Symbolic Execution

zv <zv.github.io>

<2018-09-05 Wed>

Outline

What Is This Talk?

This talk is a quick review for novices to symbolic execution on how to get up to speed on the history, existing tooling and how to use it.

Symbolic Execution? Who Needs it?

Symbolic execution is a method of evaluating programs with "symbols" as arguments.

- Provide *symbolic* rather than concrete inputs
- Explore all paths in a program, solve for which inputs cause a particular path to be taken.
- Report crashing input paths or paths which should cause a crash
-

But where does it come from?

The term itself goes way back, to a system called SELECT.

> *The execution proceeds as in a normal execution except that values may he symbolic formulas over the input symbols.*

And then.

The authors of SELECT were interested in doing the uninteresting: Proving a few nontrivial properties of computer programs written in a special-purpose language.

And

. . . and were largely forgotten by history until a paper called EFFIGY is released, which, for reasons that in retrospect seem to be a function of chance alone, takes off.

Now.

There's not enough time to discuss how this influences future research, but the rest is history.

What is it?

If you've decided to come see this talk, you likely already know what symbolic execution is.
There have been dozens of talks on the topic and many of them focus on researcher's individual contributions rather than reviewing the best way to find bugs.

Before we get into that

Before we get into that however, we need to review just why we should be be using something that isn't a fuzzer.

Powers

- Detect subtle flaws in small, focused cases.
- Assist a fuzzer in 'jumping' difficult-to-test predicates
- Find flaws in code with no obvious defects
- Analysis is automated

Weaknesses

- Slow.
- Can never beat fuzzers in general purpose bughunting.

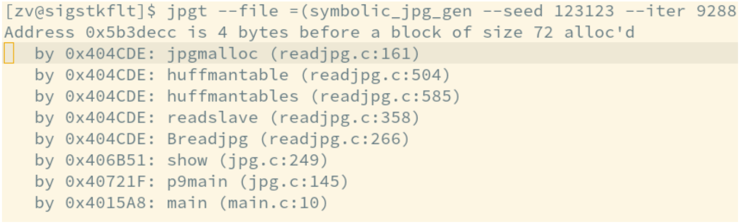
My Story or Lack Thereof

I got interested in this after witnessing a particularly brutal smackdown of ivory-tower symbolic execution (v. smart and practical chad fuzzing) on IRC.

Building a Background

I wrote my own really simple SAT solver, watched a lot of video lectures, reviewed existing methods, tried to construct my own x86 "lifter" to find real vulnerabilities in real, huge code bases.
This all was part of my journey to tell you that there's already very good tools out there.

First Vulnerability



```
[zv@sigstkflt]$ jpgt --file =(symbolic_jpg_gen --seed 123123 --iter 9288
Address 0x5b3decc is 4 bytes before a block of size 72 alloc'd
   by 0x404CDE: jpgmalloc (readjpg.c:161)
   by 0x404CDE: huffmantable (readjpg.c:504)
   by 0x404CDE: huffmantables (readjpg.c:585)
   by 0x404CDE: readslave (readjpg.c:358)
   by 0x404CDE: Breadjpg (readjpg.c:266)
   by 0x406B51: show (jpg.c:249)
   by 0x40721F: p9main (jpg.c:145)
   by 0x4015A8: main (main.c:10)
```

- A bug in the way CMYK colorspaces are handled in the JPEG parser in Plan9!
- Allowed for a 16-byte write to anywhere within 2^{16} of the local stack variable
- Generally high-quality codebase.
- Plan9 has many facilities for viewing images
- finding a vulnerability in an image library would be ideal.

Plan9 From User Space

- Plan9 is a research operating system developed by Bell Labs as a successor OS to Unix.
- CMBC couldn’t work
- KLEE
- Try to do this "manually"!

How?

This means breaking out an SMT solver and hunkering over the intel manual to figure out how to approximate each instruction. This is hard.
There are hundreds of instructions and each instruction can have unusual effects based on it's operands (in fact I decided to just hack around `rep` and instructions like `shl`)

Erlang Runtime Vulnerability (CVE-2016-10253)

A vulnerability in how regular expressions are compiled inside of the Erlang runtime can result in remote memory corruption.

How?

- Extract out the appropriate functions in ERTS with tokenrove's niffy so they can be easily tested
- Try to take out the core logic of compiling a regex/branch
- Just apply KLEE!

PCRE

- Erlang's regular expression implementation draws heavily on the existing PCRE source code.
- Symbolic execution found some bugs that had already been detected in PCRE and fixed upstream.
- But... it also found some new vulnerabilities

State of the tools

There's many existing tools out there but I'd recommend the following if you're trying to get your feet wet.

- CMBC
- KLEE
- Manticore

What's CBMC?

CBMC is a tool produced by the software verification group at Oxford that can be used to check general pointer safety conditions as well as user-defined assertions.
This means you can 'prove' particular statements in code as well as find 'ordinary' memory corruption bugs.

Stack-Based Buffer Overflow with CMBC

CBMC attempts to model some important POSIX functions, but as a general rule assumes that all library functions it does not have access to return arbitrary values.
Here is an example of a trivially exploitable buffer overflow straight from Aleph One's *Smashing the Stack for Fun and Profit*

```
void overflow(char *str)
{
    char buffer[16];
    strcpy(buffer, str);
}
```

CMBC (be sure to set `--trace`, `--unwind 64` and `--pointer-check`) will dutifully report an error in this code, although the examples it gives only give you a hint.

Result

You'll recieve something like the following:

Unbounded finite read

```
State 27 file overflow.c line 13 function main thread 0
----------------------------------------------------
src=array!0@1 + 2251799813685165 (00000011 00000111 1111111
11111111 11111111 11111111 10101101)
```

Dereference of invalid address

```
Violated property:
file overflow.c function <builtin-library-strcpy>
dereference failure: ....
```

Stack Based Buffer Overflow with Klee

Klee is now a part of LLVM and is somewhat different from CMBC. KLEE can automatically check files for bugs, but in general you'll want to indicate a symbolic variable with `klee_make_symbolic`

Running KLEE

```
clang -emit-llvm -g -c overflow.c -I klee_src/include -o ove
klee --libc=uclibc --posix-runtime overflow.bc
cat klee-last/*.err

Info:
        address: 65230480
        next: object at 65309008 of size 4
                MO2886[4] allocated at __user_main():  %i =
        prev: object at 65230464 of size 16
                MO3055[16] allocated at overflow():  %buffe
```

Heap Overflow

At a coarse level of granularity, nothing is different for a heap overflow.

As you want increasingly more sophisticated modeling of the behaviour of an allocator, CBMC and Klee both have serious issues.

Example

This is a "real" heap overflow against a program using a custom allocator.

```
for (; c < phdr->p_sz; c++) {
   if (c->nalloc) {
     c->nview = realloc(c->view, c->n_sz)
   }
}
```

`realloc` attempts to resize the block associated with `view`, however this allocator performs no check for sanity of new size is done.
This allocator attempts keeps an internal per-arena freelist. No `fwd` or `bk` pointers (`size` however is kept as heap chunk metadata for reasons unknown).
You can leverage this for a 2-in-1 memory leak and overwrite of any address which occurs *after* the block.

But. . .

There is no direct overwrite here and the allocator cannot be traced (no sound approximation could be generated in a reasonable time either).
In the end, I couldn't figure out how to make this work.

A Broader Problem

This is a broader problem - you often need a fine-grained model of the behavior of a program and symbolic analysis can't provide that in many cases.

That's It

Questions