tools/MyPinTool$ cat pin_output
Ran 453985 instructions.
```

Since Pin is frequently used for parallel program profiling, its instrumentation is very light weight.

Triton

Triton is a dynamic binary analysis toolkit built on top of Pin explicitly developed for software security research and supports the x64 architecture. Triton permits the creation of scripts that guide program analysis of client code as it is executed. It is also possible to use Triton to reason about architecture semantics. Triton is development is sponsored by Quarkslab and is used internally for software security research.

Components

Triton is composed of a number of high-level components.

Intermediate Representation

All components are unified via an intermediate representation of x64 semantics. Every instruction's behavior and side effects is modeled as one or more logical formulas, which are sometimes generically referred to as ASTs. ASTs serve two purposes: they permit analysis on

native machine instructions, and they get exposed via a Python API. The ASTs describe the precise semantics of each individual x64 instruction, including implicit flag manipulations and side effects.

For example, the x86 instruction "`add rax, rdx`" is represented by the following formulas:

```
ref!1 = (bvadd ((_ extract 63 0) ref!0) ((_ extract 63 0) ref!39))
ref!2 = (ite (= (_ bv16 64) (bvand (_ bv16 64) (bvxor ref!1 (bvxor
((_ extract 63 0) ref!0) ((_ extract 63 0) ref!39))))) (_ bv1 1) (_
bv0 1))
ref!3 = (ite (bvult ref!1 ((_ extract 63 0) ref!0)) (_ bv1 1) (_ bv0
1))
ref!4 = (ite (= ((_ extract 63 63) (bvand (bvxor ((_ extract 63 0)
ref!0) (bvnot ((_ extract 63 0) ref!39))) (bvxor ((_ extract 63 0)
ref!0) ref!1))) (_ bv1 1)) (_ bv1 1) (_ bv0 1))
ref!5 = (ite (= (parity_flag ((_ extract 7 0) ref!1)) (_ bv0 1)) (_
bv1 1) (_ bv0 1))
ref!6 = (ite (= ((_ extract 63 63) ref!1) (_ bv1 1)) (_ bv1 1) (_ bv0
1))
ref!7 = (ite (= ref!1 (_ bv0 64)) (_ bv1 1) (_ bv0 1))
```

The first formula (ref!1) represents the instruction as an addition of two 64-bit values. The remaining formulas model the instruction's side-effects on system flags.

One thing to note from the above intermediate representation, is the creation of new intermediate values after each instruction. This is a due to the representation being in Static-Single Assignment (SSA) form, simplifying further analysis by avoiding aliasing issues of real architectures. One other benefit of representing behavior via an SSA form is the simplification of propagating data taints across memory.

Intel Pin

The project is entirely based on Intel's Pin instrumentation framework. The majority of the project is implemented as a shared library Pintool, implemented in the C++ programming language. As client code is executed, each instruction is modeled as one or more intermediate formulas. User scripts can then register to do further analysis on each instruction, or guide all future execution of the client code.

Python

Triton's user scripts are authored in the Python programming language. To facilitate this, Triton exposes a number of standard callbacks from Pin to Python functions. This enables the analyst to prototype and implement analyses in a scripting language.

Bindings are provided for the intermediate representation, as well as to a number of solvers via the SMT2-LIB interface. It is also possible to query the entire current state of the program, which includes memory, instructions, processor state.

User Scripts

All of Triton's components are brought together via the development of user scripts. The taint engine, the solver interface, the SMT2 interface, and the data taint engine are all exposed to Python. A script developer can also introspect into the instructions, memory and intermediate representations via callback functions.

For example, the following user script will display all calls to the open system call, and print its first argument and what the call returned:

```
from triton  import *
from pintool import *

def openEntry(threadId):
    path_addr, path, val = getRegisterValue(REG.RDI), '', 1
    while val != 0:
        val = getCurrentMemoryValue(path_addr)
        path, path_addr = path + chr(val), path_addr + 1
    print 'open(%s)'%(path),


def openExit(threadId):
    ptrAllocated = getRegisterValue(REG.RAX)
    print '-> %#x' %(ptrAllocated)

if __name__ == '__main__':
    setArchitecture(ARCH.X86_64)
    startAnalysisFromEntry()

    addCallback(openEntry, CALLBACK.ROUTINE_ENTRY, "open")
    addCallback(openExit, CALLBACK.ROUTINE_EXIT, "open")
```

