Finding vulnerabilities
by fuzzing, dynamic and static analysis

Brandon Azad
Stanford CS155 guest lecture
April 17, 2024

About me

2014: Took this class as a student

2017: Course assistant for this class

Feel free to blame me for “Part 3” of Project 1 :-)

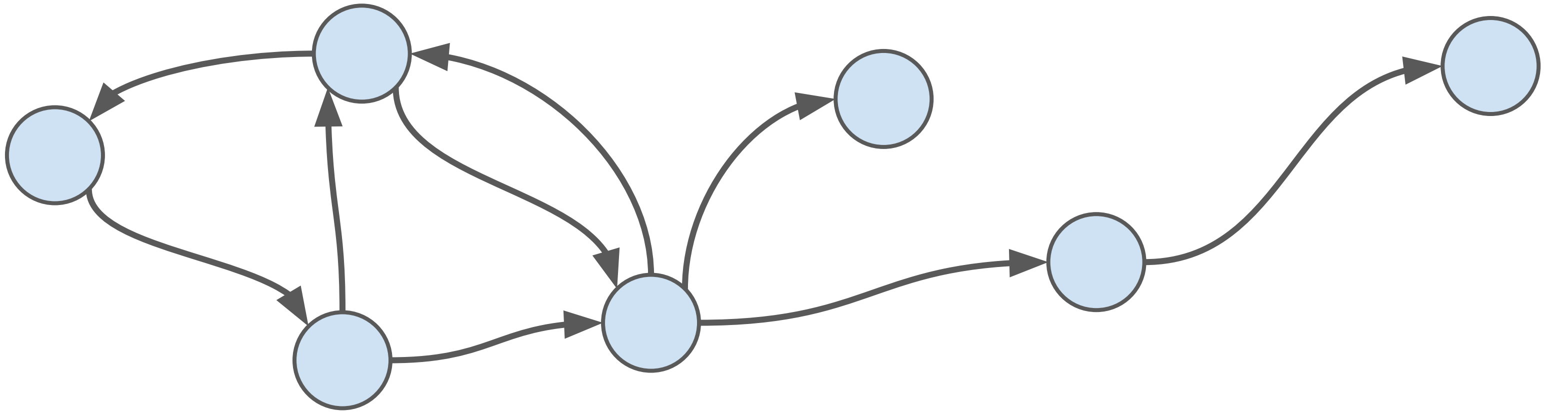
2018: Joined Google Project Zero

Mission: Make 0-day hard

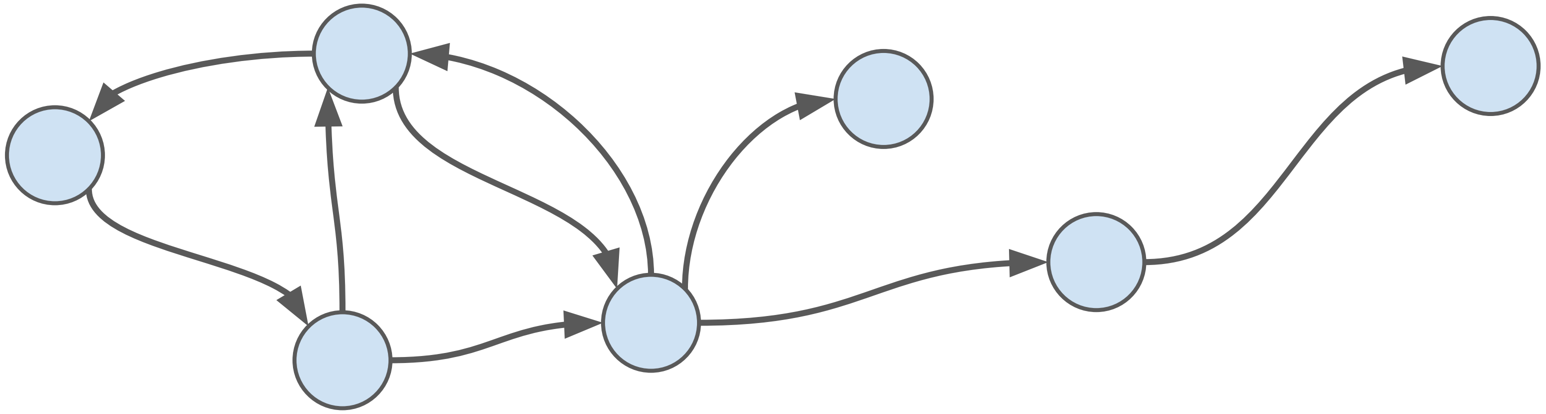
2020-now: Building mitigations, writing exploits, thinking about fundamentals

Conceptualizing vulnerabilities and exploits

Computer programs: finite state machines

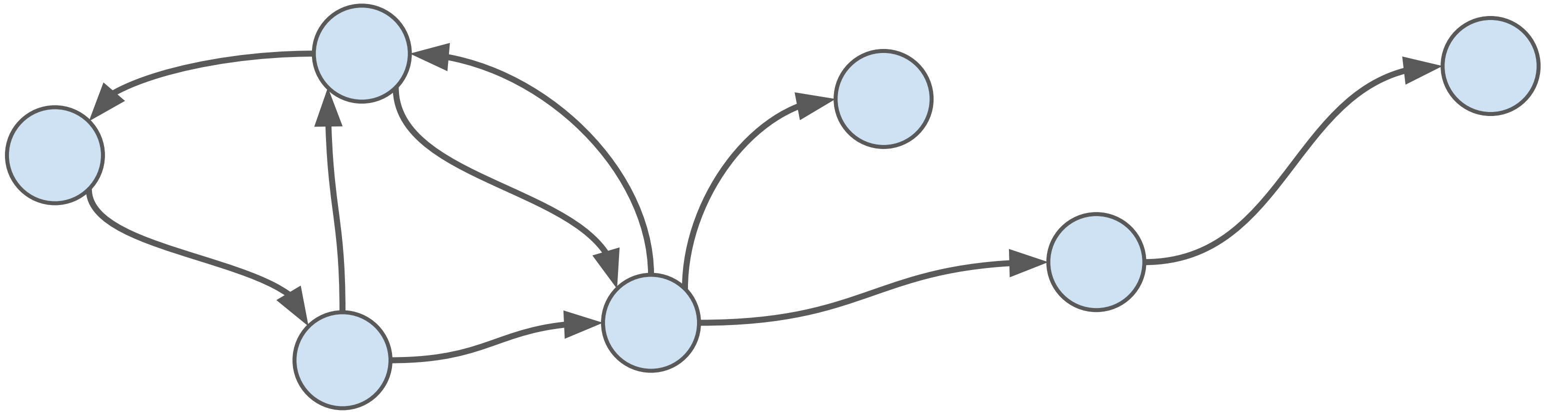


Computer programs: finite state machines



This is a *conceptual* state machine describing the *intended* operation of the program

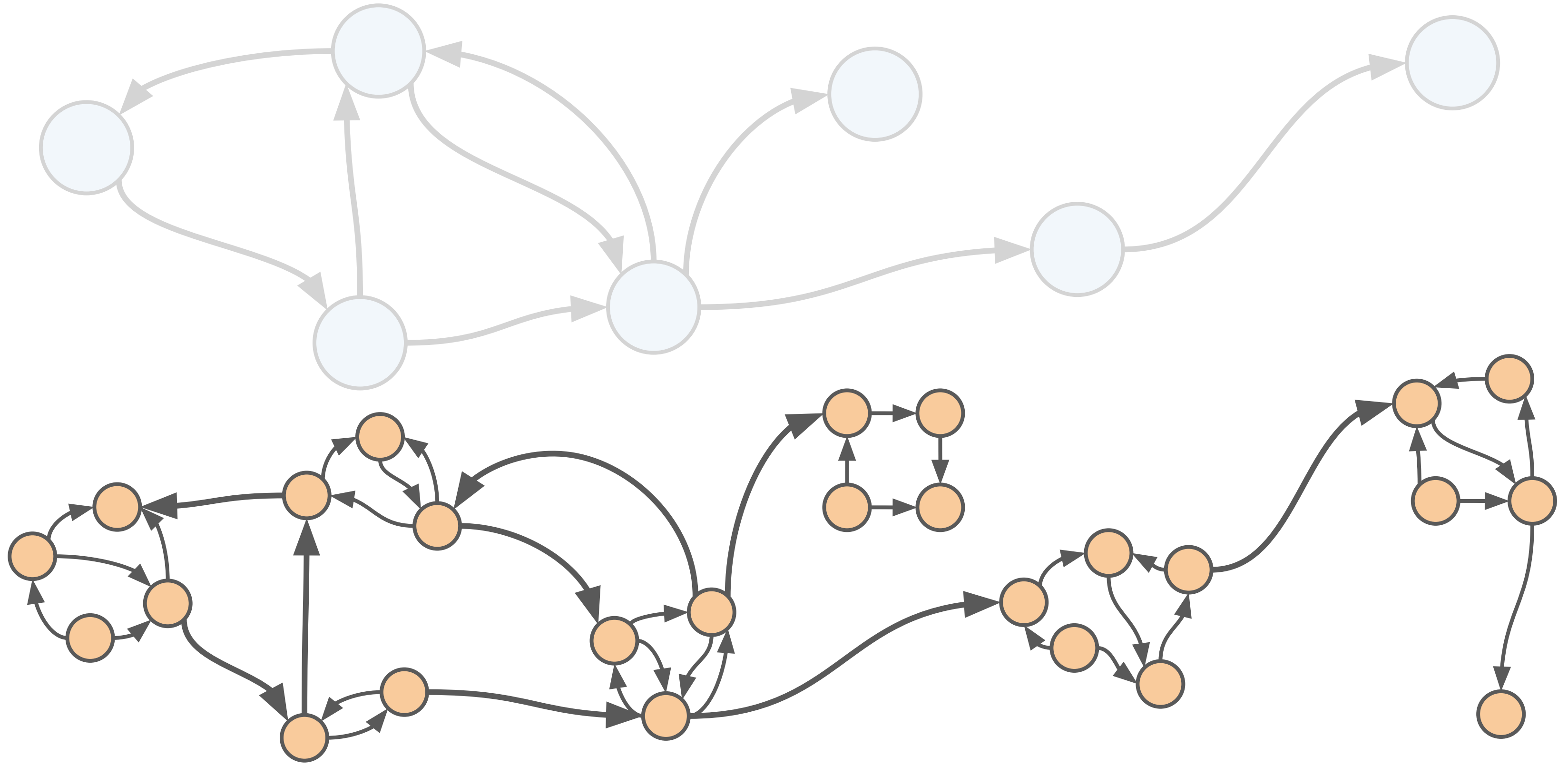
Computer programs: finite state machines



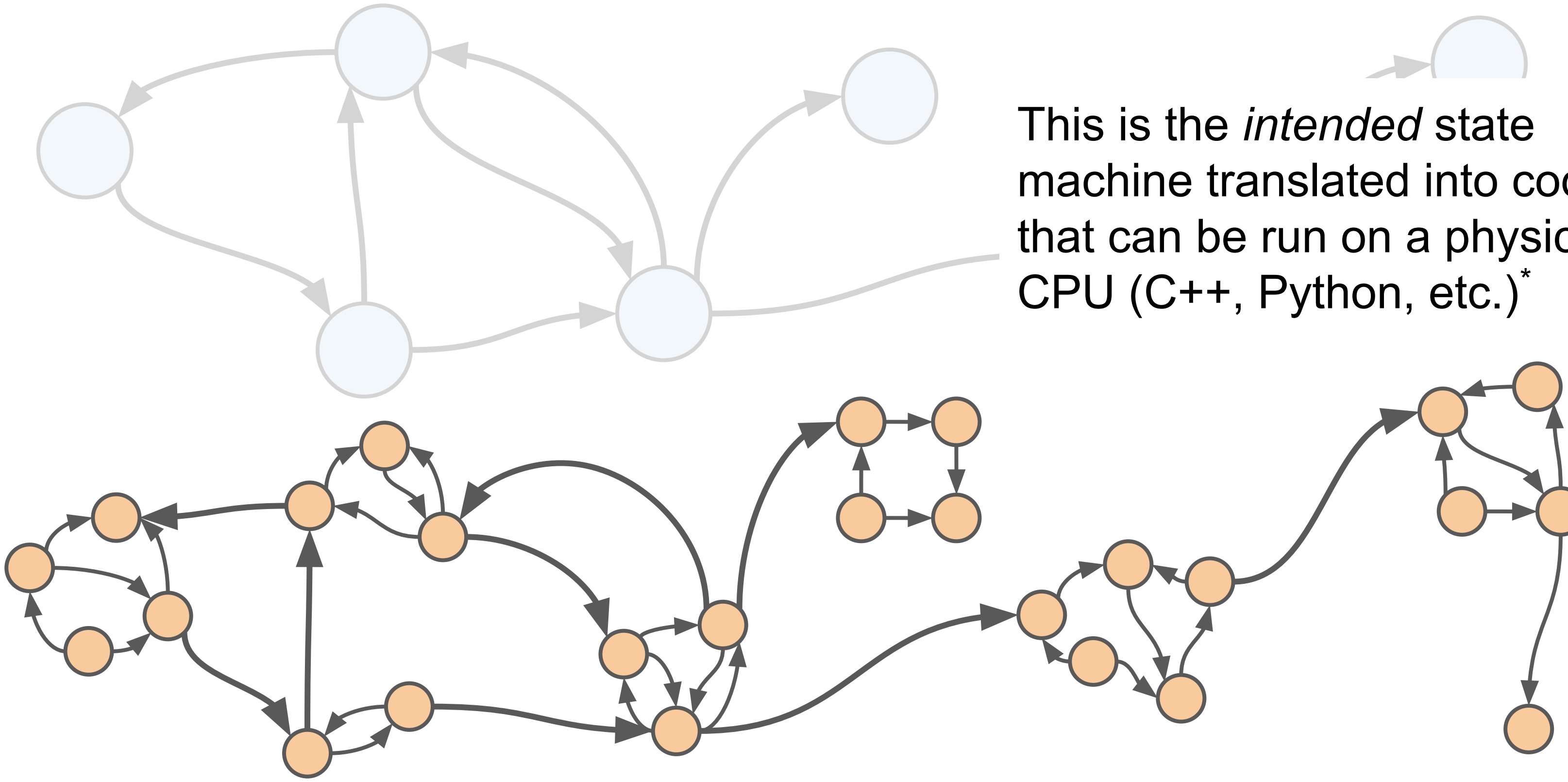
This is a *conceptual* state machine describing the *intended* operation of the program

A physical CPU cannot directly execute this abstract state machine

Running code: state machines emulating state machines



Running code: state machines emulating state machines

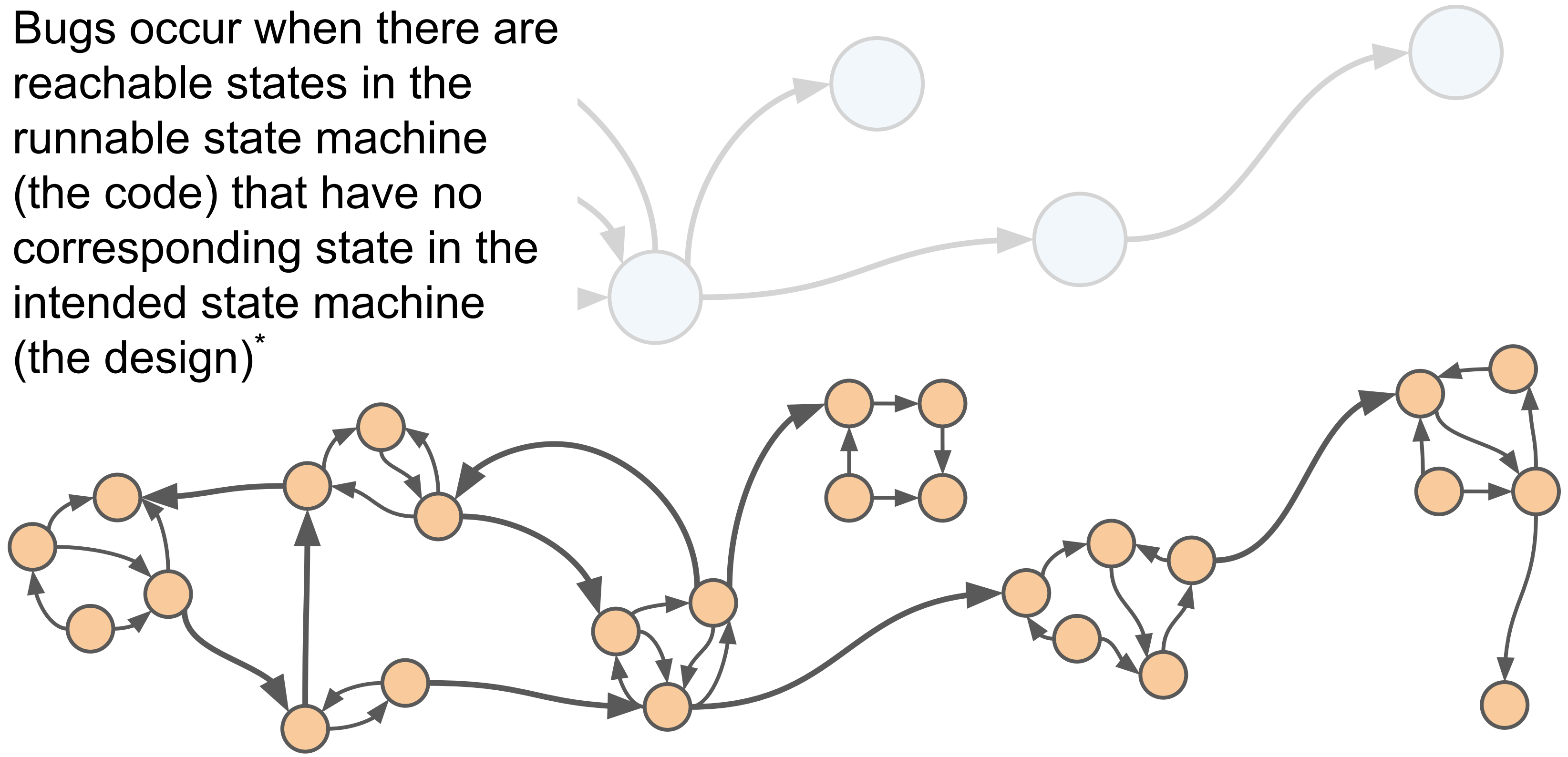


This is the *intended* state machine translated into code that can be run on a physical CPU (C++, Python, etc.)*

