

Finding vulnerabilities

by fuzzing, dynamic and static analysis

Brandon Azad
Stanford CS155 guest lecture
April 13, 2022

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

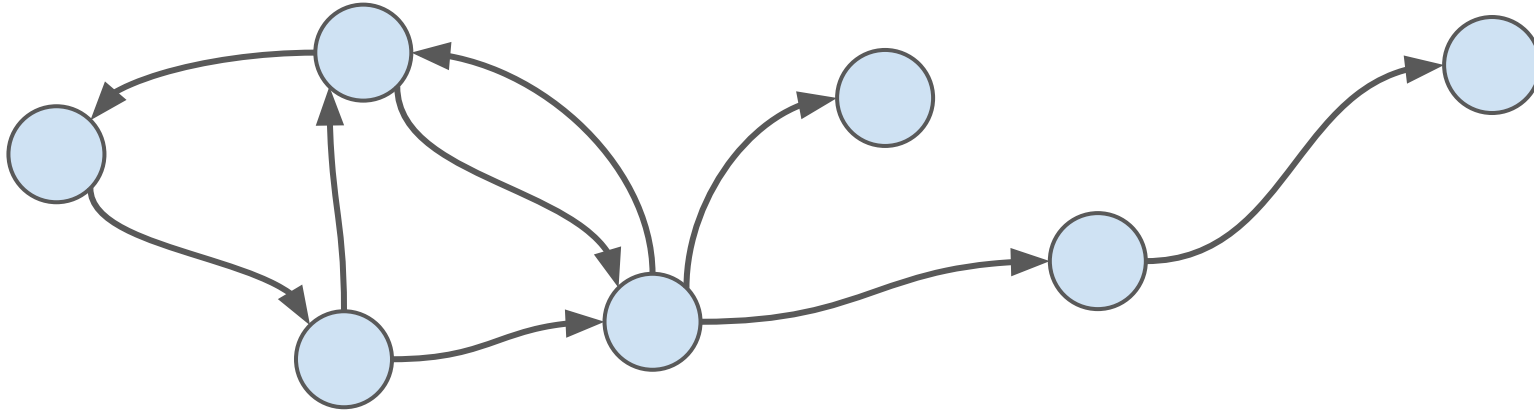
Mission: Make 0-day hard

2020: Trying something new

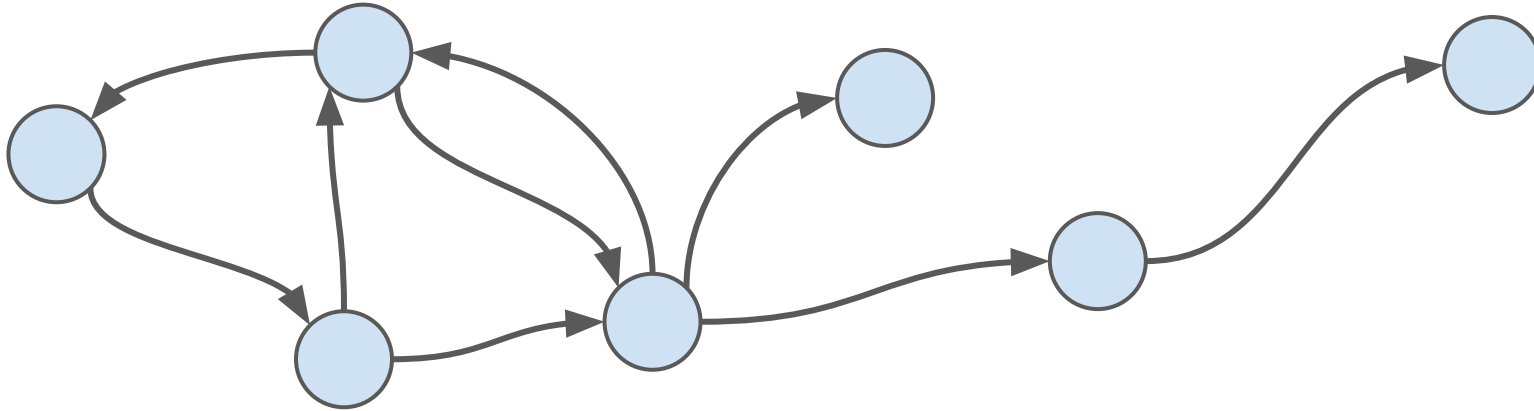


Conceptualizing vulnerabilities and exploits

Computer programs: finite state machines

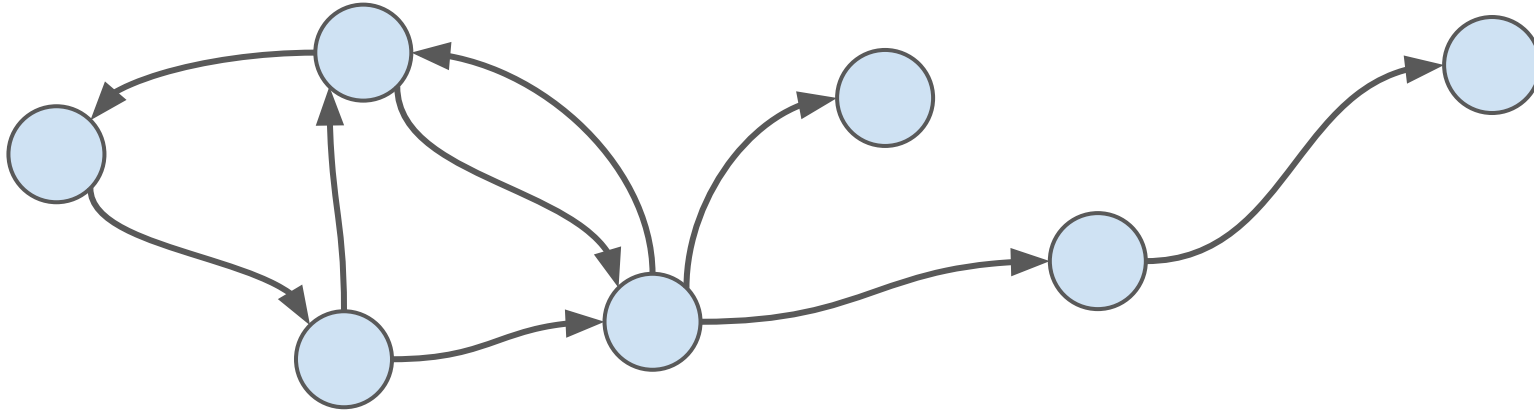


Computer programs: finite state machines