```
    runProgram()
```

Our Approach

Trail of Bits is working to combine the most promising features of researched tools and techniques into an early prototype.

Taint Tracking

Our hypothesis is that the key differentiator between unexploitable and exploitable crashes is attacker influence. If a crash can only dereference a constant invalid pointer (i.e. the value 0) then this crash is both uninteresting and has a low amount of attacker influence (a single bit). However, when an attacker has a large degree of influence, they have more control over the failure state of the application.

Taint provides a missing component of crash evaluation. Before, we would try and write a rule to capture whether or not a particular crash was exploitable, and we would argue “a pointer is dereferenced and the pointer is invalid, therefore, it is probably exploitable.” However this is not a good metric, what if the pointer is minimally influenced by the input? Now we can state a two part rule, “if a pointer is dereferenced, that pointer is invalid, and the pointer value is tainted, then the crash is probably exploitable.”

To use these new rules, we need to augment the execution of the program with the notion of taint and the propagation of taint. Taint begins with some input provided to the program, for example via the `argv[]` array in main or calls to functions like `getenv` or `read`. As our application executes, tainted data will be operated on and copied into other variables and locations in the program.

For example, consider the following program:

```
typedef int (*process)(int, int*);
process process_arr[] = { p1f, p2f, p3f };

int main(int argc, char *argv[]) {
  if(argc < 3) {
    return 1;
```

```
  }

  int pf = atoi(argv[1]);                  // P1
  int argcount = argc - 3;                 // P2
  int *arr = malloc(argcount*sizeof(int));
  for(int i = 0; i < argcount; i++) {      // P3
    arr[i] = atoi(argv[i+2]);              // P4
  }

  return process_arr[pf](argcount, arr);   // P5
}
```

Our goal is to track the taint in this program. We start with how tainted data is introduced to the program: uses of `argv` and `argc`. At point P1 we see tainted data read from `argv` and used as an argument to `atoi`. Here, our hypothetical taint tracking system needs to know what `atoi` does to a taint. Intuitively we understand that if the attacker provides any string to atoi, atoi will produce an integer that maps to that string. Therefore, our model can state that `atoi` preserves taint and if any parameter to `atoi` is tainted, then so is the return value from `atoi`.

The rule for propagating taint is even simpler at point P2. `argc` is a value that the attacker can directly control (by adding more parameters on the command line) so we just need a rule for subtraction, or really, any binary operation. Here we can say that for a binary operation a O b, if either a or b is tainted, then the result of a O b is also tainted. At this point, both `pf` and `argcount` are tainted.

At P3, the situation becomes more interesting. The integer value i is initialized to 0, however the execution of the loop and the subsequent increment of i is determined by a value that is tainted. This represents an implicit flow in the application, because while i is never tainted according to either of the rules we have established thus far, at the end of the loop, the value of i will depend on the value of argcount. At this point, we have to introduce the notion of a program counter, a value that determines which line of the program we are executing, and allow this program counter to also be tainted. Then, we say that any value written to while the program counter is tainted, is also tainted. This lets us capture that i is tainted, however, it would also result in tainting the rest of the program after the execution of the loop. We add an additional rule that states that the program counter loses its taint once we exit the loop.

Taint has to be tracked as the application executes, it can't be applied or computed once the program crashes. This is part of why we expect that applications have some idempotence with respect to inputs - inputs that produce crashes should always produce crashes assuming the application code isn't changed. Tracking taint as the application executes has some overhead, so we only do this once we identify that an input crashes.

Where this information becomes useful is at the site of the crash. With taint tracked through the execution of the program, the site of the crash now contains information about which registers and memory locations are tainted. Our prior rules let us note that at P4, every position of arr is tainted as input. At P5, something interesting happens. Our program indexes into a global array of function pointers and calls a function pointer. The value pf, used for this index, is not constrained relative to the bounds of the array, so any memory in the program might be treated as a function pointer. It's possible that the fuzzer would produce a crash at this location by choosing a very large value as the first argument to the program. Now, our system has to determine the exploitability of that crash. Examining the situation at P5, we see that the operation involved is a call, and the address of the location called is tainted.

Bare Program Tracing

Dynamic binary analysis tools excel at analyzing complex and real-world code that static analysis typically falls short on. Having concretely-executed instructions prevent the state explosion problem when relying solely on static analysis. However, dynamically instrumenting code is intrinsically invasive — either the instrumentation code executes in the same address space as the client code, or externally, but with severely degraded performance and analysis due to the interface for inter-process communication.

Trail of Bits chose to build our analysis prototype with the Triton framework. Since Triton is implemented with Pin, Pin itself, the Python programming language, and the implementation of the intermediate representation all exist in the same address space as the target binary. Not only can this have an impact on the execution environment in terms of performance, but can potentially interfere with the behavior of the client program.

To address these issues, we created a standalone tool that executes a client program to completion, recording the complete trace of execution to disk. The Triton script in turn loads the previously recorded trace and steps through the execution with a known reference, ensuring that: its presence doesn't modify the execution of the client code, and that it is still executing valid instructions.

Use

The tracer is a C++ implementation that attaches to a client program via the `ptrace(3)` system call and single-steps the program until completion. At each step, the register state is recorded as Protobuf[4] structures into a trace file.

Creating a trace file is a single command:

[4] https://developers.google.com/protocol-buffers/