* Not quite true: that code still needs to be translated to machine code, which introduces another level of state machines emulating state machines

Running code: state machines emulating state machines

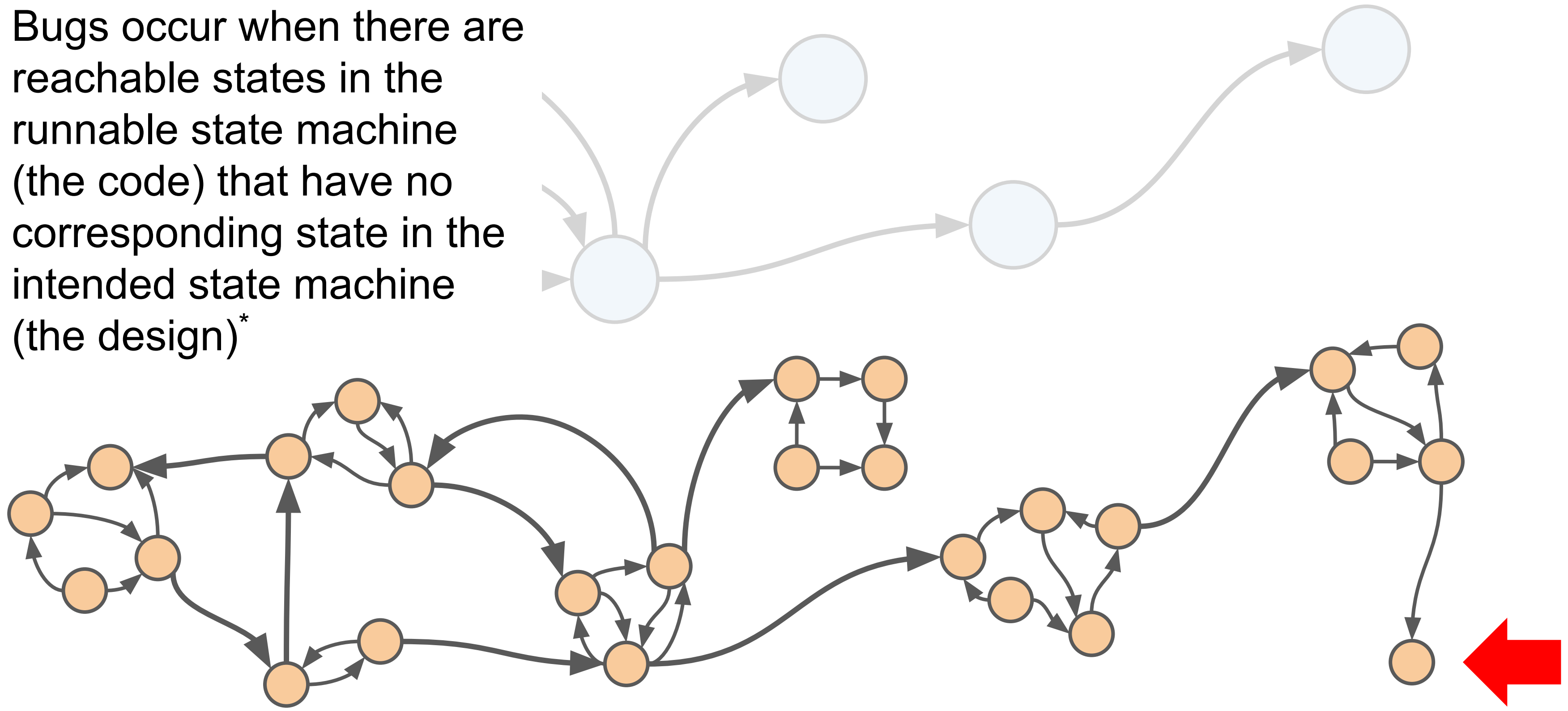
Bugs occur when there are reachable states in the runnable state machine (the code) that have no corresponding state in the intended state machine (the design)*



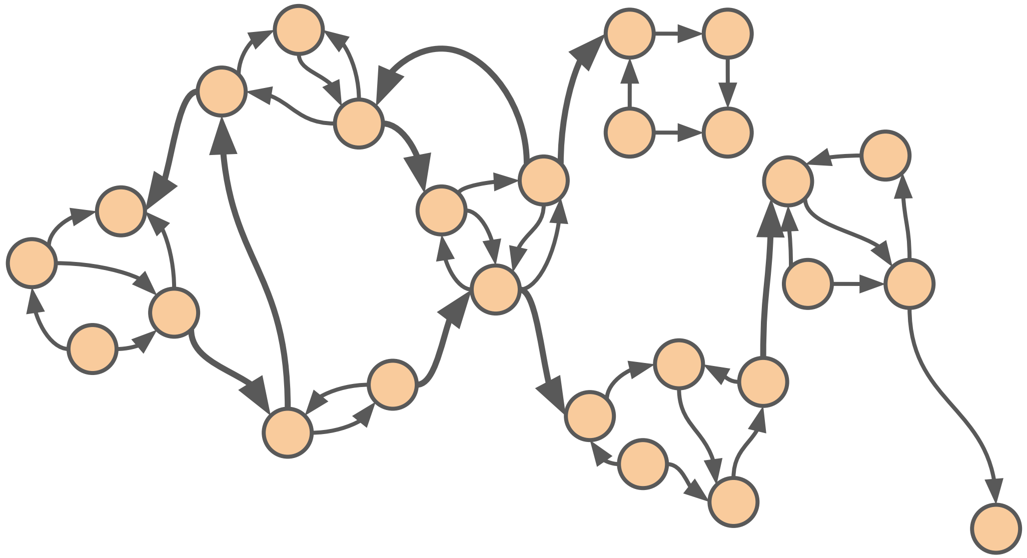
* Not the full picture: the initial design itself could have issues (design issues) which still count as software bugs

Running code: state machines emulating state machines

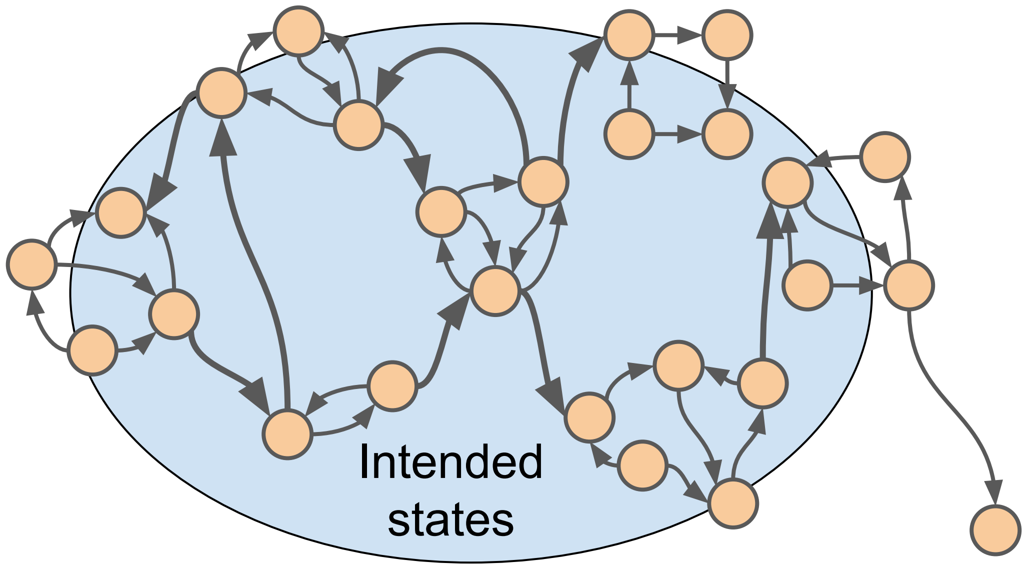
Bugs occur when there are reachable states in the runnable state machine (the code) that have no corresponding state in the intended state machine (the design)*