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

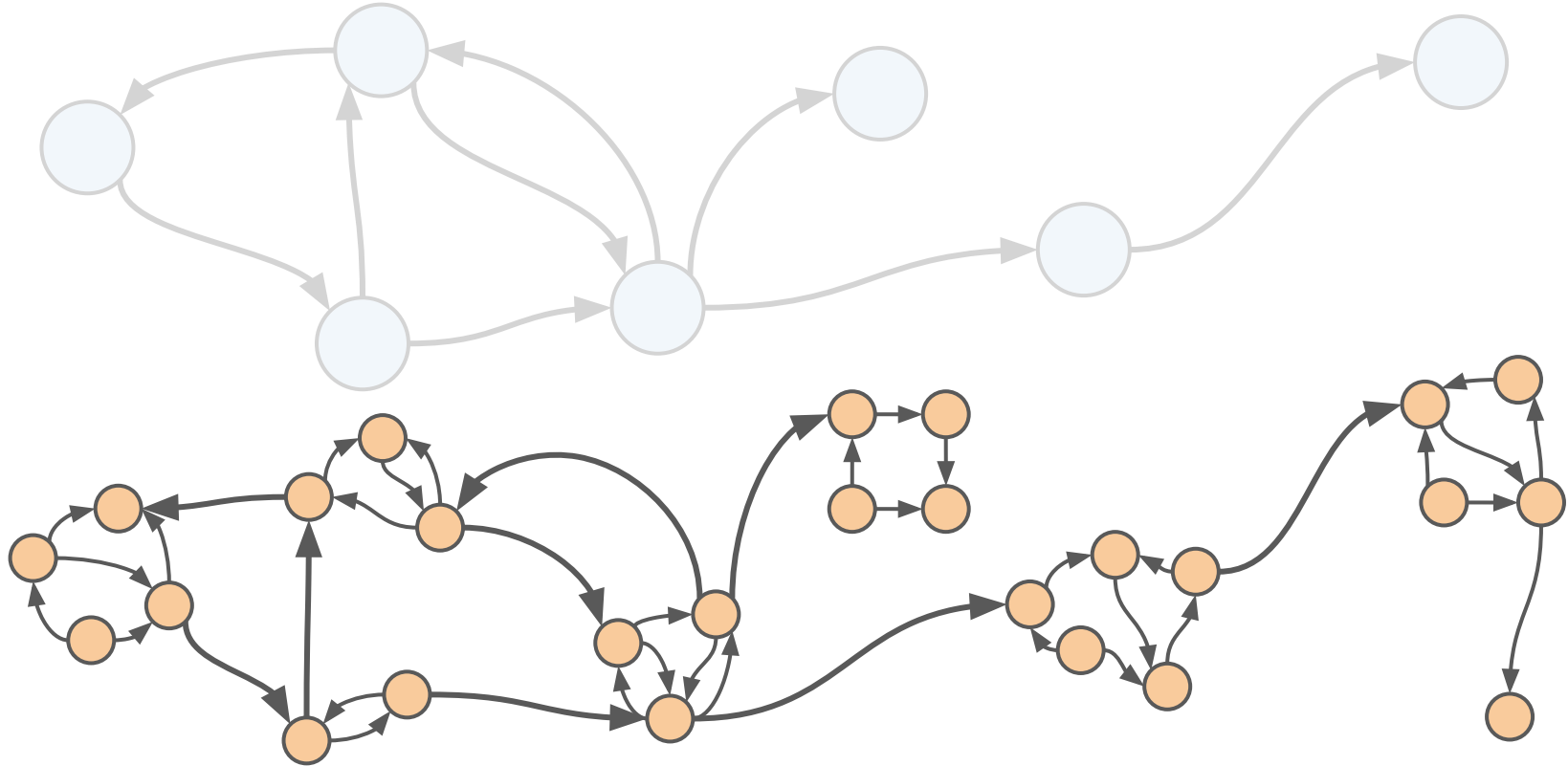
Computer programs: finite state machines



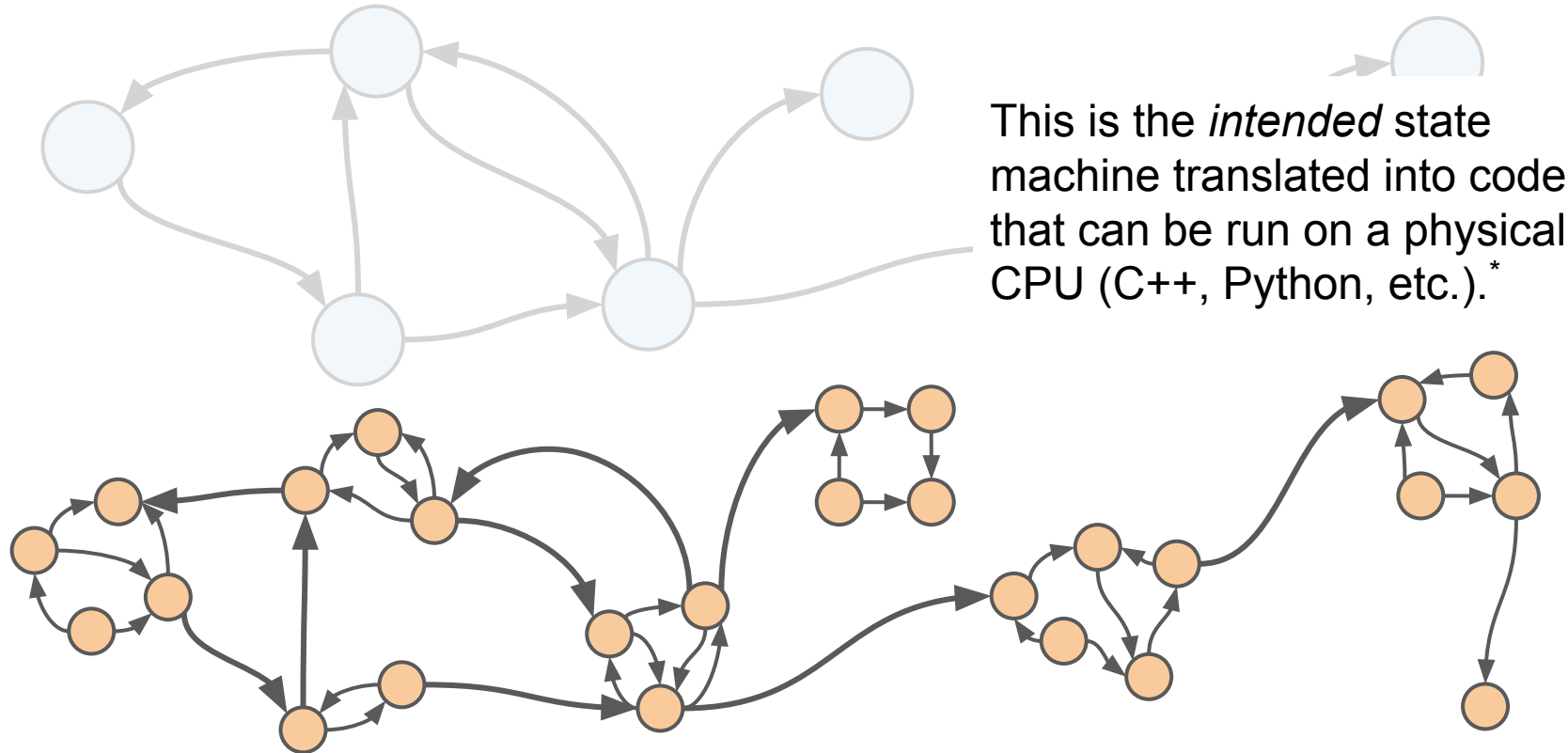
This is a *conceptual* state machine describing the *intended* operation of the program.

A physical CPU cannot directly execute this abstract state machine.

Running code: state machines emulating state machines



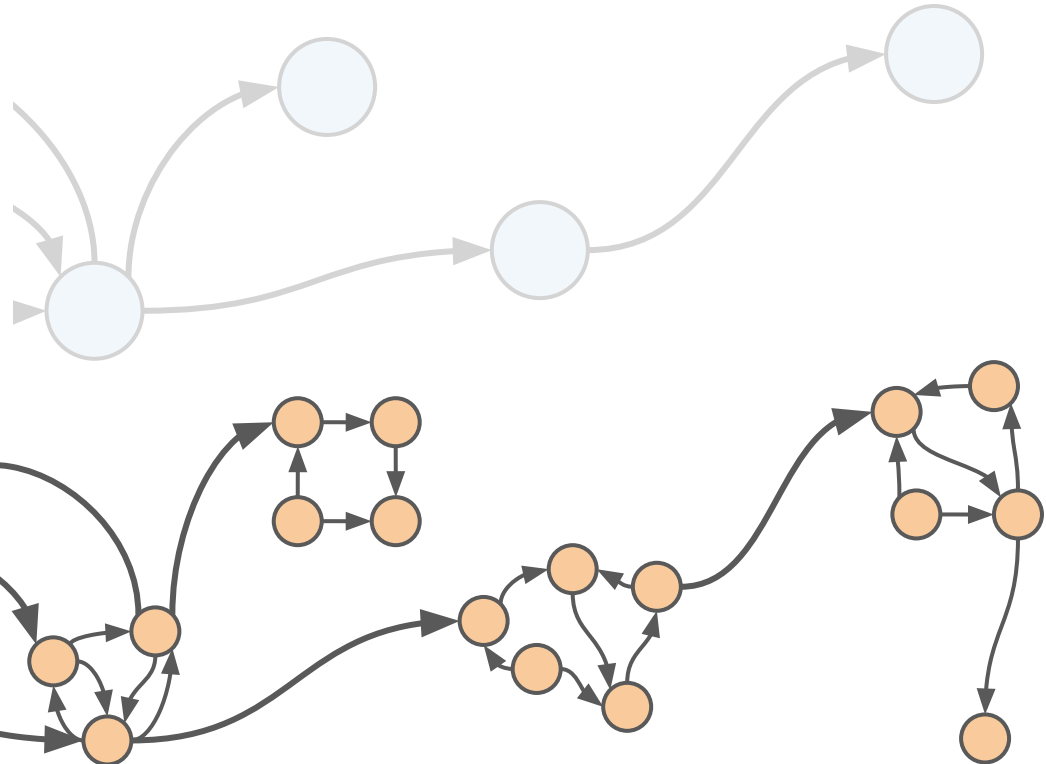
Running code: state machines emulating state machines