```
user@host:~/tracer/build/ptrace$ ./tracer /bin/echo hello --output \
echo_trace
hello
Finished with status: 0
user@host:~/src/sl-stage/tracer/build/ptrace$ ls -la echo_trace
-rw-rw-r-- 1 user user 15302 Mar 15 15:28 echo_trace
user@host:~/tracer/build/ptrace$
```

The trace file is then loaded by a Triton-based script when the binary is fully instrumented with Triton. The trace itself only stores register values that were modified since the last instruction.

Prototype

Trail of Bits began work on a crash analysis prototype melding the bare program tracer described above, with a script developed on the Triton instrumentation framework. The prototype's current incarnation was used against basic programs, written in C, to test various means of analysis. This prototype is intended to serve as a testbed for the technology required for the second phase of this effort and to determine its feasibility.

Instrumentation and Crashes

While Dynamic Binary Instrumentation works well for analyzing the influence of a specific input on a target binary, it can struggle with handling some crashing conditions. For example, Triton specifically requires all memory accesses to be to resident pages of memory. This is typically not of concern, as well-behaving programs always dereference valid addresses.

Another potential cause of concern is the implementation of Pin (and in turn, Triton) is its co-existence in the same address space as the client code. This can potentially influence the execution of client code. Furthermore, since each instruction is emulated, the semantics can potentially diverge from native code.

To address these current and potential problems, we implemented a stand-alone program tracer that records the trace of a crashing program to completion.

The tracer is implemented in C++ using the `ptrace(3)` system call and is designed to run on the Linux operating system. More information can be found in the sections below.

Taint Tracking with Triton

Triton's intermediate representation described previously is generated by C++ code that implements the semantics of each x64 instruction. For example, when Triton executes an "sub" (subtract) instruction, this eventually calls a C++ implementation of "sub" semantics. This function generates and executes the intermediate representation that implements the actual subtraction and store operation, but also emits instructions that set effected global flags, such as the Carry or Zero flags.

The code that generates and evaluates intermediate representation provides an ideal location to add instrumentation for taint propagation. Since all implicit data flows (such as flag updates) are explicitly defined, taint propagation is straightforward. Triton intersperses taint propagation code in throughout the implementation of x64 semantics, and then exposes a high-level API through its Python bindings.

Propagating taints can be accomplished in one of several ways. Let's assume client code executes the instruction "`mov al, bl`", and the `rbx` register is tainted. A perfectly-accurate model of taint propagation would only mark the lowest 8 bits of rax as tainted, but Triton marks the entirety of rax as tainted. This is technically an over-approximation, but this has the benefit of not incurring a burden to the execution time without losing significant accuracy.

The prototype we have developed interacts with Triton's taint engine via very simple means. We simply mark the pointer to input data as tainted, then let the Triton taint engine propagate it. Once we have detected that we are one instruction away from the crashing instruction, we query the taint engine to see how whether the current state was influenced by the taint introduced at the input. The below snippet installs a callback before an instruction's symbolic evaluation. Here, we check if the rax register contains one of two addresses, which implies one of two buffers are being passed as the first argument to a function. If they are, we taint that register. Later on, when we determined we are executing the last instruction prior to a crashing condition, we simply check if the current state is tainted.