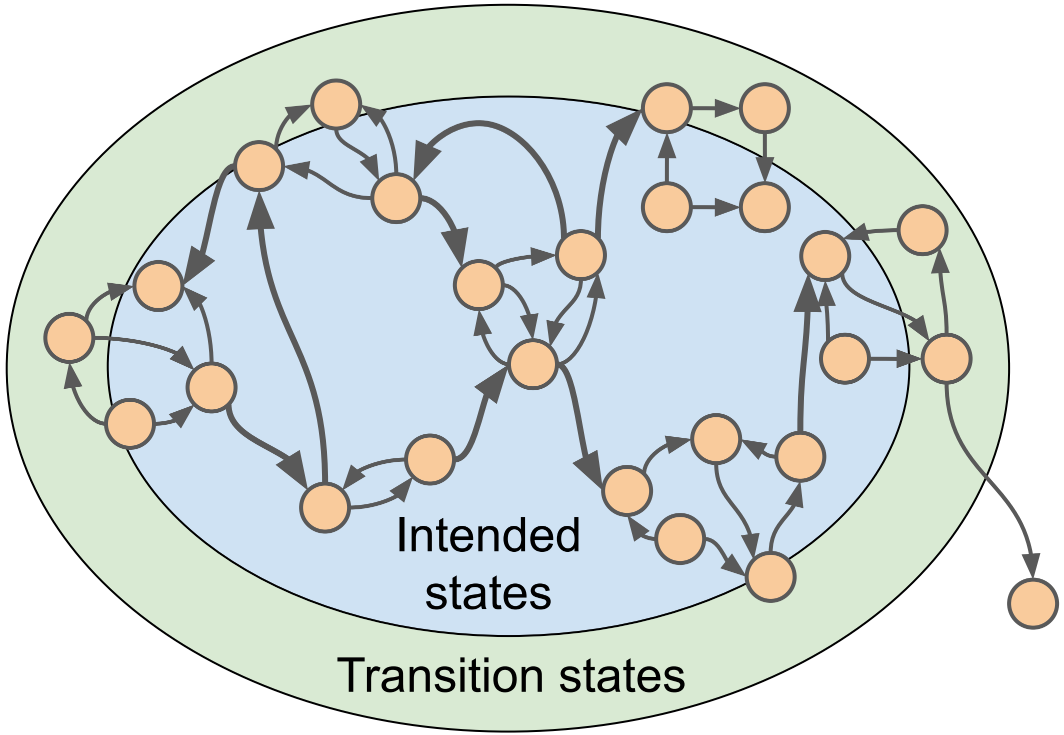
Classifying states



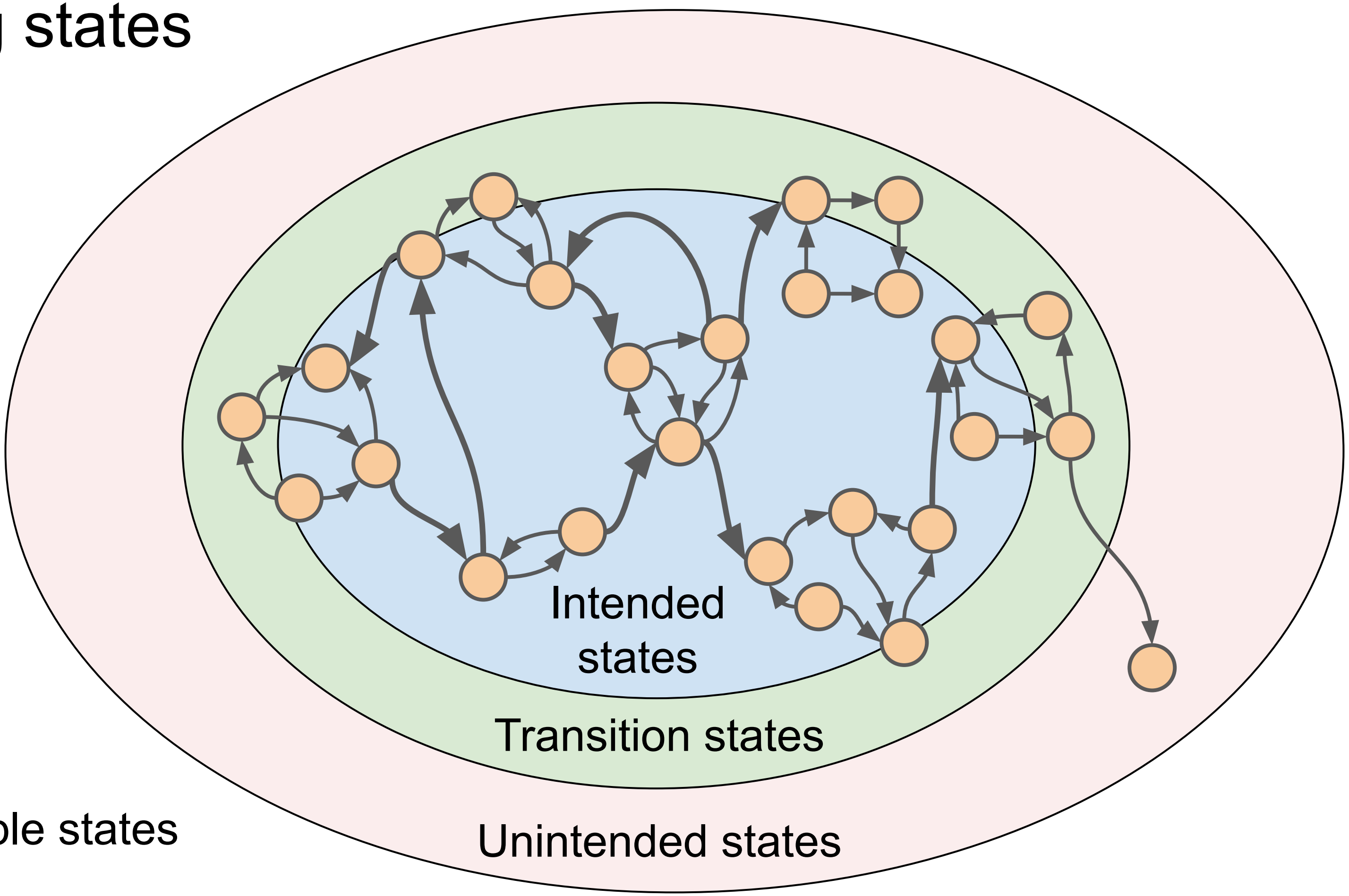
Classifying states



Classifying states

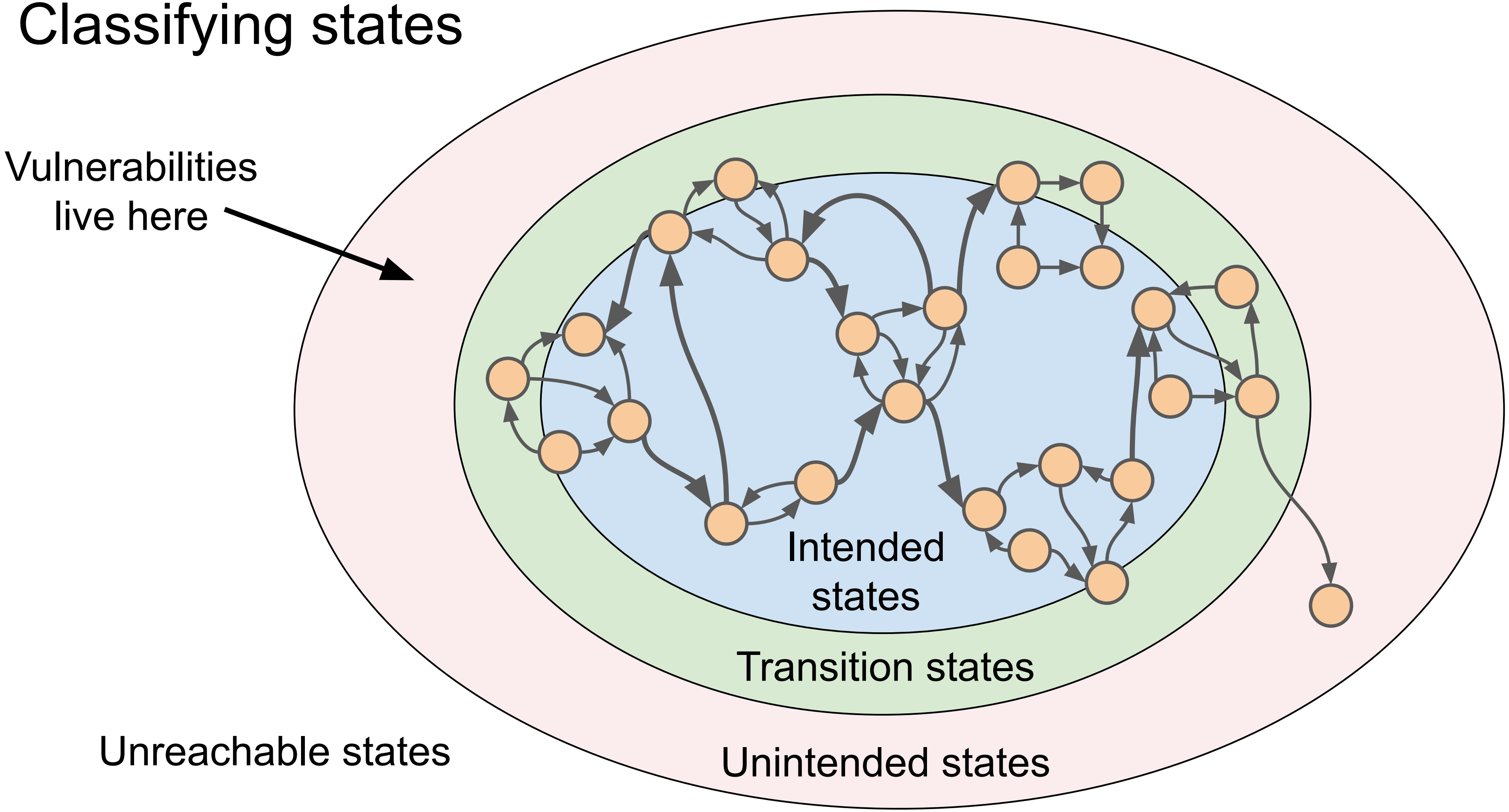


Classifying states



Unreachable states

Classifying states

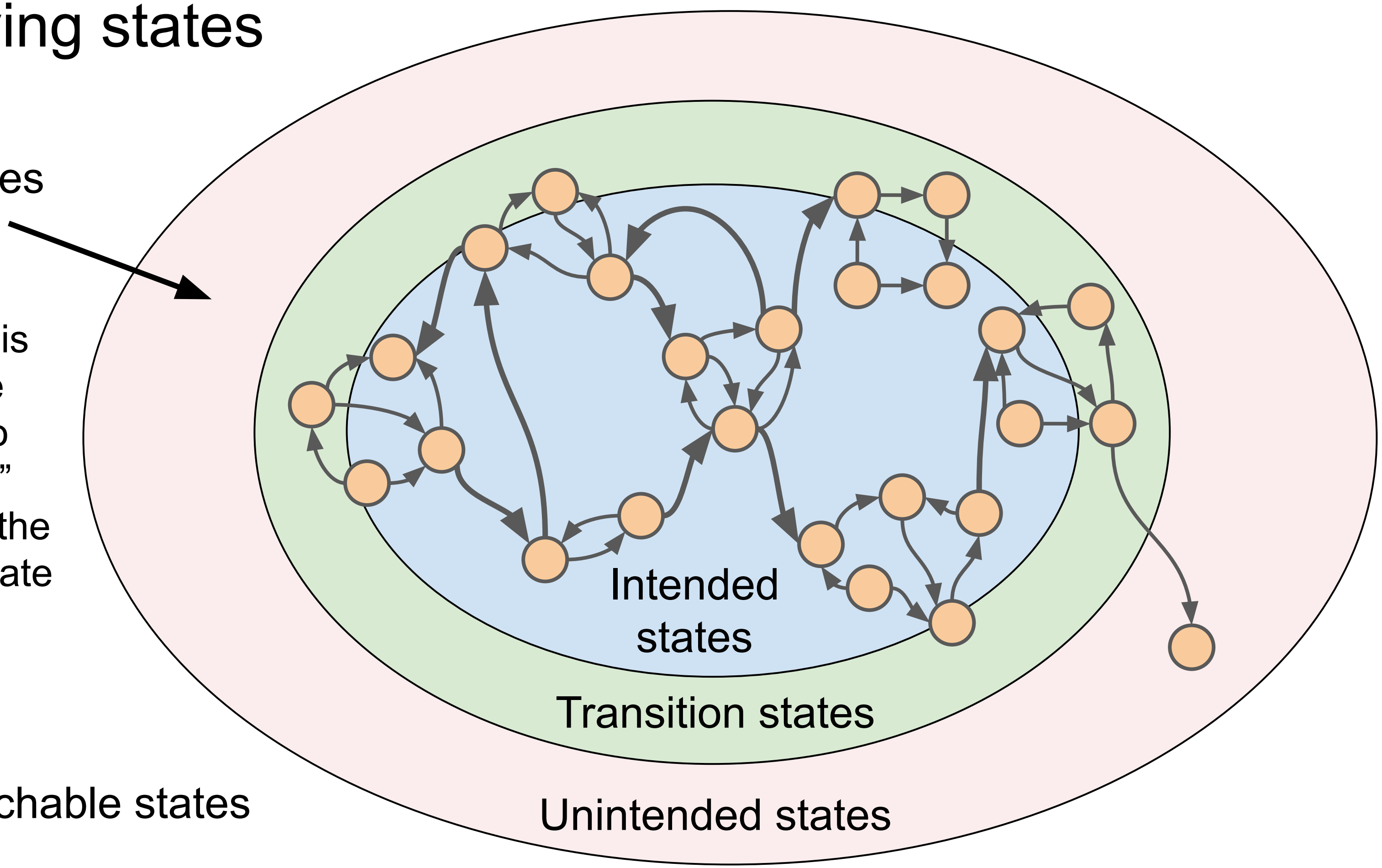


Classifying states

Vulnerabilities live here

Exploitation is making the program do “interesting” transitions in the unintended state space

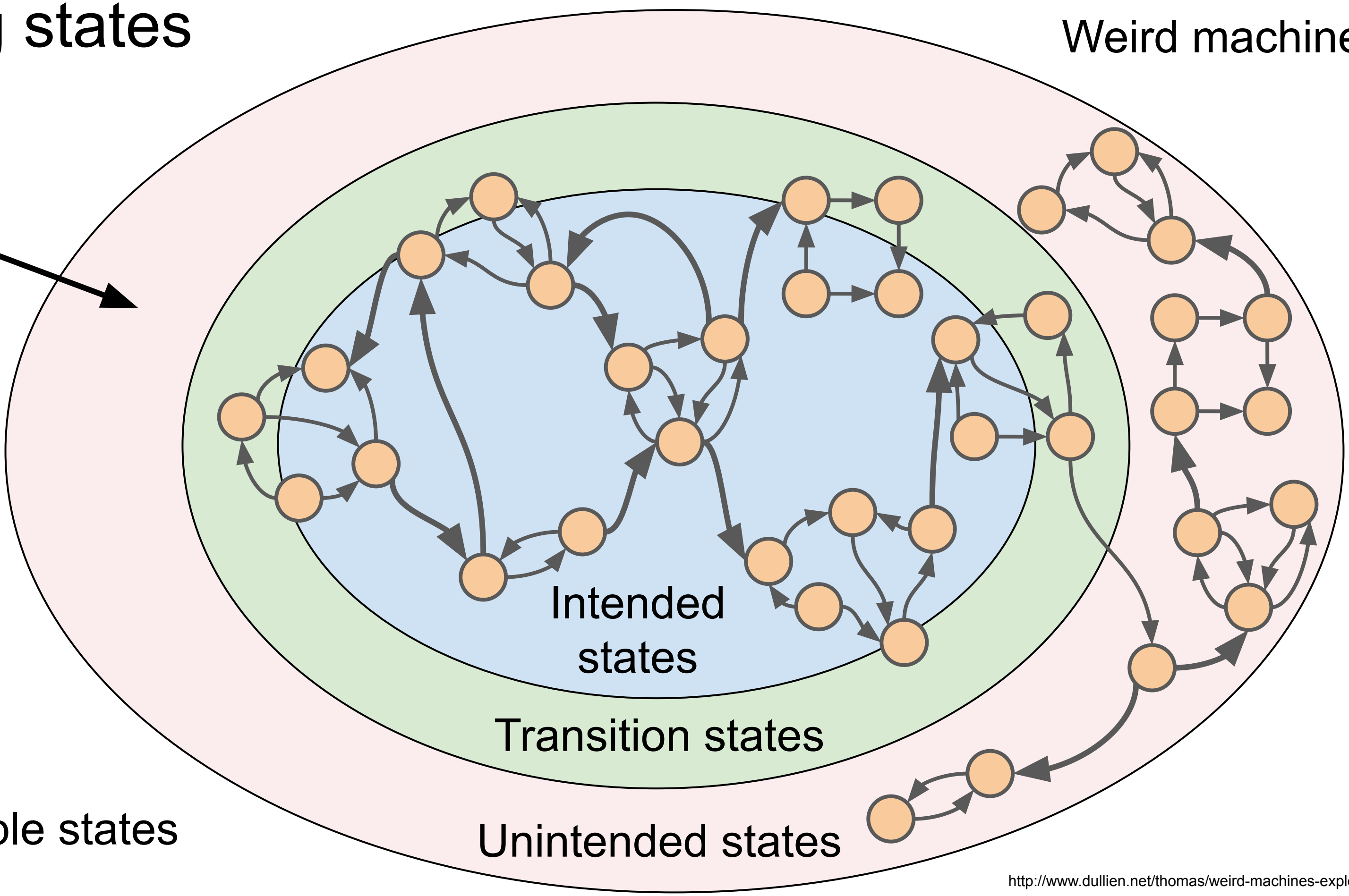
Unreachable states



Classifying states

Vulnerabilities live here

Exploitation is making the program do "interesting" transitions in the unintended state space



Unreachable states

http://www.dullien.net/thomas/weird-machines-exploitability.pdf

Common categories of software bugs

Design issue: The conceptual state machine does not meet the intended goals

The firewall’s remote interface is designed with a hardcoded admin password

Functionality bug: The code has bad transitions but only between validly represented states

The save button code is broken, no transition to “saving the file” state

Implementation bug: Code introduces new states not represented in the conceptual state machine

Lack of length checks introduces new “stack corruption” state

Other ways to reach unintended states

Hardware fault: The hardware suffers a glitch that causes a transition to an unintended state *even if the code is perfect*

> A cosmic ray causes a bit flip in a voting machine's memory, causing a state where one candidate has an impossible number of votes

Transmission error: The code is correct but is corrupted in-flight

> A program downloaded from the internet suffers packet corruption, so the program that is run has a different state machine from the one that was sent

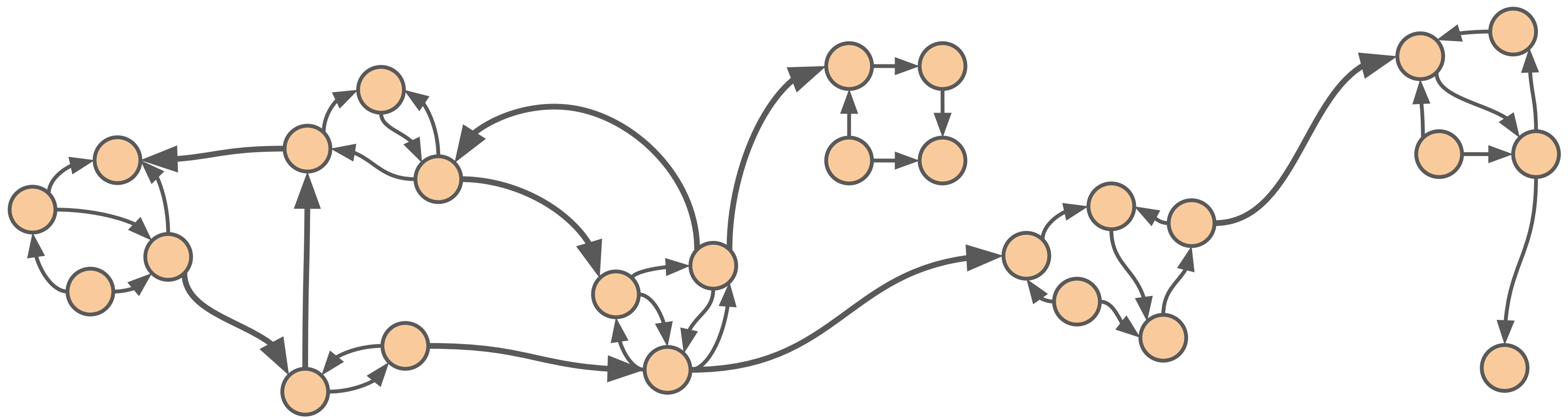
This list is not intended to be exhaustive; merely to illustrate the myriad ways that unintended states may enter a system; deciding which ones to defend against is one step of proper threat modeling

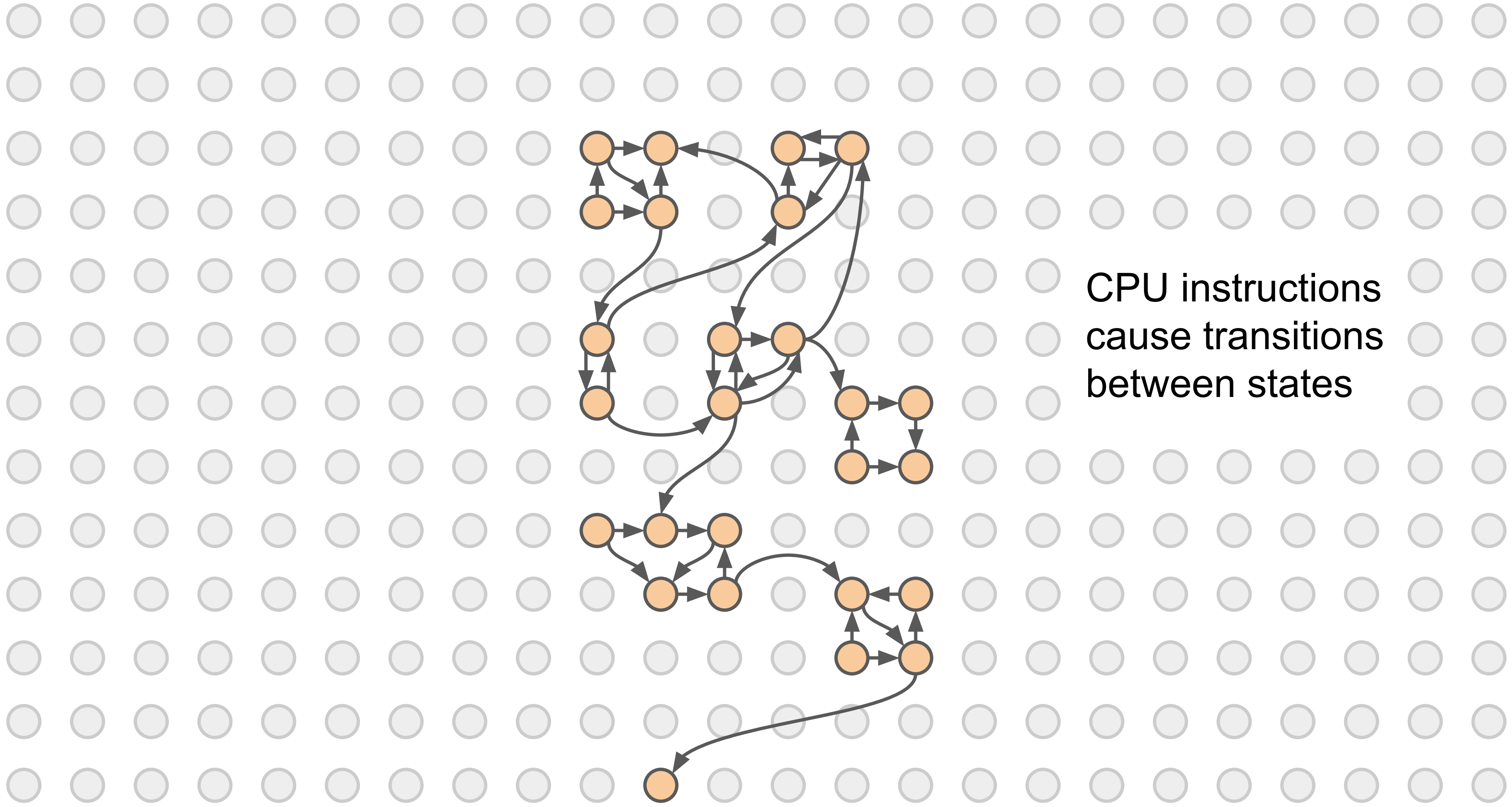
How to conceptualize this state space?