* 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

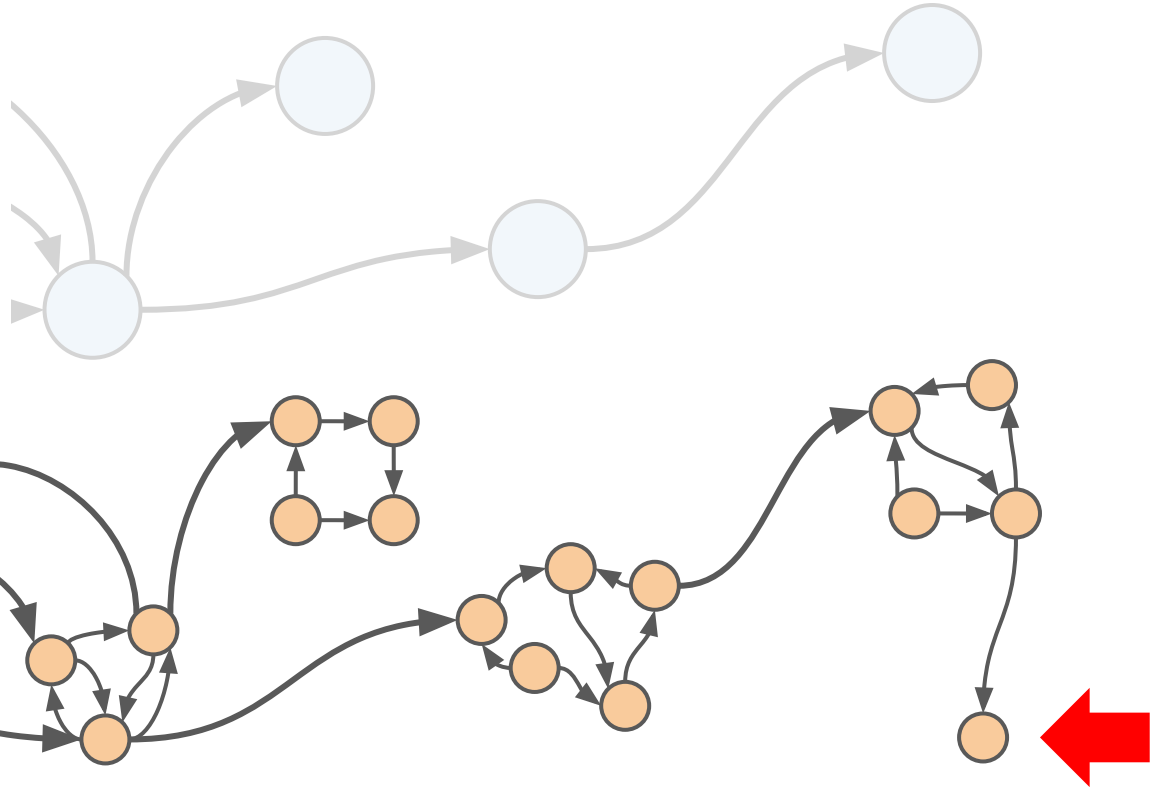
Bugs occur when there are reachable states in the runnable state machine (the code) that have no corresponding state in the intended state machine (the design).^{*}



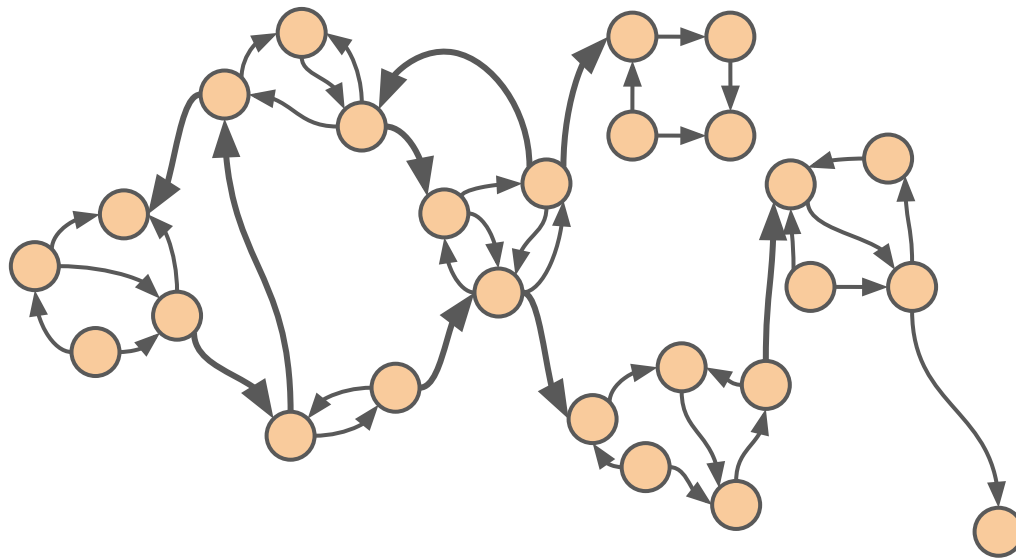
^{*} Not the full picture: the initial design itself could have issues (design issues) which still count as software bugs.

Running code: state machines emulating state machines

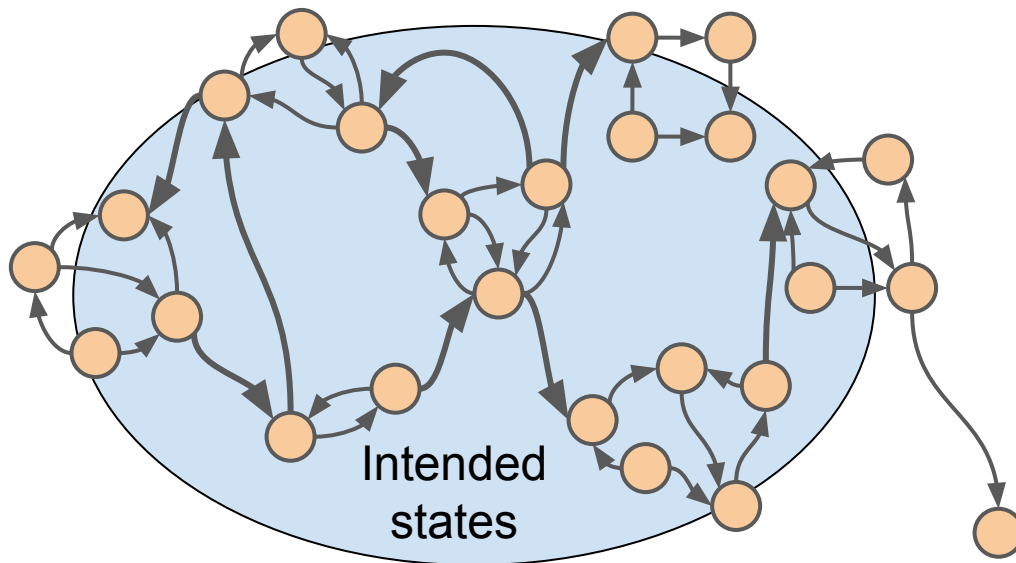
Bugs occur when there are reachable states in the runnable state machine (the code) that have no corresponding state in the intended state machine (the design).^{*}