```
def beforeSymProc(inst):
    if inst.getAddress() in [0x400886, 0x4008a4]:
      taintRegister(REG.RAX)
      print "Tainting rax"

# . . .

addCallback(beforeSymProc, CALLBACK.BEFORE_SYMPROC)
```

Triton Script

The Triton portion of the prototype is implemented via a number of utility classes and callback functions in Python. At a high level, we load the trace produced by the "`tracer`" utility, taint the addresses that contain input data, then step through the trace in parallel with the execution of the client binary.

During the execution of the client binary, at each instruction, the prototype checks how many instructions remain until the crashing condition. Once the crashing condition is impending, we query the taint engine for the influence of input to the current state. The relevant parts of the main callback function is reproduced below.

```
class LockStep(object):
  '''
    The main class implementing the crash analysis callback
  '''
  class DivergedError(Exception): pass

  def __init__(self, tracefile):
    self._tracer = trace_reader.TraceReader(tracefile)
    self._maps = proc.Proc()
    self._state = state.State()
    self._rules = exploitable.Rules
    self._last_instruction = None


  ...

  def __call__(self, instruction):
    '''
      The primary analysis callback, called on a per-instruction-executed
      basis.
    '''
    self._last_instruction = instruction

    # ...

    instr_from_trace = self._tracer.next_instruction()

    if instruction.getAddress() != self._tracer.get_register_value('rip'):
      raise DivergedError("Instruction diverged at 0x%x"%(instruction.getAst(),))
```

```
    if not self._tracer.done():
      return

    print "Trace ended; finding taint results"

    self._printTaintResults()
```

Prototype Usage and Results

To use the prototype, we first create a trace.

```
user@host:~/tracer$ ./tracer
Options:
  -t [ --text ]         Store instruction as disassembly, instead of in raw
                        form
  -i [ --input ] arg    Target program and arguments
  -o [ --output ] arg   Trace destination
  -h [ --help ]         Help

user@host:~/tracer$ ./tracer -o trace.dat -i ./crash01 1 5
Returned: 4
Finished with status: 0
user@host:~/tracer$ rm trace.dat
user@host:~/tracer$ ./crash01 1 300
Segmentation fault (core dumped)
user@host:~/tracer$ ./tracer -o trace.dat -i ./crash01 1 300
Finished with status: 2
user@host:~/tracer$ ls -la trace.dat
-rw-rw-r-- 1 user user 81935 Mar 12 16:06 trace.dat
```

We see that the the process crashed (returned 2) with arguments that were shown to crash the binary. We also verify that a trace was created; in this case it is 81kb in size.

We can now load the binary under Triton, and use our generated trace to guide the analysis.

```
user@host:~/trace/triton/scripts$ . update_environment.sh
user@host:~/trace/triton/scripts$ ~/tools/Triton/triton main.py ../../build/crash01 1
300
64
Running binary ...
Tainting rax
Tainting rax
Trace ended; finding taint results
Heuristic results:
       0
```