Assuming a computer with 16GB of memory, the number of nodes is (at least):

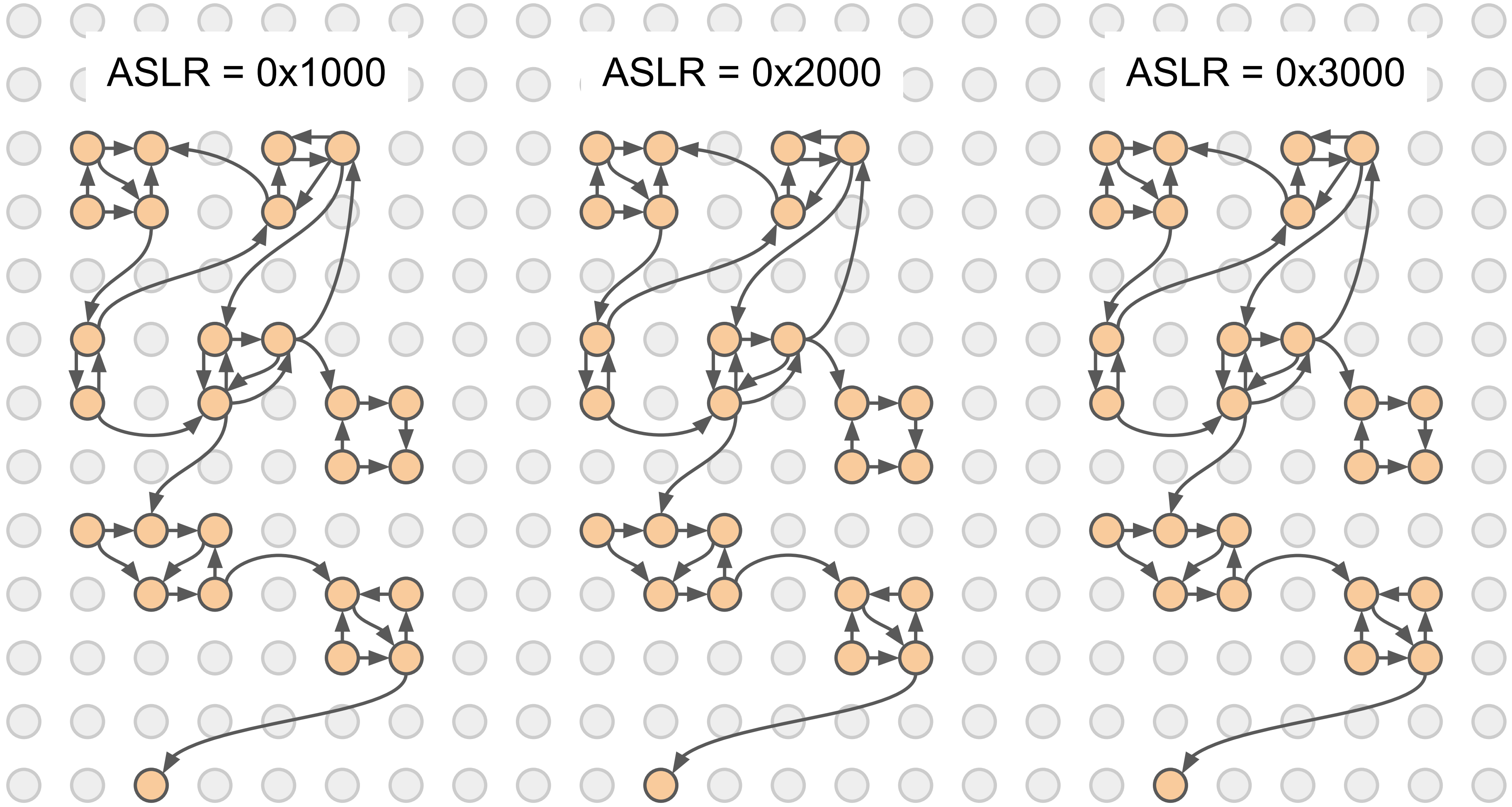
$2^{16 \times 1024 \times 1024 \times 1024} = 2^{17179869184}$ (a number with 5 billion digits)

This is very big



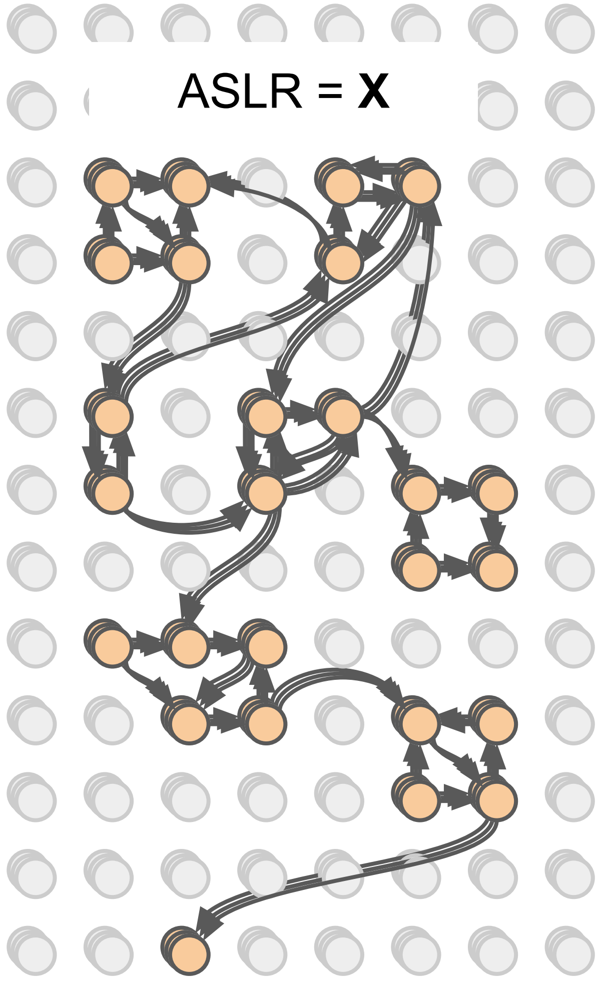


CPU instructions cause transitions between states


ASLR = 0x1000
ASLR = 0x2000
ASLR = 0x3000

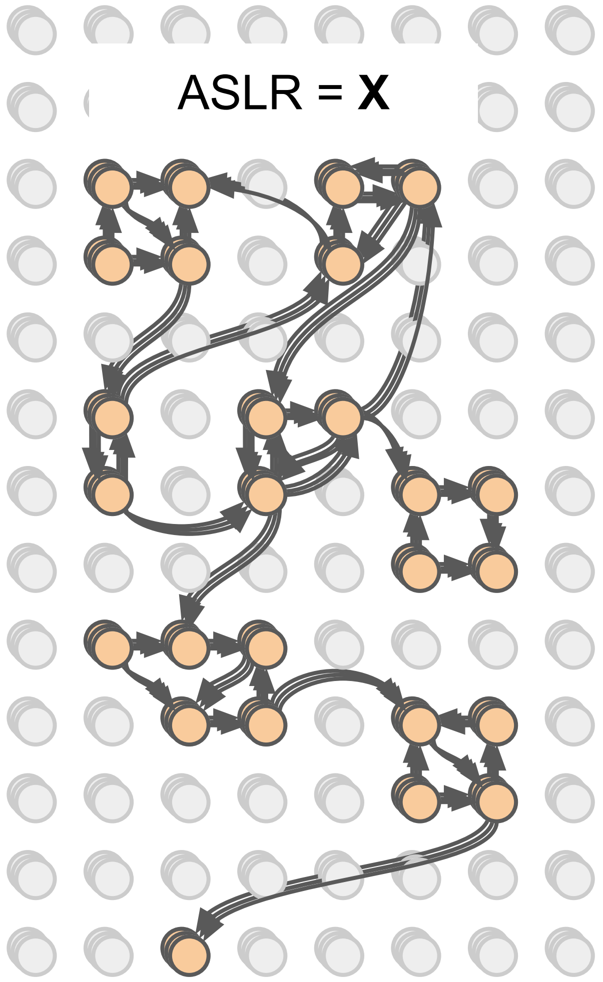
We can “quotient out” less relevant features of the full state space

But it’s still too large for humans to comprehend



We can “quotient out” less relevant features of the full state space

But it’s still too large for humans to comprehend



For any interesting program, it is essentially impossible to manually explore the full state space to find the unintended states

Fuzzing

Fuzzing

Find bugs in a program by feeding it random, corrupted, or unexpected data

Idea: Random inputs will explore a “large” part of the state space*

- Some unintended states are observable as crashes (**`SIGSEGV`**, **`abort()`**)
- Any crash is a bug, but only some bugs are exploitable

Works best on programs that parse files or process complex input data

* Or at least, large compared to manual analysis, and a very different portion of the state space than what humans tend to reason about when reading code

Fuzzing example

Fuzzing can be as simple as:

```
cat /dev/random | head -c 512 > rand.jpeg; open rand.jpeg
```

How could we do better?

Randomly corrupt real JPEG files

Reference the JPEG spec so that we generate only “JPEG-looking” data

Look at the JPEG parser to see how deep we’re getting in the code

Common fuzzing strategies

Mutation-based fuzzing

Randomly mutate test cases from some corpus of input files

Generation-based (smart) fuzzing

Generate test cases based on a specification for the input format

Coverage guided fuzzing

Measure code coverage of test cases to guide fuzzing towards new (unexplored) program states

This is neither an exhaustive list nor a rigid taxonomy: fuzzers often employ multiple strategies

Mutation-based fuzzing

Randomly mutate test cases from some corpus of input files

1. Collect a corpus of inputs that explores as many states as possible
2. Perturb inputs randomly, possibly guided by heuristics

    Modify: bit flips, integer increments

    Substitute: small integers, large integers, negative integers

3. Run the program on the inputs and check for crashes
4. Go back to step 2

Can mutation-based “dumb” fuzzing ever be successful?

This is my go-to “I need a fuzzer running in 10 minutes” code in 2024:

```
void havoc(const uint8_t *buf, size_t size) {
  switch (rnd(0,4)) {
    case 0: buf[rnd(0,size)] ^= 1 << rnd(0,8);                       break;
    case 1: buf[rnd(0,size)] = rnd(0,0xff);                          break;
    case 2: *(uint32_t *)&buf[rnd(0,size-3)] += rnd_small(0,0xffff); break;
    case 3: *(uint32_t *)&buf[rnd(0,size-3)] -= rnd_small(0,0xffff); break;
  }
  if (rnd(0,4) != 0) havoc(buf, size);
}
```

It often finds a bug within 2 minutes

Dumb fuzzing is often way more successful than it has any right to be

Mutation-based fuzzing

Advantages

- Very simple and fast to set up and run
  - Just need some example inputs and a harness to run the target code
- *No reason not to start here and parallelize with more involved VR*

Limitations

- Works best against never-been-fuzzed targets
- Results depend strongly on the quality of the initial corpus
- Coverage will be shallow for formats with checksums or validation

Generation-based (smart) fuzzing

Generate test cases based on a specification for the input format

1. Convert a specification of the input format (RFC, etc.) into a generative procedure
2. Generate test cases according to the procedure and introduce random perturbations
3. Run the program on the inputs and check for crashes
4. Go back to step 2

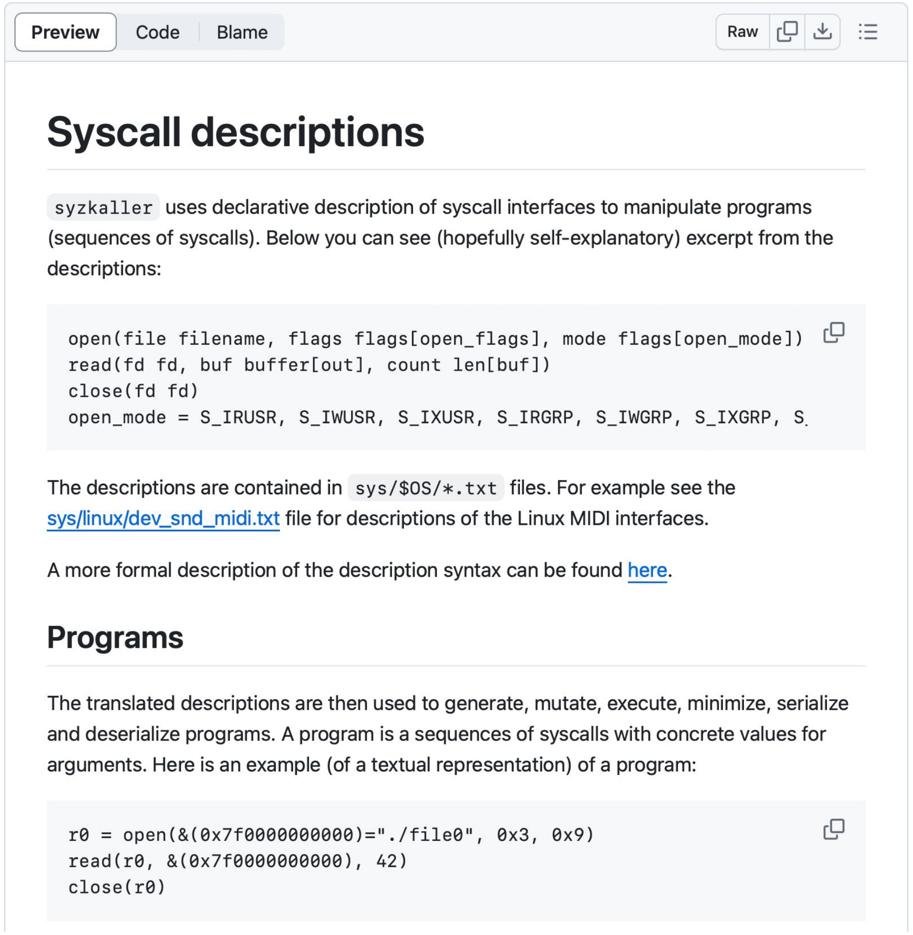
Syzkaller

A kernel system call fuzzer that uses test case generation and coverage

Test cases are sequences of syscalls generated from syscall descriptions

Runs the test case program in a VM

Kernel crashes in the VM indicate potential Local Privilege Escalation (LPE) vulnerabilities



Syscall descriptions

`syzkaller` uses declarative description of syscall interfaces to manipulate programs (sequences of syscalls). Below you can see (hopefully self-explanatory) excerpt from the descriptions:

```
open(file filename, flags flags[open_flags], mode flags[open_mode])
read(fd fd, buf buffer[out], count len[buf])
close(fd fd)
open_mode = S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S
```

The descriptions are contained in `sys/$OS/*.txt` files. For example see the sys/linux/dev_snd_midi.txt file for descriptions of the Linux MIDI interfaces.

A more formal description of the description syntax can be found here.

Programs

The translated descriptions are then used to generate, mutate, execute, minimize, serialize and deserialize programs. A program is a sequences of syscalls with concrete values for arguments. Here is an example (of a textual representation) of a program:

```
r0 = open(&(0x7f0000000000)="./file0", 0x3, 0x9)
read(r0, &(0x7f0000000000), 42)
close(r0)
```

https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md

Generation-based (smart) fuzzing

Advantages

Can get deeper coverage faster by leveraging knowledge of the input format

Input format/protocol complexity is not a limit on coverage depth

Limitations

Requires a lot of effort to set up

Successful fuzzers are often domain-specific

Coverage limited by accuracy of the spec; implementation may diverge

Coverage guided fuzzing

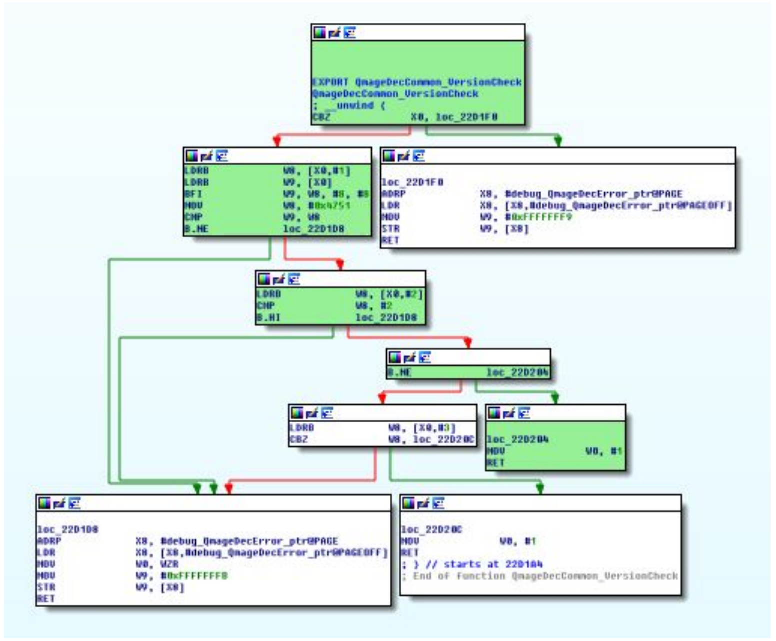
Key insight: code coverage is a useful metric, why not use it as **feedback** to guide fuzzing?