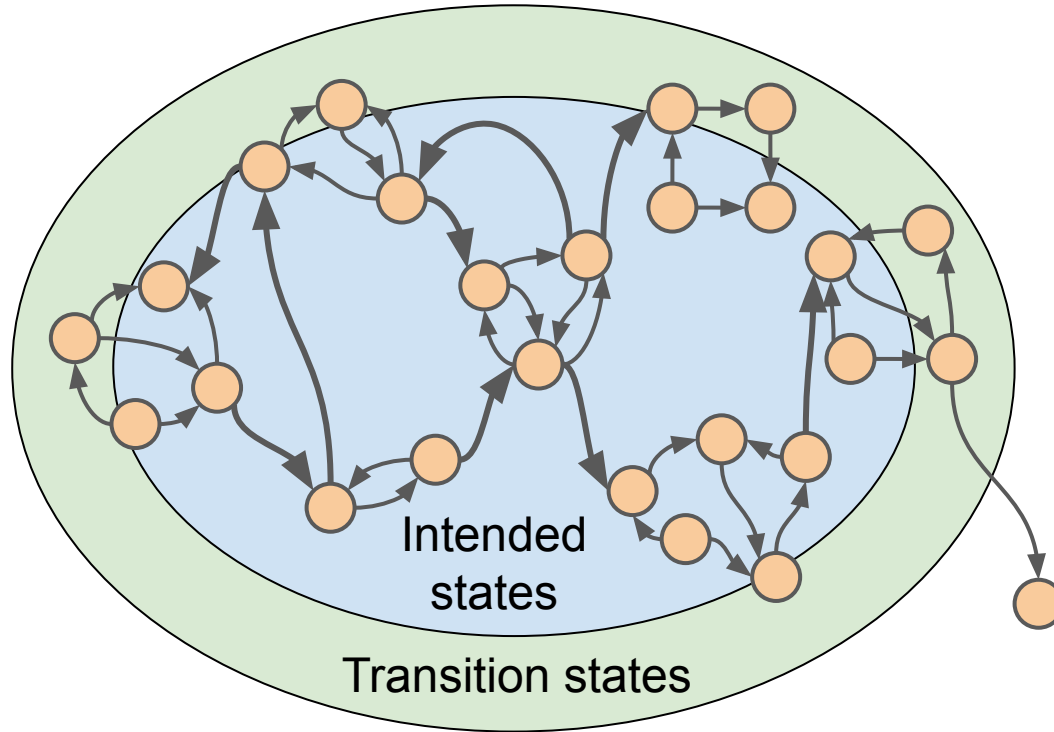
Classifying states



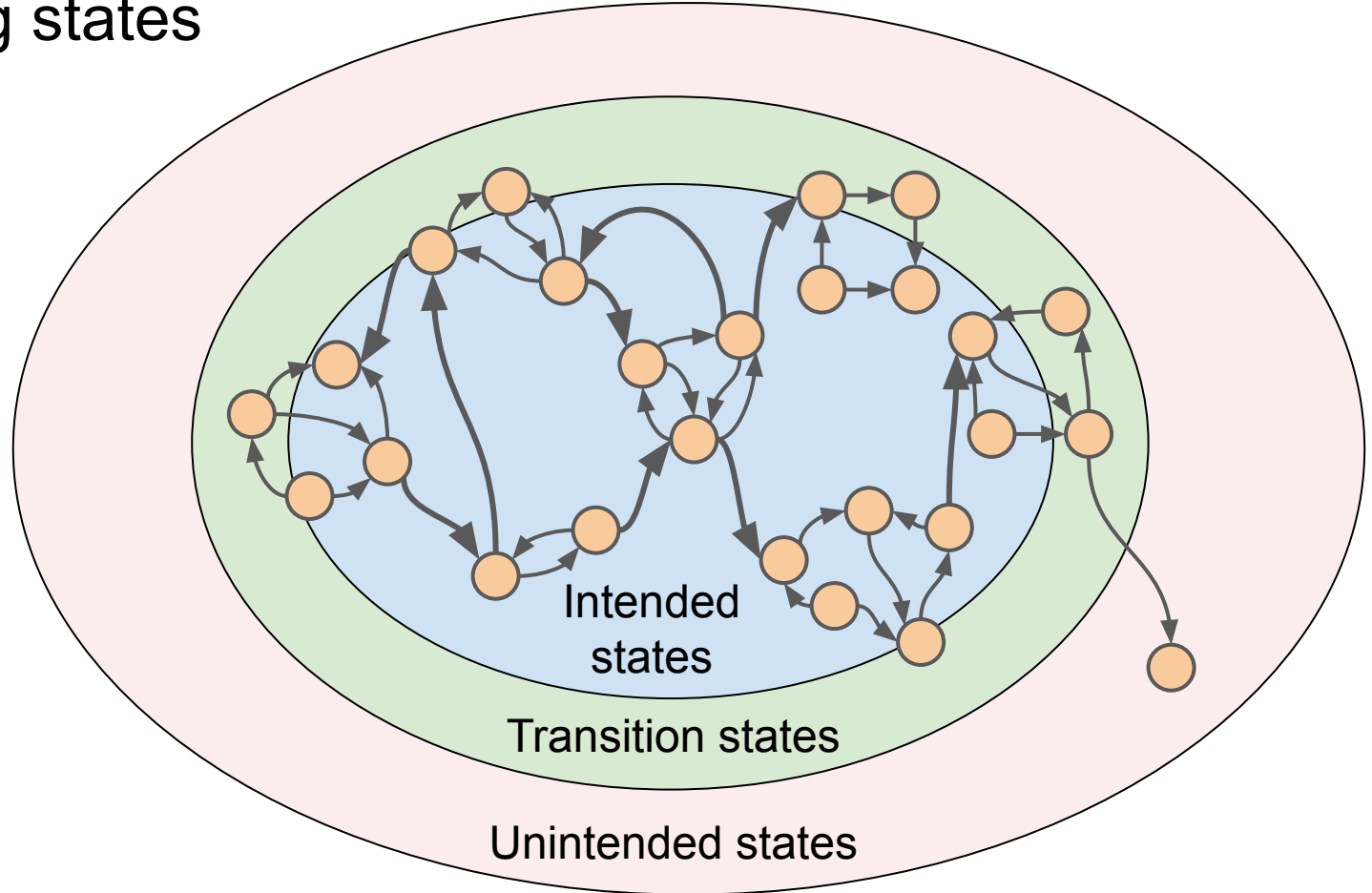
Classifying states