```
Tainted registers:
       rdi
       rsi
Tainted stack:
    7ffd92bd4020, 16 bytes
Value of rsp:     7ffd92bd4058
Last symbolic expressions:
        ? (concat ((_ extract 7 0) ref!2515) ((_ extract 7 0) ref!2474) ((_ extract 7
0) ref!2433) ((_ extract 7 0) ref!2392) ((_ extract 7 0) ref!2351) ((_ extract 7 0)
ref!2310) ((_ extract 7 0) ref!2269) ((_ extract 7 0) ref!2228))
        ? (bvadd ((_ extract 63 0) ref!12879) (_ bv8 64))
Last executed instruction
       40085e: ret
```

Here, we see that the binary executed without diverging from the initial trace, and that the crashing state had data movement registers (rdi, rsi) as well as data on the stack tainted. The instruction that crashed was a ret instruction, as the program attempted to return to an invalid location. This printed output would be the input to a function that ranks the severity of the crash, which would consider the tainted data and the faulting instruction. In this case, the algorithm would consider the the stack is tainted and the instruction is a ret. However, the data read off the stack for the return is not tainted, and the system would not classify this as exploitable. To understand why, let's look at the source code for the vulnerable application.

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

int set_element(size_t start, size_t destination)
{
  int n = 0;
  char arr[16];
  if (destination < start) {
    return 0;
  }

  n = (int) destination - start;
  while (start < destination) {
    arr[start] = 'X';
    start++;
  }

  return n;
}

int main(int argc, char *argv[])
```

```
{
  int status;
  size_t start, dest;

  start = atoi(argv[1]);
  dest = atoi(argv[2]);

  status = set_element(start, dest);

  printf("Returned: %d\n", status);
  return 0;
}
```

In this example, the tainted values come from argv and are integers, and these control the amount of data written to the stack. However, the data written to the stack is the constant byte ‘X’ (or 0x58), which is not tainted. The 16 bytes stored on the stack that the system marks as tainted are the values start and dest saved in main, and the value that overwrites the saved return address is not influenced by the attacker.

Fine grained exploitability

Hidden in this example is a source of imprecision in the taint based analysis approach: what if it is possible to carry out a return-to-libc style attack by overwriting some amount of of the saved return address with a static value? For example, what if the function “system” was stored at the address 0x0000005858585858, which could be reached by overwriting the return address with 5 ‘X’s (assuming that the first 3 bytes were already zero)? Systems programmers would rightly consider this effectively impossible but there isn’t a specific reason that it couldn’t happen, only a statistical improbability.

Even if the stars aligned to present the attacker with system at this address, exploitability would most likely still be impossible however to arrive at this conclusion we have to use a level of reasoning beyond taint analysis. Existing approaches make use of forward symbolic execution with a specific goal of reaching an “arbitrary code execution” state and relying on the search performed by the symbolic executor to discover exactly how to get to that state. If the search comes up empty, then as far as that system is concerned, the vulnerability is not exploitable. Even then, it is possible that the forward symbolic executor is incorrect because it only attempts the strategies it knows to try. It is hoped that by giving the symbolic executor as general a strategy as possible, it will subsume all reasoning performed by a human expert, however sometimes the expert can be more creative than the computer. Additionally, the forward symbolic executor is very heavy-weight and could run without producing a conclusive answer one way or another.

Either way, the taint based approach still provides a rank ordering of exploitability. Consider an alternative program where the data written onto the stack is wholly user controlled. Then exploitability is trivial, and the taint analysis system would (correctly) say as much. This highlights some of the tension between making a scalable system and a sound system. Systems that scale frequently have some unsoundness, while sound systems are difficult to scale.

Future work

Our proof of concept and research shows that using taint information to understand the impact of crashes shows promise. The prototype can disambiguate crashes that would appear to be exploitable by current tools from crashes that actually might be exploitable. Originally, we thought that a notion of quantified influence would be useful, however after surveying existing tools we think that incorporating any information about influence at all is worthwhile. Following work could examine:

- Scaling our trace-and-analyze technique to larger programs. The trace format produced by the tracer makes attempts to compact the trace, however sometimes this doesn't work. The tracer also doesn't deal with some program behavior observed by larger programs and multi-threading.
- Quantifying attacker influence in traces. This requires re-visiting the taint propagation system in Triton, which is possible as the taint system is open source. The taint system in Triton is not currently bit-level, however it could be extended to be bit-level. Then the system could about individual bits of attacker influence.
- Using automatic exploit generation to quantify difficulty of exploit generation. This is a complicated and theoretical idea and automatic exploit generation is not generally solved.

The existing prototype represents a necessary starting point for all of these efforts. With further work, we would extend the prototype to reason more deeply about crashing conditions and generate longer form output more easily interpretable by software security practitioners.