Prefer test cases that reach new states

Basic block coverage: Has this basic block in the CFG been run?

Edge coverage: Has this branch been taken?

Path coverage: Has this particular path through the program been taken?



```
EXPORT QmageDecCommon_VersionCheck
QmageDecCommon_VersionCheck
; __unwind {
CBZ             X0, loc_22D1F0
```

```
LDRB            W8, [X0,#1]
LDRB            W9, [X0]
BFI             W9, W8, #8, #8
MOV             W8, #0x4751
CMP             W9, W8
B.NE            loc_22D1D8
```

```
loc_22D1F0
ADRP            X8, #debug_QmageDecError_ptr@PAGE
LDR             X8, [X8,#debug_QmageDecError_ptr@PAGEOFF]
MOV             W9, #0xFFFFFFF9
STR             W9, [X8]
RET
```

```
LDRB            W8, [X0,#2]
CMP             W8, #2
B.HI            loc_22D1D8
```

```
B.NE            loc_22D204
```

```
LDRB            W8, [X0,#3]
CBZ             W8, loc_22D20C
```

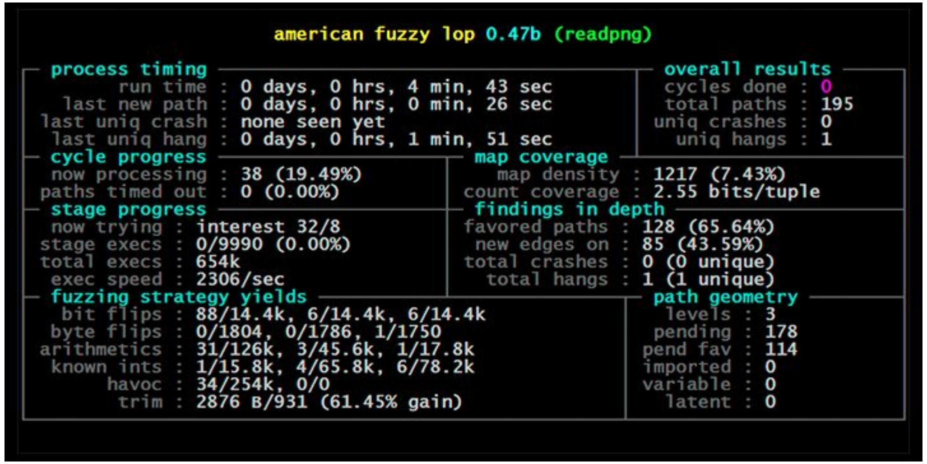
```
loc_22D204
MOV             W0, #1
RET
```

```
loc_22D1D8
ADRP            X8, #debug_QmageDecError_ptr@PAGE
LDR             X8, [X8,#debug_QmageDecError_ptr@PAGEOFF]
MOV             W0, WZR
MOV             W9, #0xFFFFFFF8
STR             W9, [X8]
RET
```

```
loc_22D20C
MOV             W0, #1
RET
; } // starts at 22D1A4
; End of function QmageDecCommon_VersionCheck
```

https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-2-effective-fuzzing-qmage.html

american fuzzy lop (AFL)



```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec        │  total paths : 195    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec        │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 38 (19.49%)       │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)        │  new edges on : 85 (43.59%)            │
│ total execs : 654k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec              │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k            │    levels : 3          │
│  byte flips : 0/1804, 0/1786, 1/1750                │   pending : 178        │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k             │  pend fav : 114        │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k             │  imported : 0          │
│       havoc : 34/254k, 0/0                          │  variable : 0          │
│        trim : 2876 B/931 (61.45% gain)              │    latent : 0          │
└─────────────────────────────────────────────────────┴────────────────────────┘
```

1. Compile the program with instrumentation to measure coverage

2. Trim the test cases in the queue to the smallest size that doesn't change the program behavior

3. Create new test cases by mutating the files in the queue using traditional fuzzing strategies

4. If new coverage is found in a mutated file, add it into the queue

5. Go back to step 2

https://lcamtuf.coredump.cx/afl/README.txt

Coverage guided fuzzing

Advantages

Very good at finding new program states, even if the initial corpus is limited

Combines well with other fuzzing strategies

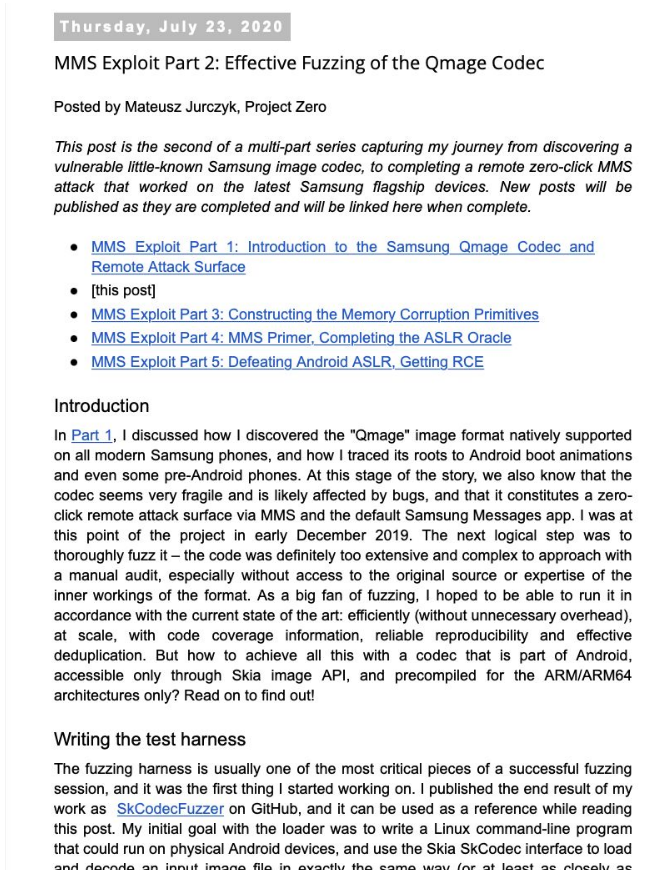
Wildly successful track record

Limitations

Not a panacea to bypass checksums or input validation

Still doesn’t find all types of bugs (e.g. race conditions)

Real world example: Fuzzing the Samsung Qmage codec



Thursday, July 23, 2020

MMS Exploit Part 2: Effective Fuzzing of the Qmage Codec

Posted by Mateusz Jurczyk, Project Zero

This post is the second of a multi-part series capturing my journey from discovering a vulnerable little-known Samsung image codec, to completing a remote zero-click MMS attack that worked on the latest Samsung flagship devices. New posts will be published as they are completed and will be linked here when complete.

- MMS Exploit Part 1: Introduction to the Samsung Qmage Codec and Remote Attack Surface
- [this post]
- MMS Exploit Part 3: Constructing the Memory Corruption Primitives
- MMS Exploit Part 4: MMS Primer, Completing the ASLR Oracle
- MMS Exploit Part 5: Defeating Android ASLR, Getting RCE

Introduction

In Part 1, I discussed how I discovered the "Qmage" image format natively supported on all modern Samsung phones, and how I traced its roots to Android boot animations and even some pre-Android phones. At this stage of the story, we also know that the codec seems very fragile and is likely affected by bugs, and that it constitutes a zero-click remote attack surface via MMS and the default Samsung Messages app. I was at this point of the project in early December 2019. The next logical step was to thoroughly fuzz it – the code was definitely too extensive and complex to approach with a manual audit, especially without access to the original source or expertise of the inner workings of the format. As a big fan of fuzzing, I hoped to be able to run it in accordance with the current state of the art: efficiently (without unnecessary overhead), at scale, with code coverage information, reliable reproducibility and effective deduplication. But how to achieve all this with a codec that is part of Android, accessible only through Skia image API, and precompiled for the ARM/ARM64 architectures only? Read on to find out!

Writing the test harness

The fuzzing harness is usually one of the most critical pieces of a successful fuzzing session, and it was the first thing I started working on. I published the end result of my work as SkCodecFuzzer on GitHub, and it can be used as a reference while reading this post. My initial goal with the loader was to write a Linux command-line program that could run on physical Android devices, and use the Skia SkCodec interface to load and decode an input image file in exactly the same way (or at least as closely as

In 2019, Mateusz Jurczyk discovered the Qmage image codec included on Samsung smartphones

- Reachable via zero-click MMS

The code looks fragile but the library is closed source

- Very few examples of Qmage files

Mateusz developed a harness to enable large-scale coverage-guided fuzzing of the Qmage codec

https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-2-effective-fuzzing-qmage.html

Fuzzing the Samsung Qmage image codec: harness

A **fuzzing harness** was written to call the interesting functions in the library and supply the test case input from the fuzzer

```
d2s:/data/local/tmp $ ./loader accessibility_light_easy_off.qmg
[+] Detected image characteristics:
[+] Dimensions:      344 x 344
[+] Color type:      4
[+] Alpha type:      3
[+] Bytes per pixel: 4
[+] codec->GetAndroidPixels() completed successfully
d2s:/data/local/tmp $
```

Could find bugs fuzzing on-device, but Mateusz wanted to fuzz at-scale

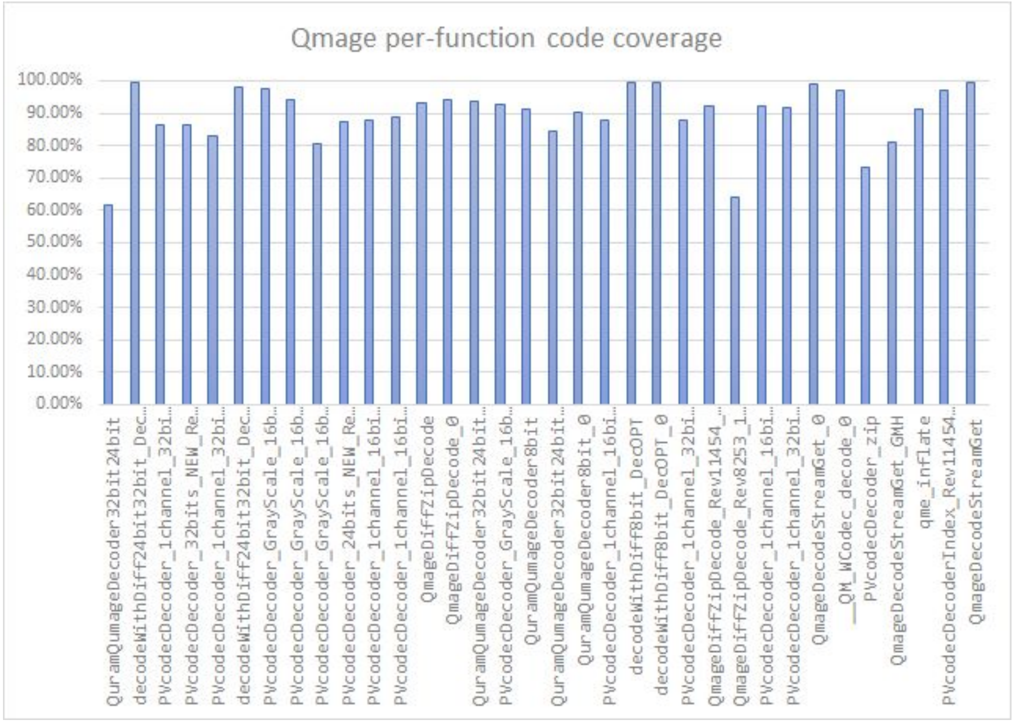
An emulator (qemu-aarch64) was used to run the harness and Qmage library on a Linux machine

- Easier to get 1000 Linux cores than 1000 Samsung Galaxy phones

Fuzzing the Samsung Qmage image codec: coverage

Code coverage was collected by modifying qemu-aarch64 to trace executed PC addresses

Coverage feedback compensated for the small number of initial test cases



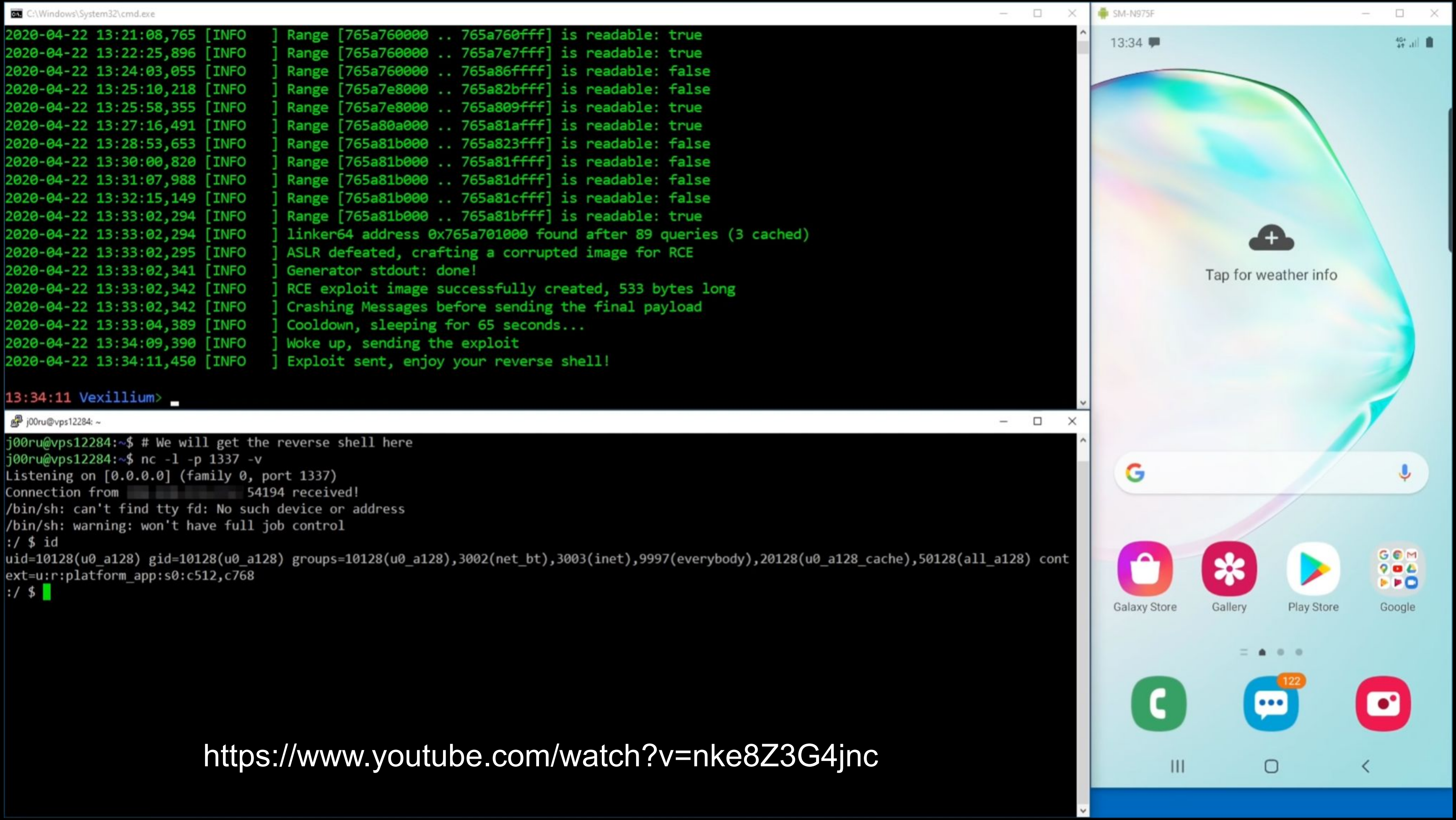
Fuzzing the Samsung Qmage image codec: results

Category	Count	Percentage
write	174	3.33%
read-memcpy	124	2.38%
read-vector	18	0.34%
read-32	3	0.06%
read-16	52	1.00%
read-8	34	0.65%
read-4	703	13.47%
read-2	393	7.53%
read-1	3322	63.66%
sigabrt	3	0.06%
null-deref	392	7.51%