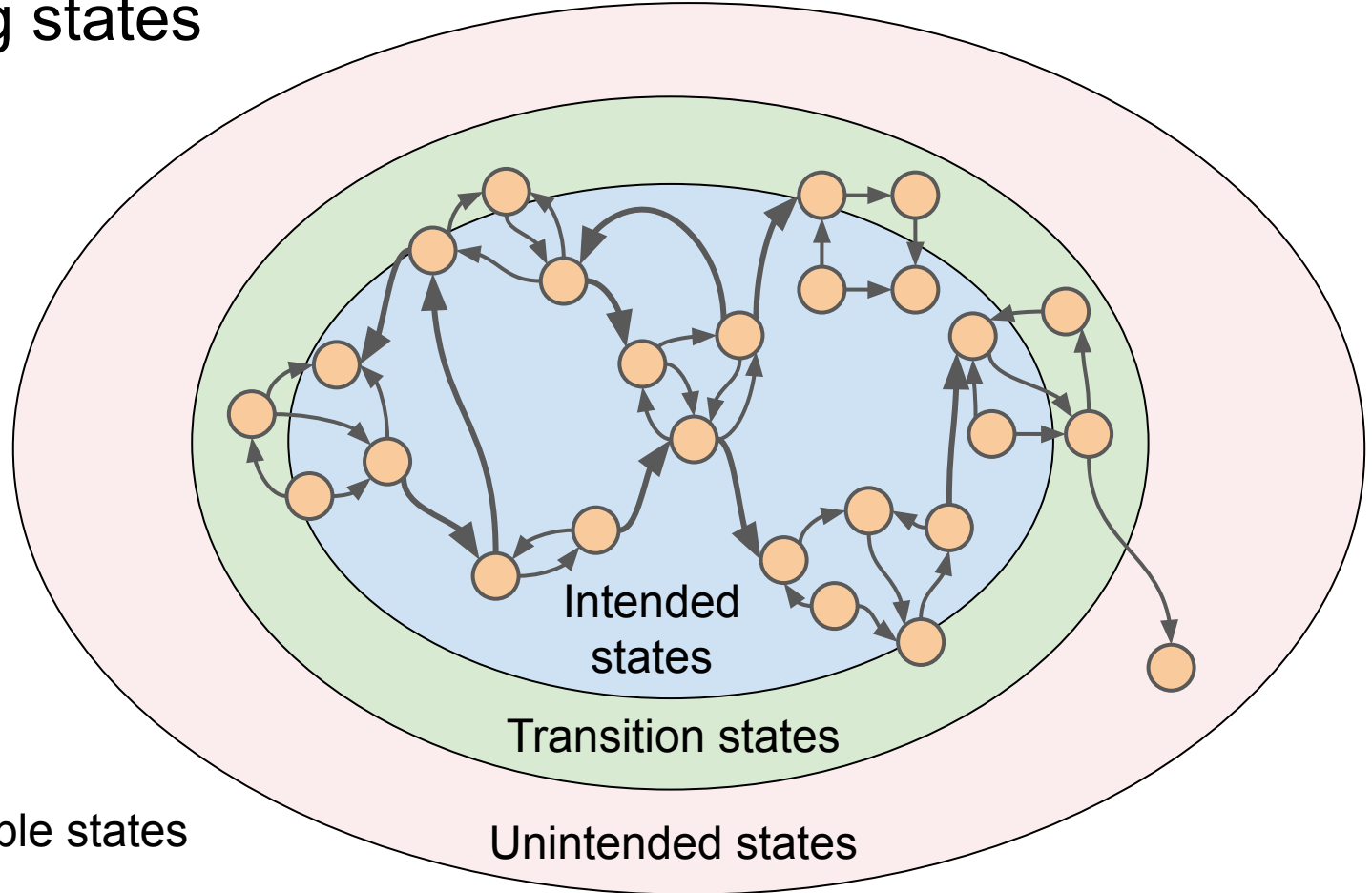
Classifying states



Classifying states

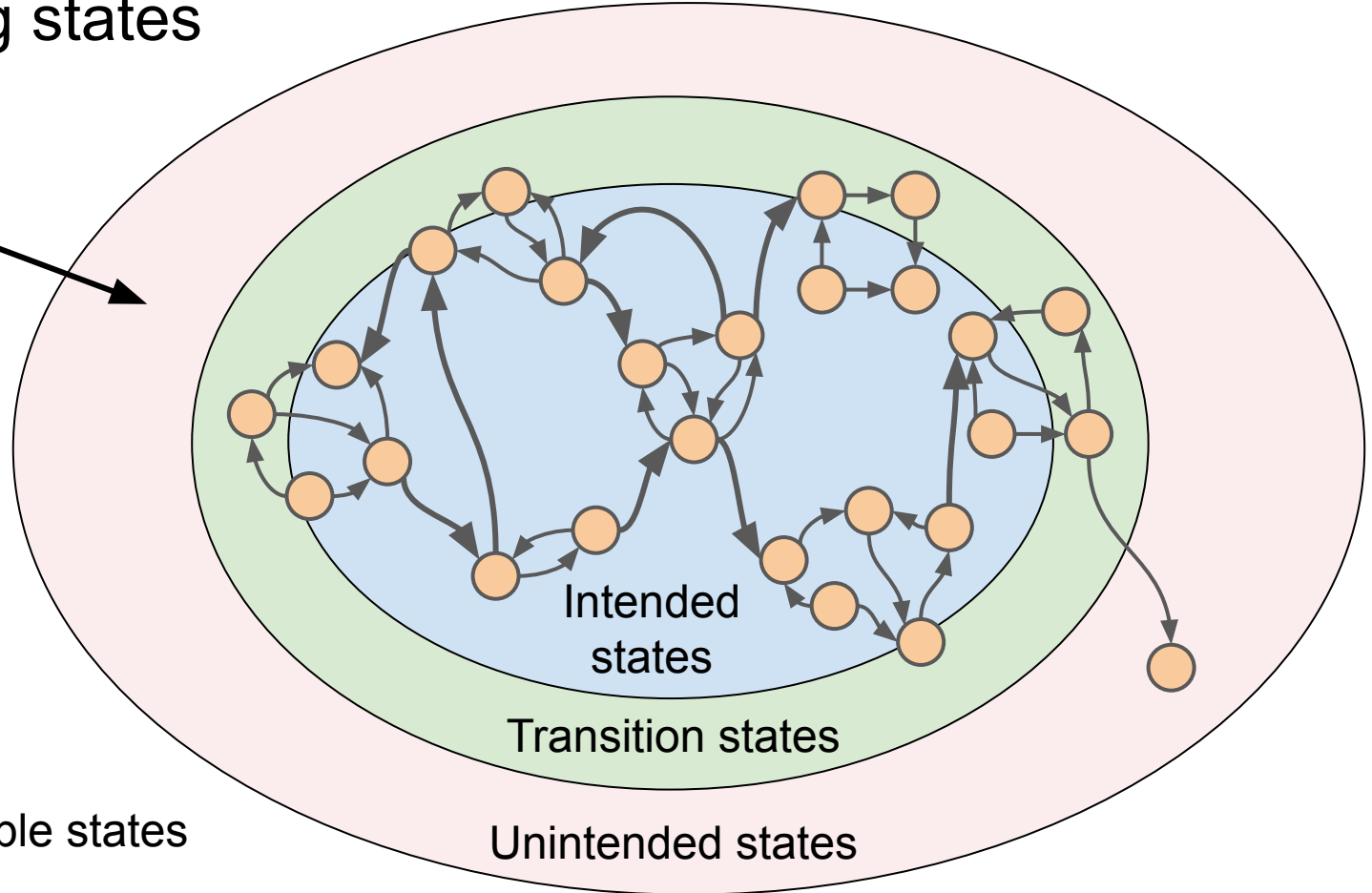
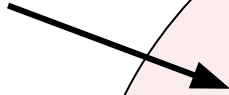