4 weeks of fuzzing at scale

87.3% coverage of the Qmage codec

5218 unique crashes



```
2020-04-22 13:21:08,765 [INFO   ] Range [765a760000 .. 765a760fff] is readable: true
2020-04-22 13:22:25,896 [INFO   ] Range [765a760000 .. 765a7e7fff] is readable: true
2020-04-22 13:24:03,055 [INFO   ] Range [765a760000 .. 765a86ffff] is readable: false
2020-04-22 13:25:10,218 [INFO   ] Range [765a7e8000 .. 765a82bfff] is readable: false
2020-04-22 13:25:58,355 [INFO   ] Range [765a7e8000 .. 765a809fff] is readable: true
2020-04-22 13:27:16,491 [INFO   ] Range [765a80a000 .. 765a81afff] is readable: true
2020-04-22 13:28:53,653 [INFO   ] Range [765a81b000 .. 765a823fff] is readable: false
2020-04-22 13:30:00,820 [INFO   ] Range [765a81b000 .. 765a81ffff] is readable: false
2020-04-22 13:31:07,988 [INFO   ] Range [765a81b000 .. 765a81dfff] is readable: false
2020-04-22 13:32:15,149 [INFO   ] Range [765a81b000 .. 765a81cfff] is readable: false
2020-04-22 13:33:02,294 [INFO   ] Range [765a81b000 .. 765a81bfff] is readable: true
2020-04-22 13:33:02,294 [INFO   ] linker64 address 0x765a701000 found after 89 queries (3 cached)
2020-04-22 13:33:02,295 [INFO   ] ASLR defeated, crafting a corrupted image for RCE
2020-04-22 13:33:02,341 [INFO   ] Generator stdout: done!
2020-04-22 13:33:02,342 [INFO   ] RCE exploit image successfully created, 533 bytes long
2020-04-22 13:33:02,342 [INFO   ] Crashing Messages before sending the final payload
2020-04-22 13:33:04,389 [INFO   ] Cooldown, sleeping for 65 seconds...
2020-04-22 13:34:09,390 [INFO   ] Woke up, sending the exploit
2020-04-22 13:34:11,450 [INFO   ] Exploit sent, enjoy your reverse shell!

13:34:11 Vexillium> _
```

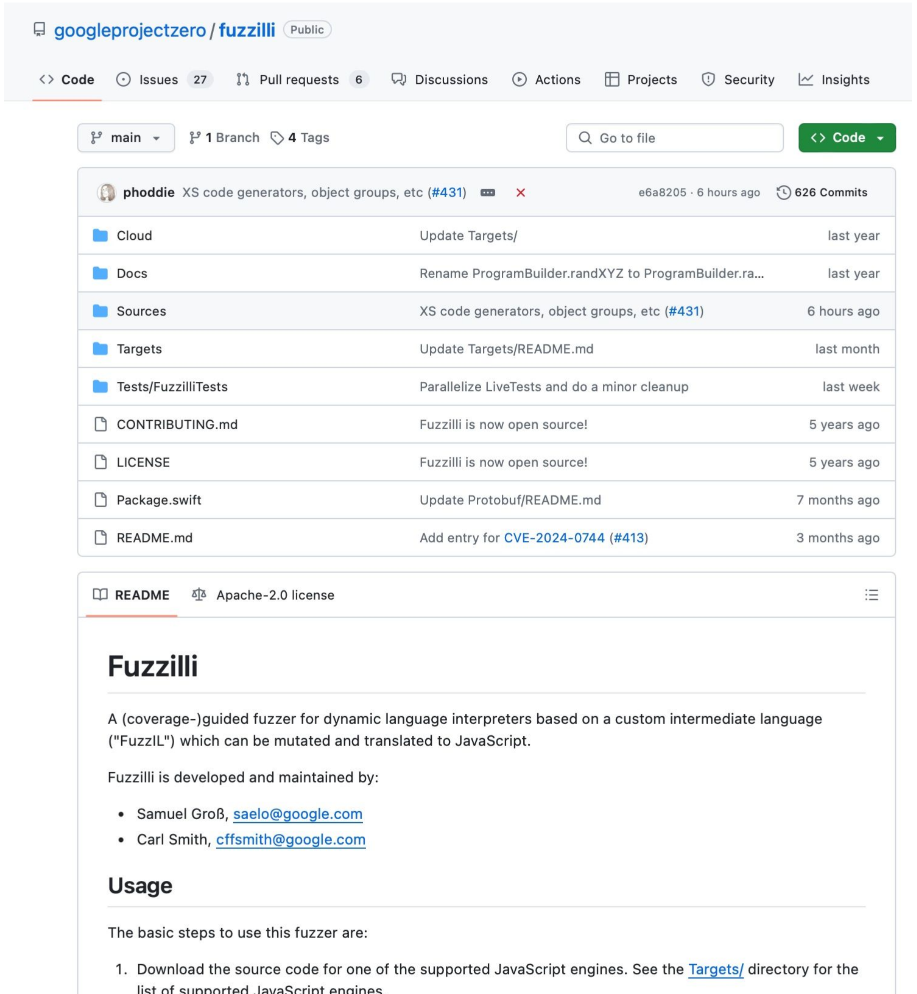
```
j00ru@vps12284:~$ # We will get the reverse shell here
j00ru@vps12284:~$ nc -l -p 1337 -v
Listening on [0.0.0.0] (family 0, port 1337)
Connection from [redacted] 54194 received!
/bin/sh: can't find tty fd: No such device or address
/bin/sh: warning: won't have full job control
:/ $ id
uid=10128(u0_a128) gid=10128(u0_a128) groups=10128(u0_a128),3002(net_bt),3003(inet),9997(everybody),20128(u0_a128_cache),50128(all_a128) cont
ext=u:r:platform_app:s0:c512,c768
:/ $
```

https://www.youtube.com/watch?v=nke8Z3G4jnc

Another cool fuzzer: Fuzzilli

Very successful JavaScript fuzzer

Principle: Translate JavaScript to a dense Intermediate Language (IL), and fuzz the IL



googleprojectzero / fuzzilli Public

Code · Issues 27 · Pull requests 6 · Discussions · Actions · Projects · Security · Insights

main · 1 Branch · 4 Tags · Go to file · Code

phoddie XS code generators, object groups, etc (#431) · e6a8205 · 6 hours ago · 626 Commits

Cloud	Update Targets/	last year
Docs	Rename ProgramBuilder.randXYZ to ProgramBuilder.ra...	last year
Sources	XS code generators, object groups, etc (#431)	6 hours ago
Targets	Update Targets/README.md	last month
Tests/FuzzilliTests	Parallelize LiveTests and do a minor cleanup	last week
CONTRIBUTING.md	Fuzzilli is now open source!	5 years ago
LICENSE	Fuzzilli is now open source!	5 years ago
Package.swift	Update Protobuf/README.md	7 months ago
README.md	Add entry for CVE-2024-0744 (#413)	3 months ago

README · Apache-2.0 license

Fuzzilli

A (coverage-)guided fuzzer for dynamic language interpreters based on a custom intermediate language ("FuzzIL") which can be mutated and translated to JavaScript.

Fuzzilli is developed and maintained by:

- Samuel Groß, saelo@google.com
- Carl Smith, cffsmith@google.com

Usage

The basic steps to use this fuzzer are:

1. Download the source code for one of the supported JavaScript engines. See the Targets/ directory for the list of supported JavaScript engines.

https://github.com/googleprojectzero/fuzzilli

Fuzzing summary

Off-the-shelf fuzzers are excellent at finding bugs

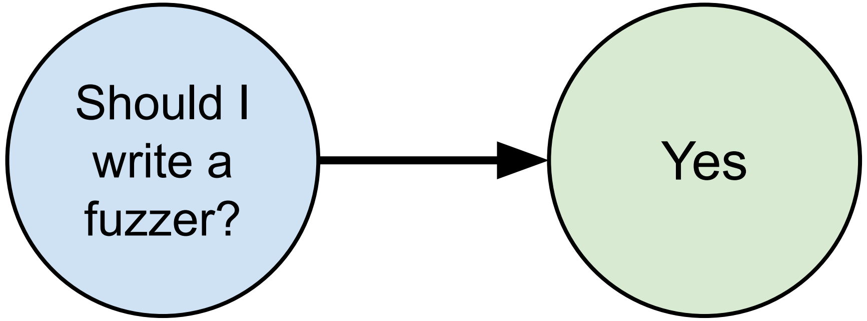
Custom fuzzers are also excellent at finding bugs

Different fuzzers often find different bugs

Easy to get started

Fuzzing doesn't find all types of bugs

This code parses untrusted data



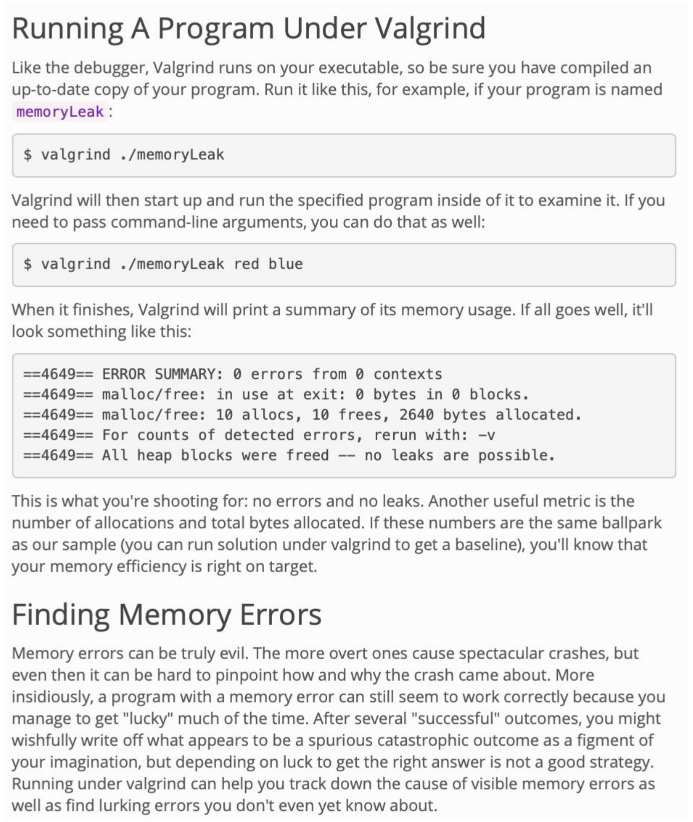
Dynamic analysis

Dynamic analysis

Analyze a program's behavior by actually running its code

- Sometimes combined with compile-time modifications like instrumentation
- Can modify the program's behavior dynamically
  - Useful for rapid experimentation

Often complements fuzzing very well



Running A Program Under Valgrind

Like the debugger, Valgrind runs on your executable, so be sure you have compiled an up-to-date copy of your program. Run it like this, for example, if your program is named `memoryLeak`:

```
$ valgrind ./memoryLeak
```

Valgrind will then start up and run the specified program inside of it to examine it. If you need to pass command-line arguments, you can do that as well:

```
$ valgrind ./memoryLeak red blue
```

When it finishes, Valgrind will print a summary of its memory usage. If all goes well, it'll look something like this:

```
==4649== ERROR SUMMARY: 0 errors from 0 contexts
==4649== malloc/free: in use at exit: 0 bytes in 0 blocks.
==4649== malloc/free: 10 allocs, 10 frees, 2640 bytes allocated.
==4649== For counts of detected errors, rerun with: -v
==4649== All heap blocks were freed -- no leaks are possible.
```

This is what you're shooting for: no errors and no leaks. Another useful metric is the number of allocations and total bytes allocated. If these numbers are the same ballpark as our sample (you can run solution under valgrind to get a baseline), you'll know that your memory efficiency is right on target.

Finding Memory Errors

Memory errors can be truly evil. The more overt ones cause spectacular crashes, but even then it can be hard to pinpoint how and why the crash came about. More insidiously, a program with a memory error can still seem to work correctly because you manage to get "lucky" much of the time. After several "successful" outcomes, you might wishfully write off what appears to be a spurious catastrophic outcome as a figment of your imagination, but depending on luck to get the right answer is not a good strategy. Running under valgrind can help you track down the cause of visible memory errors as well as find lurking errors you don't even yet know about.

https://web.stanford.edu/class/cs107/resources/valgrind.html

AddressSanitizer (ASan)

Fast memory error detector for C/C++ using compiler instrumentation and a runtime library that replaces **`malloc()`** to surround allocations with redzones

- Out-of-bounds accesses
- Use-after-free
- Double-free / invalid free

Typically 2x slowdown

`-fsanitize=address`

Not hardened! Don't turn on in production

```
==9901==ERROR: AddressSanitizer:heap-use-after-free on address 0x60700000dfb5 at pc 0x45917b
bp 0x7fff4490c700 sp 0x7fff4490c6f8
READ of size 1 at 0x60700000dfb5 thread T0
    #0 0x45917a in main use-after-free.c:5
    #1 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
    #2 0x459074 in _start (a.out+0x459074)
0x60700000dfb5 is located 5 bytes inside of 80-byte region [0x60700000dfb0,0x60700000e000)
freed by thread T0 here:
    #0 0x4441ee in __interceptor_free projects/compiler-rt/lib/asan/asan_malloc_linux.cc:64
    #1 0x45914a in main use-after-free.c:4
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
previously allocated by thread T0 here:
    #0 0x44436e in __interceptor_malloc projects/compiler-rt/lib/asan/asan_malloc_linux.cc:74
    #1 0x45913f in main use-after-free.c:3
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

https://github.com/google/sanitizers/wiki/AddressSanitizer

AddressSanitizer (ASan)

Fast memory error detector for C/C++ using compiler instrumentation and a runtime library th   n redzones

- Out-of-bour
- Use-after-fr
- Double-free

Pro tip: Once coverage guided fuzzing plateaus, run the generated corpus under ASan to find bugs the fuzzer missed!

```
s 0x60700000dfb5 at pc 0x45917b

ibc-2.15/csu/libc-start.c:226
0x60700000dfb0,0x60700000e000)
ib/asan/asan_malloc_linux.cc:64
ibc-2.15/csu/libc-start.c:226
    #0 0x44436e in __interceptor_malloc projects/compiler-rt/lib/asan/asan_malloc_linux.cc:74
    #1 0x45913f in main use-after-free.c:3
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

Typically 2x slov

`-fsanitize=address`

Not hardened! Don’t turn on in production

https://github.com/google/sanitizers/wiki/AddressSanitizer

ThreadSanitizer (TSan)

Data race detector for C/C++

Similar in principle to AddressSanitizer but for race conditions

High overhead

5-10x memory

5-15x slowdown

`-fsanitize=thread`

Also not hardened!

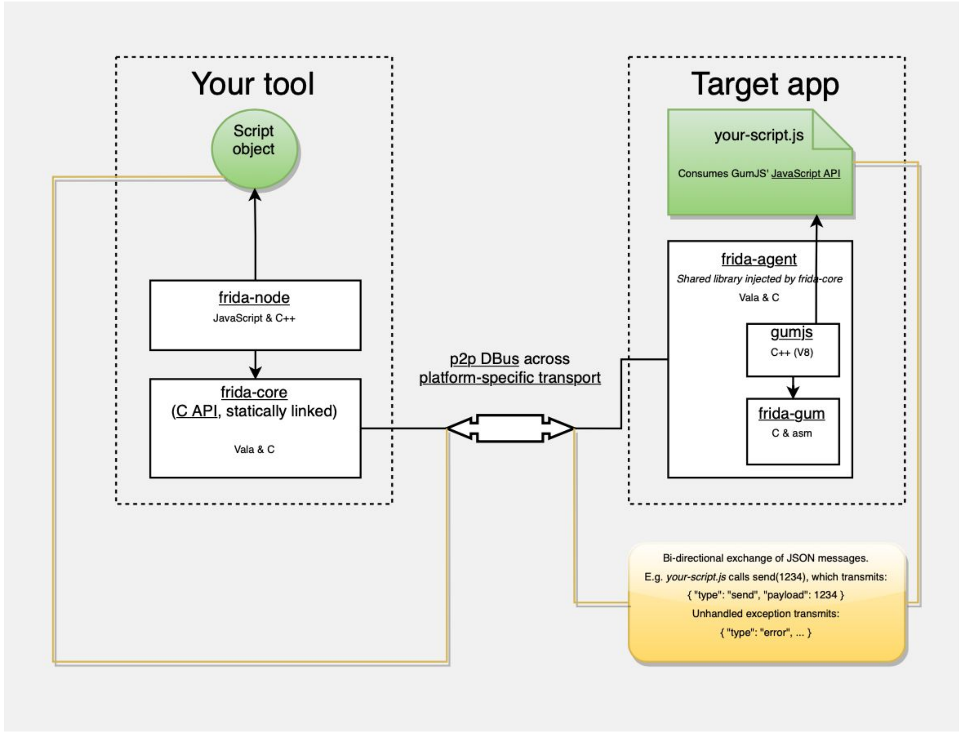
```
WARNING: ThreadSanitizer: data race (pid=19219)
  Write of size 4 at 0x7fcf47b21bc0 by thread T1:
    #0 Thread1 tiny_race.c:4 (exe+0x00000000a360)

  Previous write of size 4 at 0x7fcf47b21bc0 by main thread:
    #0 main tiny_race.c:10 (exe+0x00000000a3b4)

  Thread T1 (running) created at:
    #0 pthread_create tsan_interceptors.cc:705 (exe+0x00000000c790)
    #1 main tiny_race.c:9 (exe+0x00000000a3a4)
```

https://clang.llvm.org/docs/ThreadSanitizer.html

Frida


Your tool
Script object
frida-node
JavaScript & C++
frida-core
(C API, statically linked)
Vala & C
p2p DBus across platform-specific transport
Target app
your-script.js
Consumes GumJS' JavaScript API
frida-agent
Shared library injected by frida-core
Vala & C
gumjs
C++ (V8)
frida-gum
C & asm
Bi-directional exchange of JSON messages.
E.g. *your-script.js* calls send(1234), which transmits:
{ "type": "send", "payload": 1234 }
Unhandled exception transmits:
{ "type": "error", ... }

Dynamic instrumentation for closed-source binaries

- Execute custom scripts inside the analyzed process
- Hook functions, trace execution, modify behavior

Great way to fuzz internal functions without writing a harness

https://frida.re/docs/hacking/

Frida

```
Java.perform(function () {
  var Cipher = Java.use('javax.crypto.Cipher');
  var Exception = Java.use('java.lang.Exception');
  var Log = Java.use('android.util.Log');

  var init = Cipher.init.overload('int', 'java.security.Key');
  init.implementation = function (opmode, key) {
    var result = init.call(this, opmode, key);

    console.log('Cipher.init() opmode:', opmode, 'key:', key);
    console.log(stackTraceHere());

    return result;
  };

  function stackTraceHere() {
    return Log.getStackTraceString(Exception.$new());
  }
});
```

Dynamic instrumentation for closed-source binaries

- Execute custom scripts inside the analyzed process
- Hook functions, trace execution, modify behavior

Great way to fuzz internal functions without writing a harness

https://frida.re/docs/hacking/

Static analysis

Static analysis

Using a tool to analyze a program’s behavior without actually running it

Test whether a certain property holds or find places where it is violated

Static analysis can *prove* some properties about the program that fuzzing and dynamic analysis can’t

E.g., can prove that a program is free of NULL pointer dereferences

Despite lots of work in this area, there are countless interesting topics and huge scope for improvements!