Classifying states



Classifying states

Vulnerabilities
live here



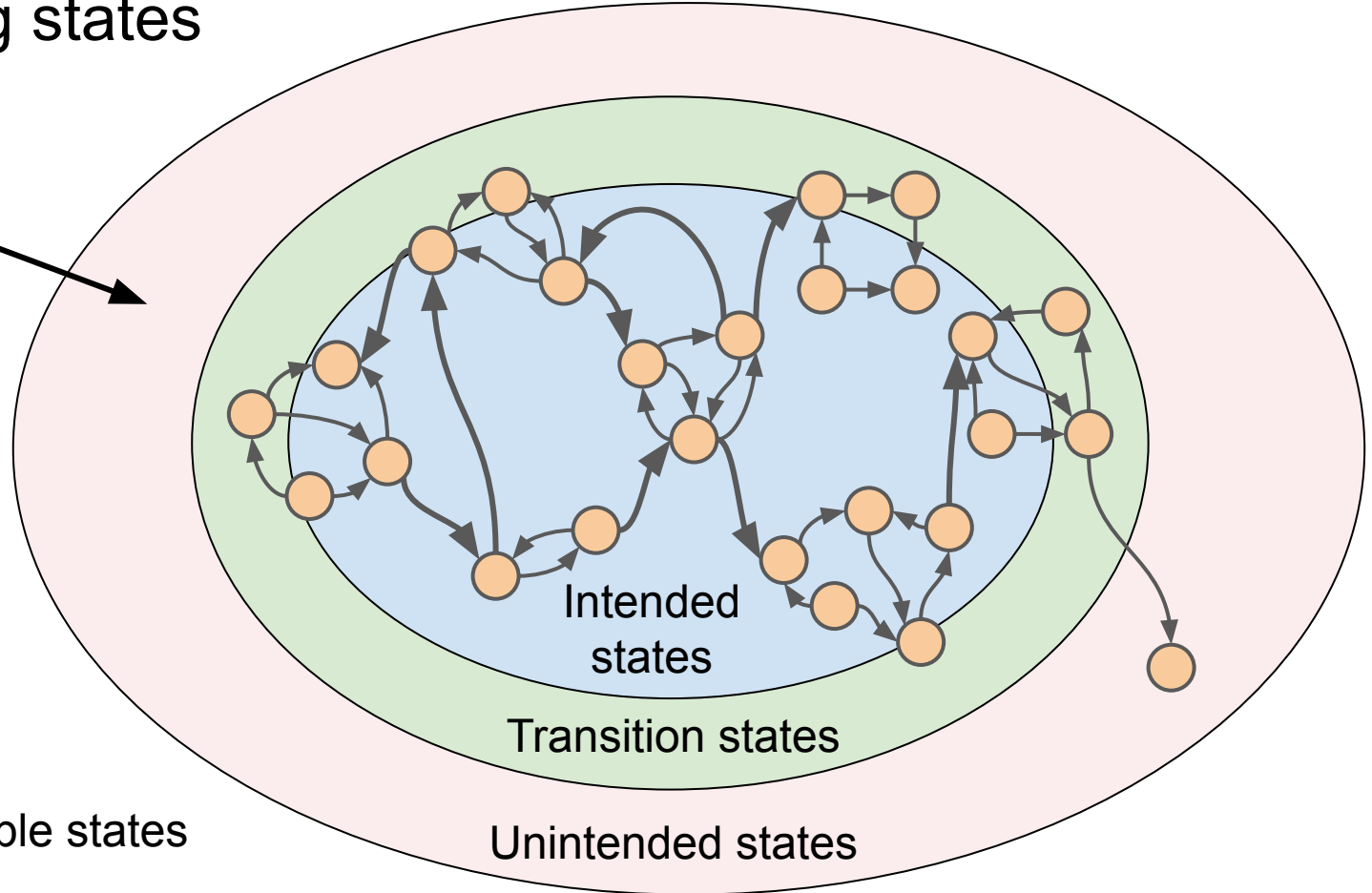
Unreachable states

Unintended states

Classifying states

Vulnerabilities
live here

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



Unreachable states

Unintended states

Classifying states

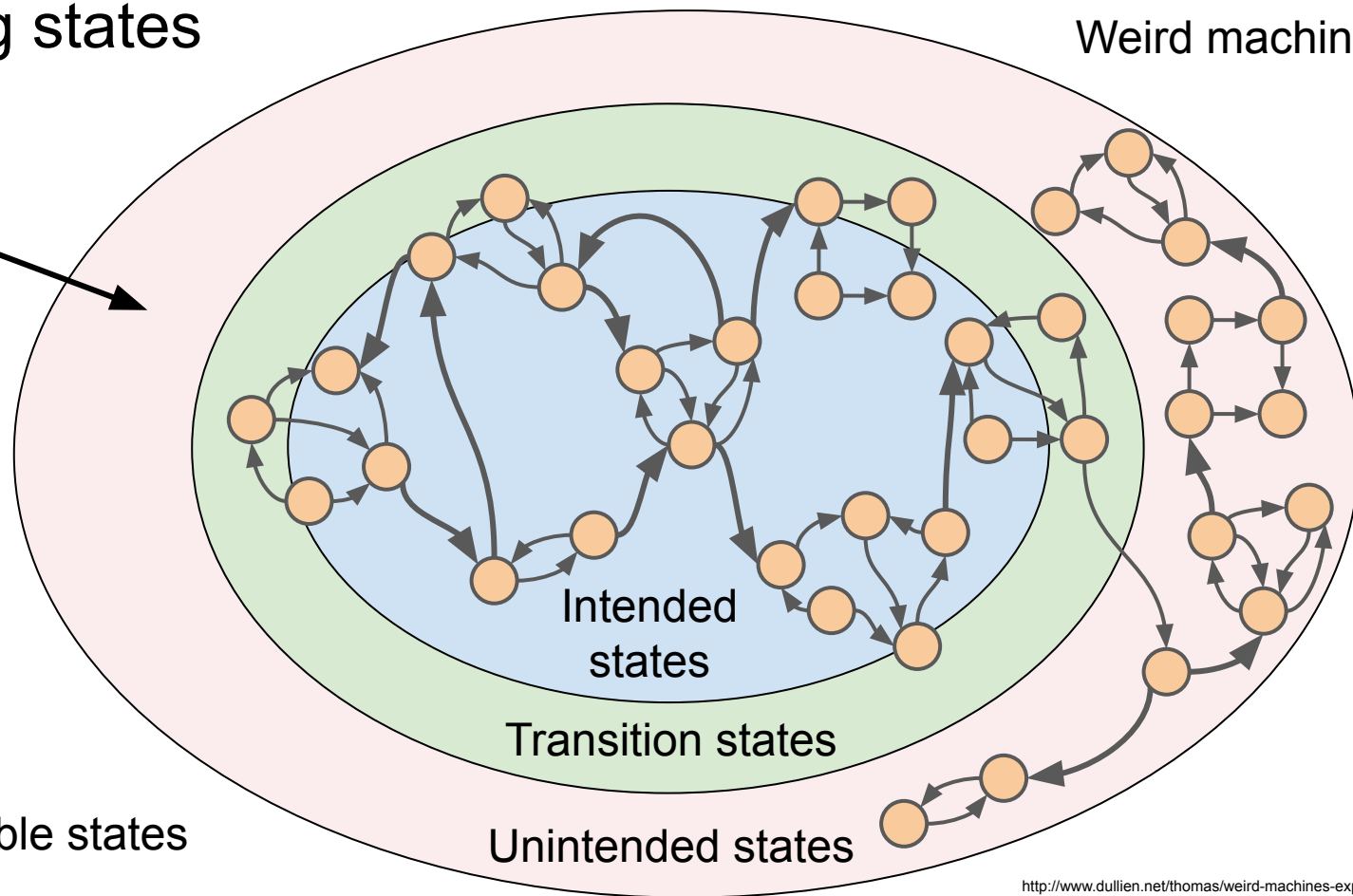
Vulnerabilities
live here

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

Weird machines

Unreachable states

Unintended states



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

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

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

- Measure 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

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 be successful?

In 2010, Charlie Miller fuzzed PDF viewers using the following mutation program:

```
numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
for j in range(numwrites):
    rbyte = random.randrange(256)
    rn = random.randrange(len(buf))
    buf[rn] = "%c"%(rbyte)
```

Found 64 exploitable-looking crashes

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

Mutation-based fuzzing

Advantages

- Simple to set up and run

- Can use off-the-shelf software (possibly with a harness) for many programs

Limitations

- Results depend strongly on the quality of the initial corpus

- Coverage may 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

Raw Blame

304 lines (235 sloc) 15.7 KB

Syscall descriptions

`syszaller` 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]) fd
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_IROTH, S
```

The descriptions are contained in `sys/$05/*.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)
```

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

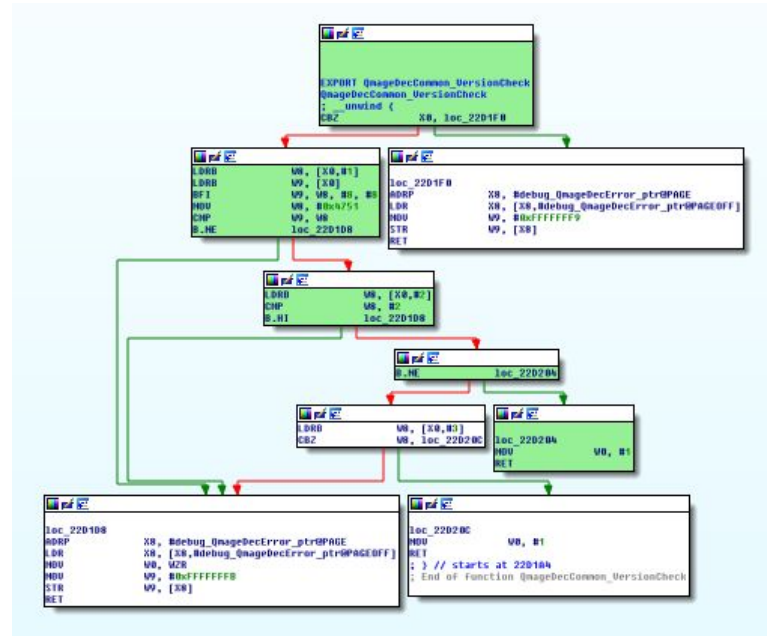
Key insight: code coverage is a useful metric, why not use it as **feedback** to guide fuzzing?

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?



american fuzzy lop (AFL)

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

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	

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 strong 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

Few examples of Qmage files

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

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

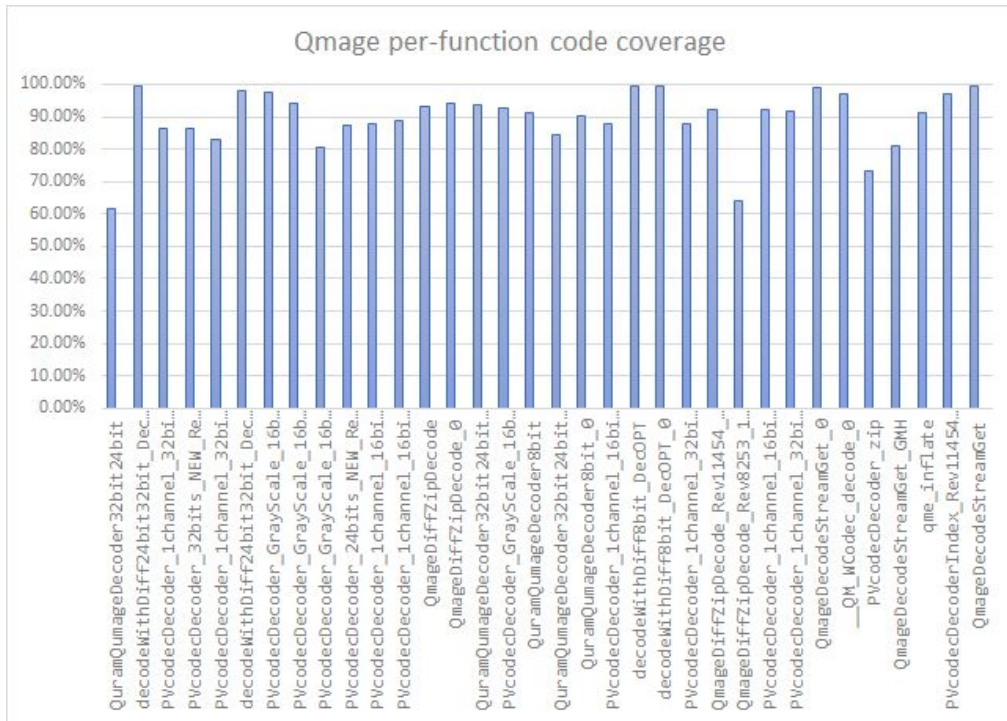
An emulator (qemu-aarch64) was used to run the harness and Qmage library on a Linux machine

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

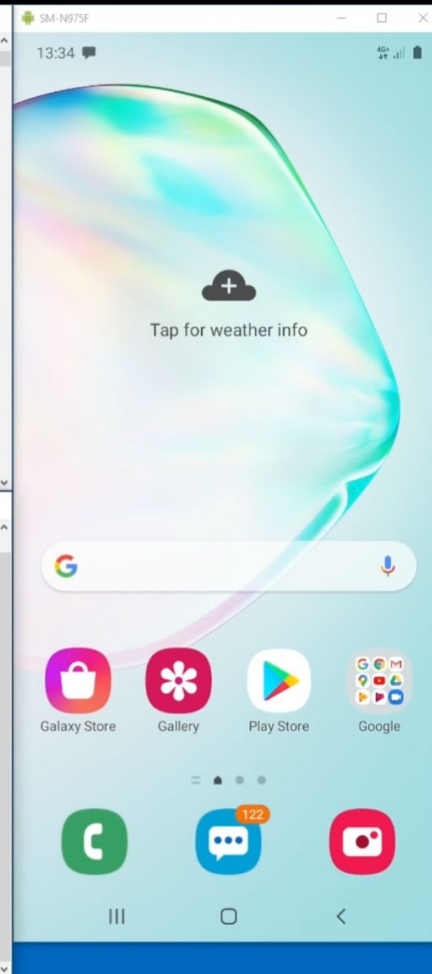
```
C:\Windows\System32\cmd.exe
2020-04-22 13:21:08,765 [INFO] Range [765a760000 .. 765a760fff] is readable: true
2020-04-22 13:22:25,896 [INFO] Range [765a760000 .. 765a7e7fff] is readable: true
2020-04-22 13:24:03,055 [INFO] Range [765a760000 .. 765a86ffff] is readable: false
2020-04-22 13:25:10,218 [INFO] Range [765a7a8000 .. 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: ~