Undecidability of static analysis

Goal: Determine whether a given program satisfies a given property

This is theoretically undecidable: it reduces to the halting problem!

```
def solve_halting_problem(P, a):
    def new_P():
        P(a)
        bug()
    return static_analyzer_for_bug(new_P)
```

Soundness and completeness

The best static analyzer can only satisfy one of the following:*

- **Soundness**: Everything that the static analyzer finds is a bug
  - But some bugs may be missed!
- **Completeness**: The static analyzer finds every bug
  - But there may be false positives!

Most static analyzers are neither sound nor complete

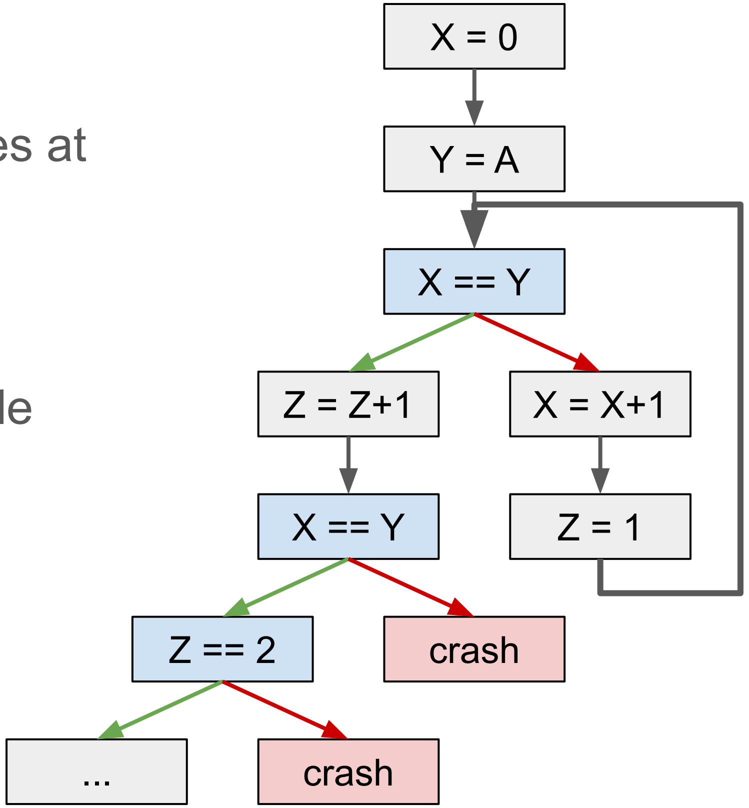
* We are assuming termination.

Data flow analysis

Determine the possible values of variables at points in the control flow graph

Approximations are usually needed

- Expressing the precise set of possible values may be arbitrarily complex

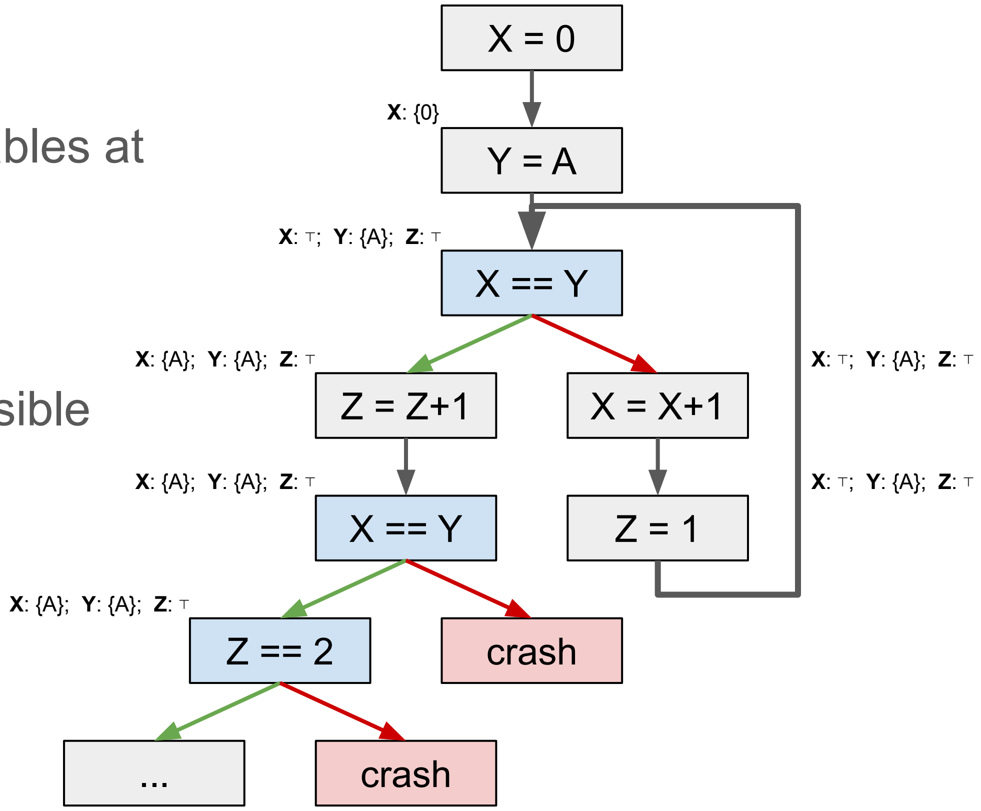


Data flow analysis

Determine the possible values of variables at points in the control flow graph

Approximations are usually needed

> Expressing the precise set of possible values may be arbitrarily complex

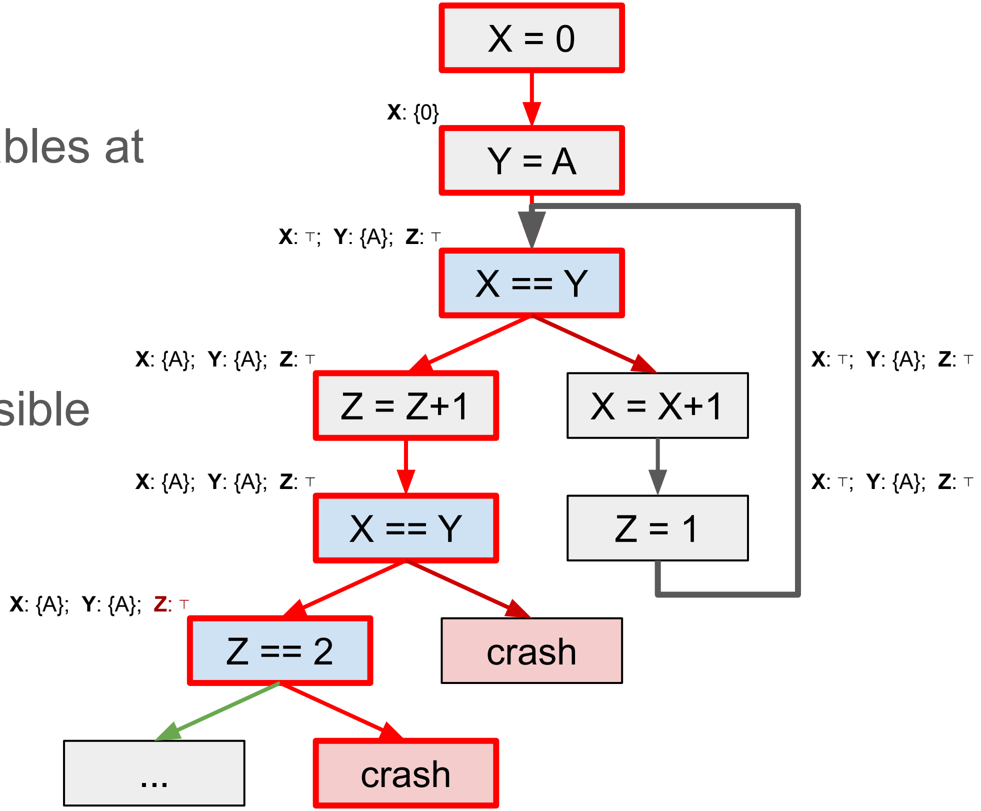


Data flow analysis

Determine the possible values of variables at points in the control flow graph

Approximations are usually needed

- Expressing the precise set of possible values may be arbitrarily complex



```
static int __vipx_ioctl_get_container(struct vs4l_container_list *karg,
    struct vs4l_container_list __user *uarg)
{
...
  ret = copy_from_user(karg, uarg, sizeof(*karg));
...
  ucon = karg->containers;
  size = karg->count * sizeof(*kcon);
  kcon = kzalloc(size, GFP_KERNEL);
...
  karg->containers = kcon;
  ret = copy_from_user(kcon, ucon, size);
  if (ret) {
    vipx_err("Copy failed [CONTAINER] (%d)\n", ret);
    goto p_err_free;
  }
  for (idx = 0; idx < karg->count; ++idx) {
    ubuf = kcon[idx].buffers;
    size = kcon[idx].count * sizeof(*kbuf);
    kbuf = kzalloc(size, GFP_KERNEL);
...
    kcon[idx].buffers = kbuf;
    ret = copy_from_user(kbuf, ubuf, size);
    if (ret) {
      vipx_err("Copy failed [CONTAINER] (%d)\n", ret);
      goto p_err_free;
    }
  }
...
  return 0;
p_err_free:
  for (idx = 0; idx < karg->count; ++idx)
    kfree(kcon[idx].buffers);
  kfree(kcon);
p_err:
  return ret;
}
```

Taint analysis

Identify sources of “tainted” data

- User/attacker input
- Reads from files/network

Check to see if tainted data flows into a “trusted sink”

```
memcpy(_, _, size)
free(ptr)
bzero(_, size)
```

```c
static int __vipx_ioctl_get_container(struct vs4l_container_list *karg,
    struct vs4l_container_list __user *uarg)
{
...
  ret = copy_from_user(karg, uarg, sizeof(*karg));
...
  ucon = karg->containers;
  size = karg->count * sizeof(*kcon);
  kcon = kzalloc(size, GFP_KERNEL);
...
  karg->containers = kcon;
  ret = copy_from_user(kcon, ucon, size);
  if (ret) {
    vipx_err("Copy failed [CONTAINER] (%d)\n", ret);
    goto p_err_free;
  }
  for (idx = 0; idx < karg->count; ++idx) {
    ubuf = kcon[idx].buffers;
    size = kcon[idx].count * sizeof(*kbuf);
    kbuf = kzalloc(size, GFP_KERNEL);
...
    kcon[idx].buffers = kbuf;
    ret = copy_from_user(kbuf, ubuf, size);
    if (ret) {
      vipx_err("Copy failed [CONTAINER] (%d)\n", ret);
      goto p_err_free;
    }
  }
...
  return 0;
p_err_free:
  for (idx = 0; idx < karg->count; ++idx)
    kfree(kcon[idx].buffers);
  kfree(kcon);
p_err:
  return ret;
}
```

Taint analysis

Identify sources of “tainted” data

- User/attacker input
- Reads from files/network

Check to see if tainted data flows into a “trusted sink”

```
memcpy(_, _, size)
free(ptr)
bzero(_, size)
```

```
static int __vipx_ioctl_get_container(struct vs4l_container_list *karg,
    struct vs4l_container_list __user *uarg)
{
...
  ret = copy_from_user(karg, uarg, sizeof(*karg));
...
  ucon = karg->containers;
  size = karg->count * sizeof(*kcon);
  kcon = kzalloc(size, GFP_KERNEL);
...
  karg->containers = kcon;
  ret = copy_from_user(kcon, ucon, size);
  if (ret) {
    vipx_err("Copy failed [CONTAINER] (%d)\n", ret);
    goto p_err_free;
  }
  for (idx = 0; idx < karg->count; ++idx) {
    ubuf = kcon[idx].buffers;
    size = kcon[idx].count * sizeof(*kbuf);
    kbuf = kzalloc(size, GFP_KERNEL);
...
    kcon[idx].buffers = kbuf;
    ret = copy_from_user(kbuf, ubuf, size);
    if (ret) {
      vipx_err("Copy failed [CONTAINER] (%d)\n", ret);
      goto p_err_free;
    }
  }
...
  return 0;
p_err_free:
  for (idx = 0; idx < karg->count; ++idx)
    kfree(kcon[idx].buffers);
  kfree(kcon);
p_err:
  return ret;
}
```

Taint analysis

Identify sources of “tainted” data

- User/attacker input
- Reads from files/network

Check to see if tainted data flows into a “trusted sink”

```
memcpy(_, _, size)
free(ptr)
bzero(_, size)
```

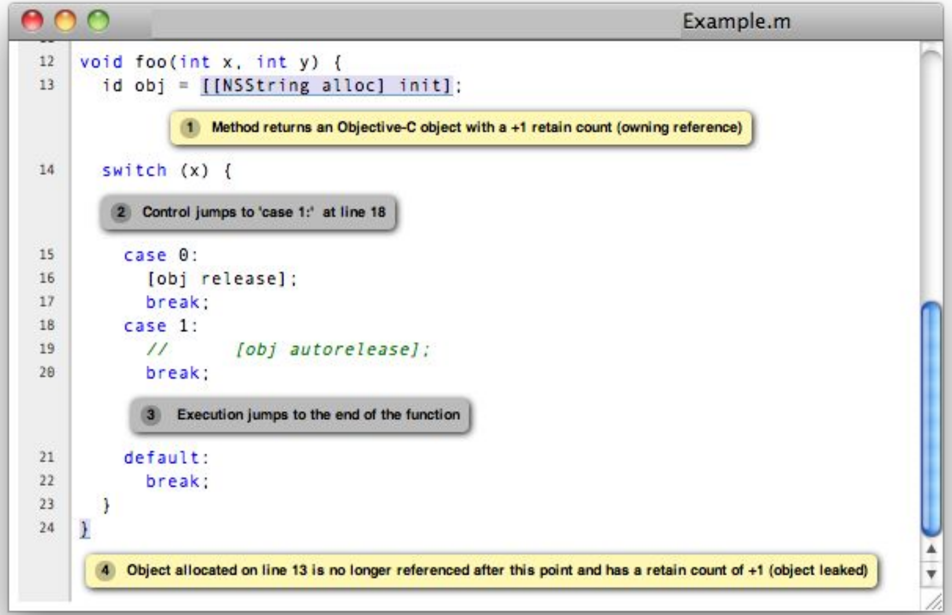
https://bugs.chromium.org/p/project-zero/issues/detail?id=1978

Clang static analyzer

Check for common security issues with a static analysis framework in the compiler

Built in checkers:

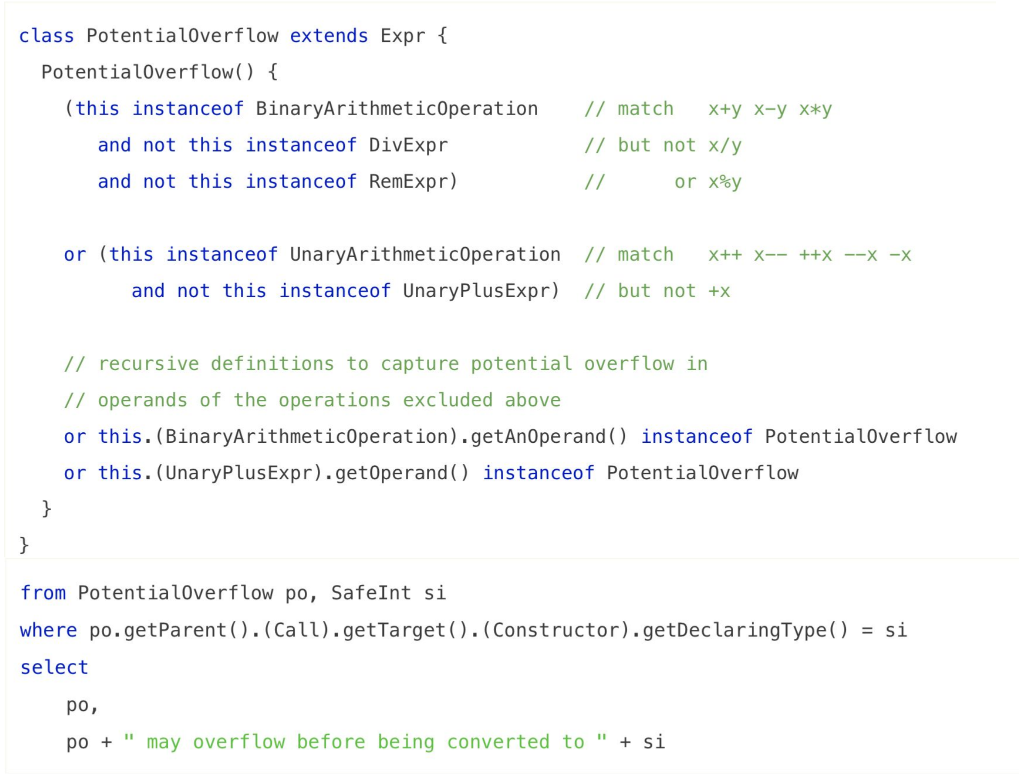
- Buffer overflows (with taint)
- Refcount errors
- **`malloc()`** integer overflows
- Insecure API use
- Uninitialized value use



```
void foo(int x, int y) {
  id obj = [[NSString alloc] init];
```
1 Method returns an Objective-C object with a +1 retain count (owning reference)
```
  switch (x) {
```
2 Control jumps to 'case 1:' at line 18
```
    case 0:
      [obj release];
      break;
    case 1:
      //      [obj autorelease];
      break;
```
3 Execution jumps to the end of the function
```
    default:
      break;
  }
}
```
4 Object allocated on line 13 is no longer referenced after this point and has a retain count of +1 (object leaked)

https://clang-analyzer.llvm.org/images/analyzer_html.png

CodeQL (Semmle)



```
class PotentialOverflow extends Expr {
  PotentialOverflow() {
    (this instanceof BinaryArithmeticOperation    // match   x+y x-y x*y
       and not this instanceof DivExpr            // but not x/y
       and not this instanceof RemExpr)           //      or x%y

    or (this instanceof UnaryArithmeticOperation  // match   x++ x-- ++x --x -x
          and not this instanceof UnaryPlusExpr)  // but not +x

    // recursive definitions to capture potential overflow in
    // operands of the operations excluded above
    or this.(BinaryArithmeticOperation).getAnOperand() instanceof PotentialOverflow
    or this.(UnaryPlusExpr).getOperand() instanceof PotentialOverflow
  }
}
```

```
from PotentialOverflow po, SafeInt si
where po.getParent().(Call).getTarget().(Constructor).getDeclaringType() = si
select
    po,
    po + " may overflow before being converted to " + si
```

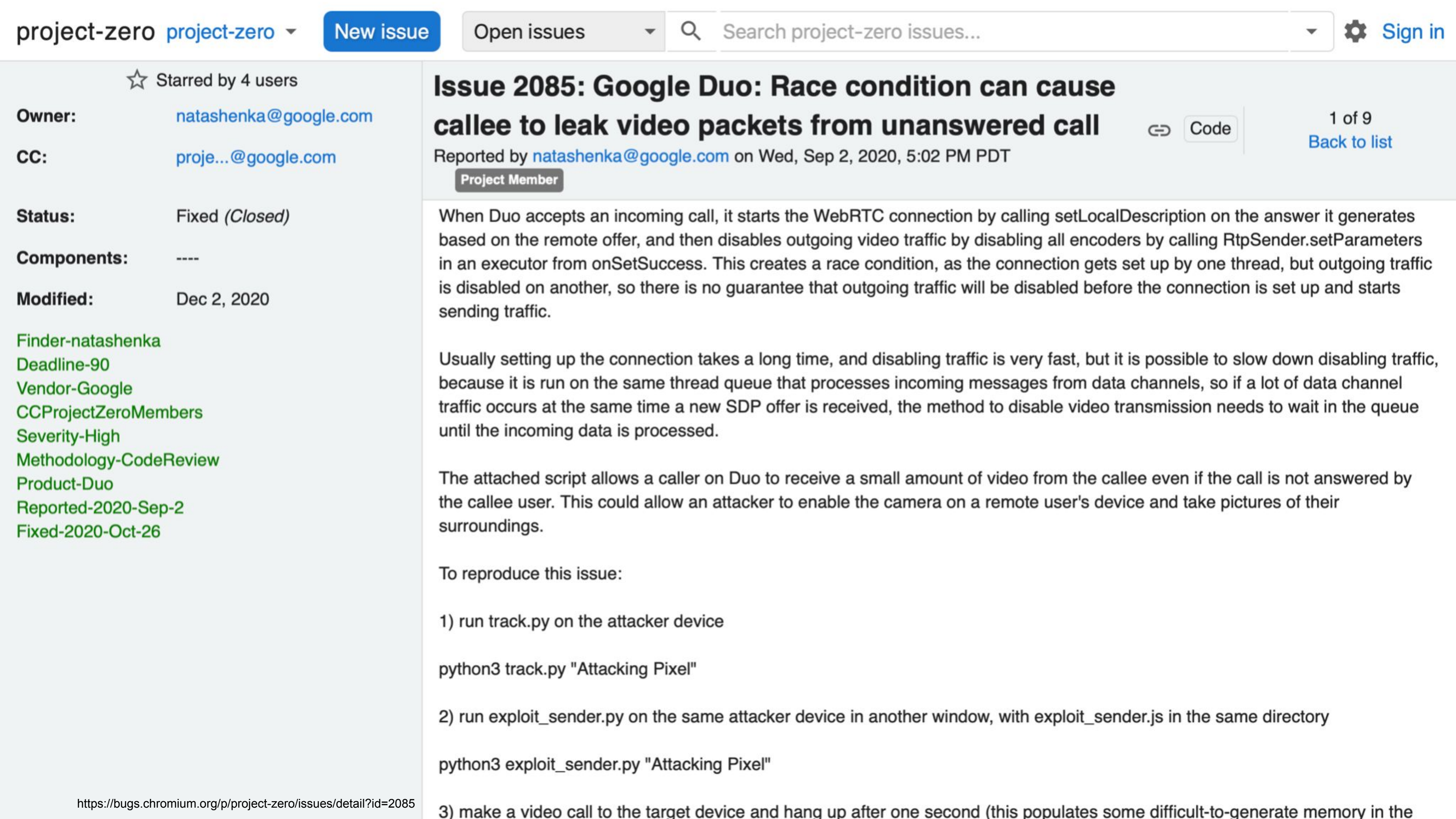
Query language for finding patterns in large codebases

"SQL for searching code"

Works best when you have a specific bad code pattern in mind

https://msrc-blog.microsoft.com/2018/08/16/vulnerability-hunting-with-semmle-ql-part-1/

Manual analysis



☆ Starred by 4 users

Owner:	natashenka@google.com
CC:	proje...@google.com
Status:	Fixed *(Closed)*
Components:	----
Modified:	Dec 2, 2020

Finder-natashenka
Deadline-90
Vendor-Google
CCProjectZeroMembers
Severity-High
Methodology-CodeReview
Product-Duo
Reported-2020-Sep-2
Fixed-2020-Oct-26

Issue 2085: Google Duo: Race condition can cause callee to leak video packets from unanswered call

Reported by natashenka@google.com on Wed, Sep 2, 2020, 5:02 PM PDT

Project Member

When Duo accepts an incoming call, it starts the WebRTC connection by calling setLocalDescription on the answer it generates based on the remote offer, and then disables outgoing video traffic by disabling all encoders by calling RtpSender.setParameters in an executor from onSetSuccess. This creates a race condition, as the connection gets set up by one thread, but outgoing traffic is disabled on another, so there is no guarantee that outgoing traffic will be disabled before the connection is set up and starts sending traffic.

Usually setting up the connection takes a long time, and disabling traffic is very fast, but it is possible to slow down disabling traffic, because it is run on the same thread queue that processes incoming messages from data channels, so if a lot of data channel traffic occurs at the same time a new SDP offer is received, the method to disable video transmission needs to wait in the queue until the incoming data is processed.

The attached script allows a caller on Duo to receive a small amount of video from the callee even if the call is not answered by the callee user. This could allow an attacker to enable the camera on a remote user's device and take pictures of their surroundings.

To reproduce this issue:

1) run track.py on the attacker device

```
python3 track.py "Attacking Pixel"
```

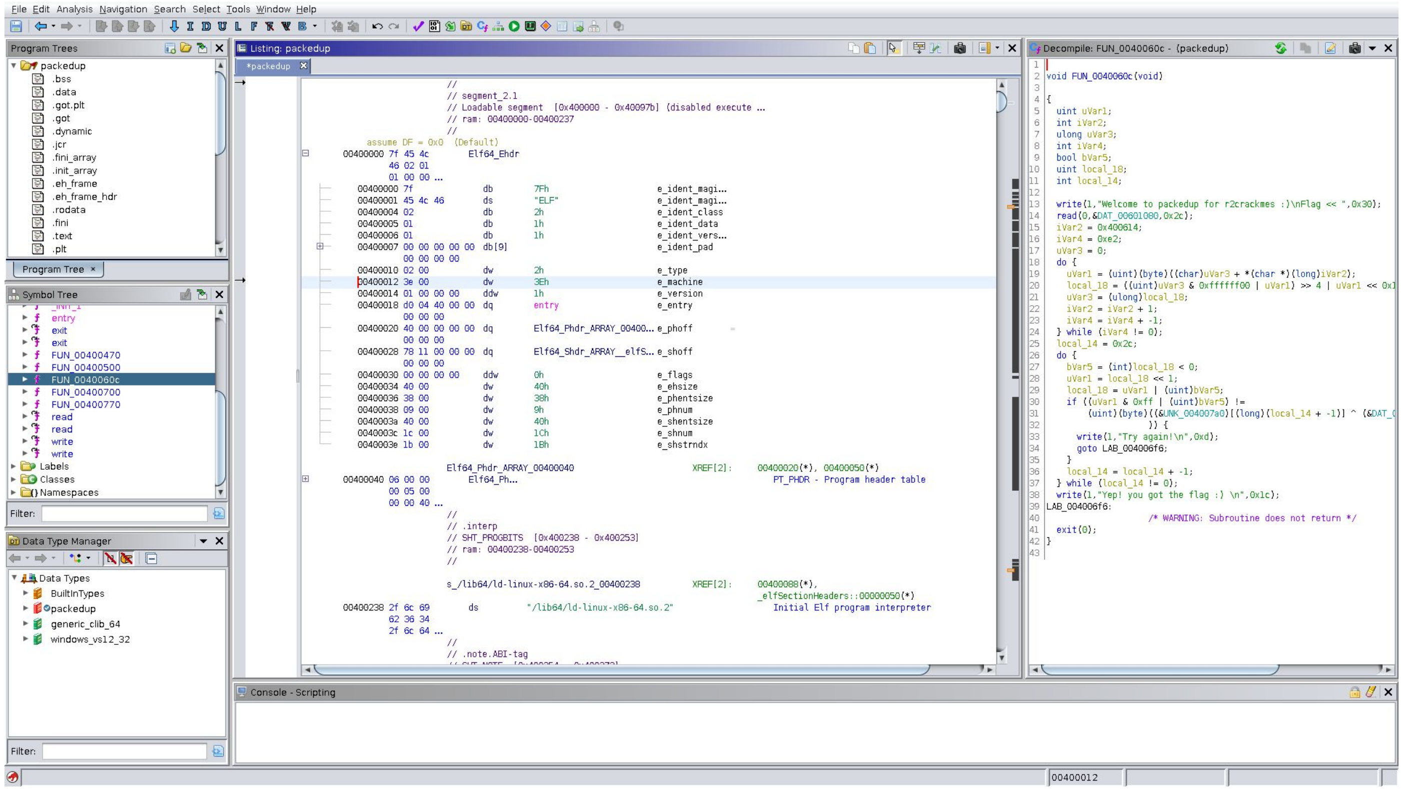
2) run exploit_sender.py on the same attacker device in another window, with exploit_sender.js in the same directory

```
python3 exploit_sender.py "Attacking Pixel"
```

3) make a video call to the target device and hang up after one second (this populates some difficult-to-generate memory in the

https://bugs.chromium.org/p/project-zero/issues/detail?id=2085

Reverse engineering



```
                       //
                       // segment_2.1
                       // Loadable segment  [0x400000 - 0x40097b] (disabled execute ...
                       // ram: 00400000-00400237
                       //
          assume DF = 0x0  (Default)
   00400000 7f 45 4c        Elf64_Ehdr
            46 02 01
            01 00 00 ...
      00400000 7f             db        7Fh                    e_ident_magi...
      00400001 45 4c 46       ds        "ELF"                  e_ident_magi...
      00400004 02             db        2h                     e_ident_class
      00400005 01             db        1h                     e_ident_data
      00400006 01             db        1h                     e_ident_vers...
      00400007 00 00 00 00 00 db[9]                            e_ident_pad
               00 00 00 00
      00400010 02 00          dw        2h                     e_type
      00400012 3e 00          dw        3Eh                    e_machine
      00400014 01 00 00 00    ddw       1h                     e_version
      00400018 d0 04 40 00 00 dq        entry                  e_entry
               00 00 00
      00400020 40 00 00 00 00 dq        Elf64_Phdr_ARRAY_00400... e_phoff
               00 00 00
      00400028 78 11 00 00 00 dq        Elf64_Shdr_ARRAY__elfS... e_shoff
               00 00 00
      00400030 00 00 00 00    ddw       0h                     e_flags
      00400034 40 00          dw        40h                    e_ehsize
      00400036 38 00          dw        38h                    e_phentsize
      00400038 09 00          dw        9h                     e_phnum
      0040003a 40 00          dw        40h                    e_shentsize
      0040003c 1c 00          dw        1Ch                    e_shnum
      0040003e 1b 00          dw        1Bh                    e_shstrndx

                       Elf64_Phdr_ARRAY_00400040          XREF[2]:     00400020(*), 00400050(*)
   00400040 06 00 00        Elf64_Ph...                                  PT_PHDR - Program header table
            00 05 00
            00 00 40 ...
                       //
                       // .interp
                       // SHT_PROGBITS  [0x400238 - 0x400253]
                       // ram: 00400238-00400253
                       //

                       s_/lib64/ld-linux-x86-64.so.2_00400238       XREF[2]:     00400088(*),
                                                                                 _elfSectionHeaders::00000050(*)
   00400238 2f 6c 69        ds        "/lib64/ld-linux-x86-64.so.2"               Initial Elf program interpreter
            62 36 34
            2f 6c 64 ...
                       //
                       // .note.ABI-tag
```

```c

void FUN_0040060c(void)

{
  uint uVar1;
  int iVar2;
  ulong uVar3;
  int iVar4;
  bool bVar5;
  uint local_18;
  int local_14;

  write(1,"Welcome to packedup for r2crackmes :)\nFlag << ",0x30);
  read(0,&DAT_00601080,0x2c);
  iVar2 = 0x400614;
  iVar4 = 0xe2;
  uVar3 = 0;
  do {
    uVar1 = (uint)(byte)((char)uVar3 + *(char *)(long)iVar2);
    local_18 = ((uint)uVar3 & 0xffffff00 | uVar1) >> 4 | uVar1 << 0x1
    uVar3 = (ulong)local_18;
    iVar2 = iVar2 + 1;
    iVar4 = iVar4 + -1;
  } while (iVar4 != 0);
  local_14 = 0x2c;
  do {
    bVar5 = (int)local_18 < 0;
    uVar1 = local_18 << 1;
    local_18 = uVar1 | (uint)bVar5;
    if ((uVar1 & 0xff | (uint)bVar5) !=
        (uint)(byte)((&UNK_004007a0)[(long)(local_14 + -1)] ^ (&DAT_0
                    )) {
      write(1,"Try again!\n",0xd);
      goto LAB_004006f6;
    }
    local_14 = local_14 + -1;
  } while (local_14 != 0);
  write(1,"Yep! you got the flag :) \n",0x1c);
LAB_004006f6:
                    /* WARNING: Subroutine does not return */
  exit(0);
}

```

Decompile a program to see how it works

Closed source programs often have shallower bugs

Tips for writing (more) secure software

Software tests

One of the most effective ways to reduce bugs

Unit tests: Check that each piece of code behaves as expected in isolation

Goal: Unit tests should cover all code, including error handling

So many exploitable bugs would be eliminated with basic unit tests

Regression tests: Check that old bugs haven’t been reintroduced

If you don’t run regression tests, attackers will run them for you!

Integration tests: Check that modules work together as expected

General tips

1. Do not use a memory-unsafe language for new codebases

   Starting Rust, Swift, etc. is a one-time cost; fixing C++ memory stompers is a cost you'll pay over and over again forever

2. Integrate security experts very early in the design process

   Better to learn about fundamental flaws early to avoid re-doing everything

3. Design APIs so that the easiest way to use them is the safe way

   Engineers using a new API tend to take the path of least resistance

Thank you!

bazad@cs.stanford.edu