```
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
:/ $ █
```

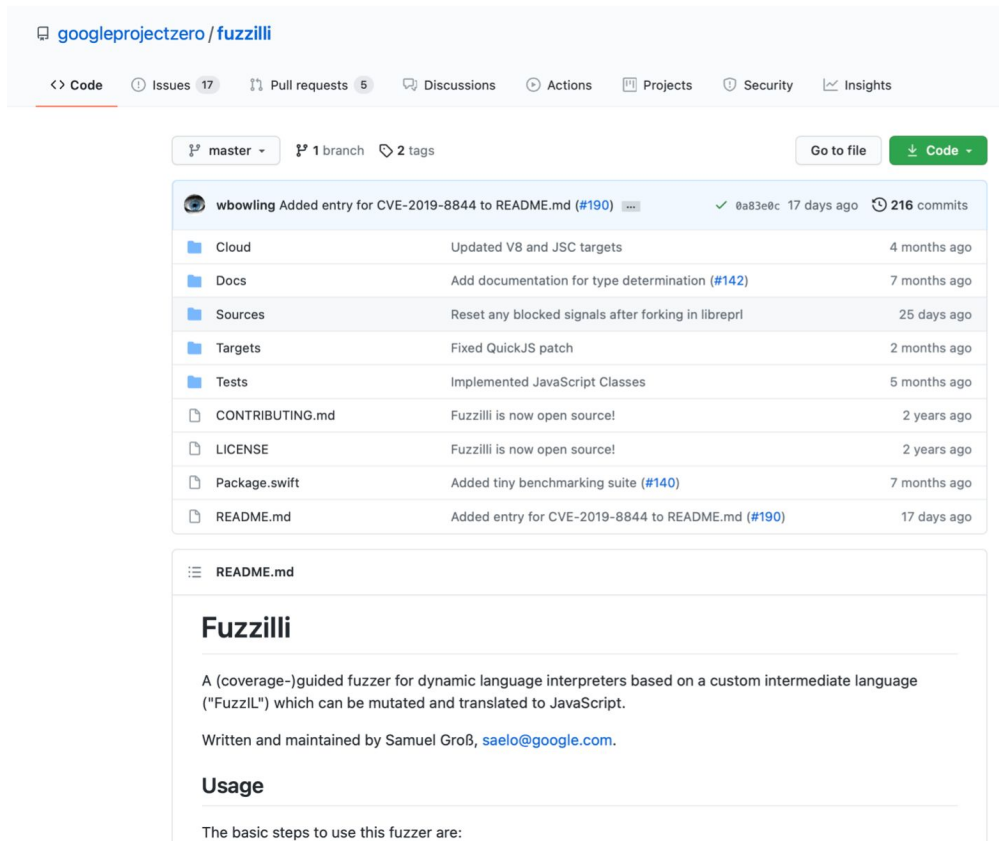
<https://www.youtube.com/watch?v=nke8Z3G4jnc>



Another cool fuzzer: Fuzzilli

Very successful JavaScript fuzzer

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



The screenshot shows the GitHub repository for `googleprojectzero/fuzzilli`. At the top, there are navigation links for Code, Issues (17), Pull requests (5), Discussions, Actions, Projects, Security, and Insights. Below these, the repository is shown with the `master` branch selected, 1 branch, and 2 tags. A commit history table lists recent changes, including updates to targets, documentation, and the README. The README section is partially visible, showing the title "Fuzzilli" and a description of the fuzzer.

File	Commit Message	Time Ago
Cloud	Updated V8 and JSC targets	4 months ago
Docs	Add documentation for type determination (#142)	7 months ago
Sources	Reset any blocked signals after forking in librepl	25 days ago
Targets	Fixed QuickJS patch	2 months ago
Tests	Implemented JavaScript Classes	5 months ago
CONTRIBUTING.md	Fuzzilli is now open source!	2 years ago
LICENSE	Fuzzilli is now open source!	2 years ago
Package.swift	Added tiny benchmarking suite (#140)	7 months ago
README.md	Added entry for CVE-2019-8844 to README.md (#190)	17 days ago

Fuzzilli

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

Written and maintained by Samuel Groß, saelo@google.com.

Usage

The basic steps to use this fuzzer are:

Fuzzing summary

Off-the-shelf fuzzers are excellent at finding bugs

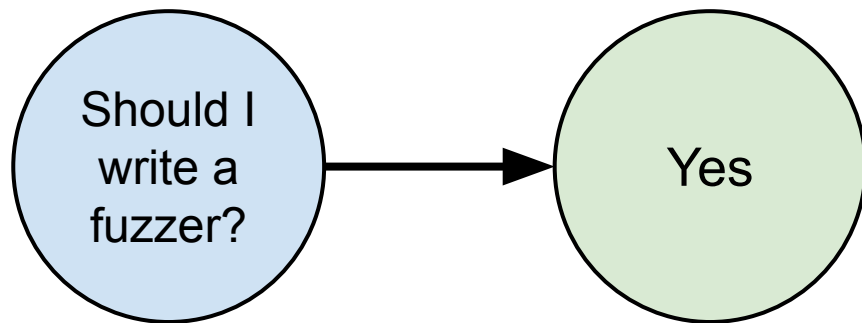
Custom fuzzers are also excellent at finding bugs

Different fuzzers often find different bugs

Relatively 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

May be 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.

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

Use-after-return

Use-after-scope

Double-free, invalid free

Memory leaks

Typically 2x slowdown

```
==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 0x7f9e9f25e76c in __libc_start_main /build/builddd/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 0x7f9e9f25e76c in __libc_start_main /build/builddd/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 0x7f9e9f25e76c in __libc_start_main /build/builddd/eglibc-2.15/csu/libc-start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

AddressSanitizer (ASan)

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

Out-of-bound

Use-after-free

Use-after-re

Use-after-s

Double-free, invalid free

Memory leaks

Typically 2x slowdown

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

```
#1 0x45914a in main use-after-free.c:4
#2 0x7fce9f25e76c in __libc_start_main /build/builddd/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/builddd/eglibc-2.15/csu/libc-start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

ThreadSanitizer (TSan)

Data race detector for C/C++

Similar in principle to AddressSanitizer but for race conditions

High overhead

5-10x memory

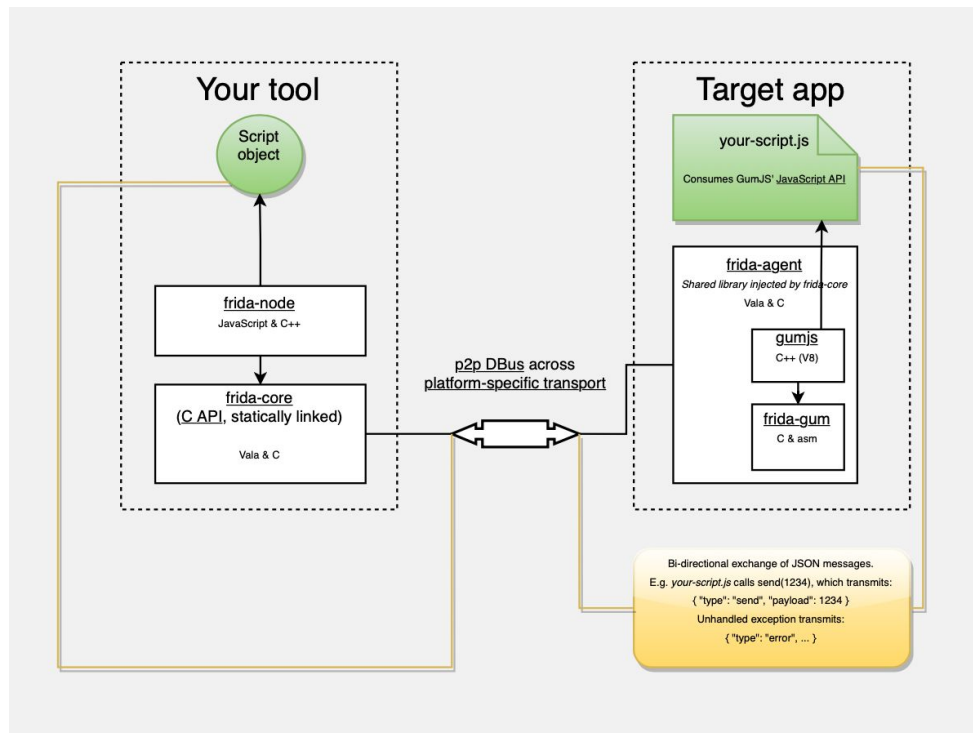
5-15x slowdown

```
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)
```

Frida



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

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

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!

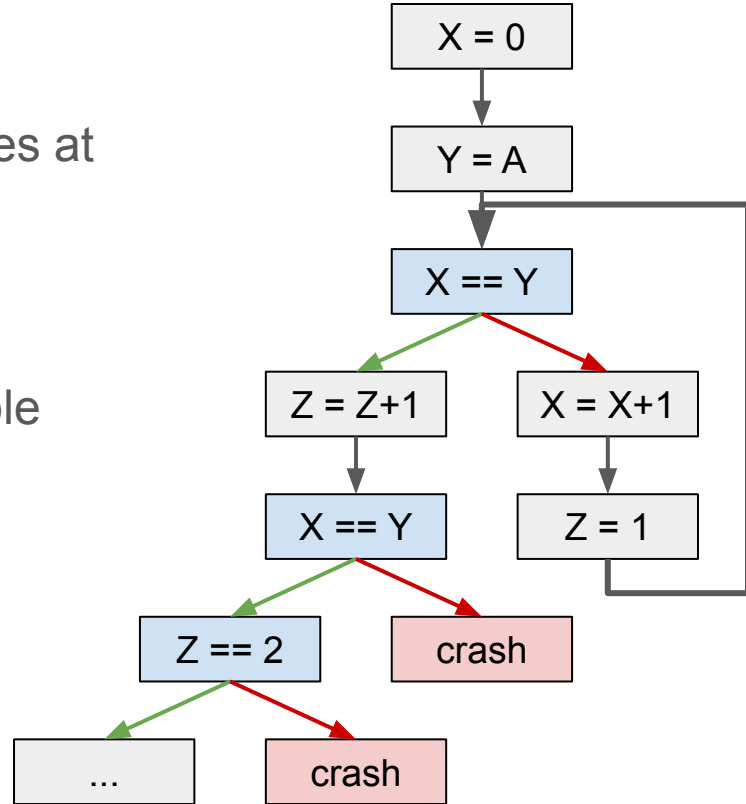
Most static analyzers are neither sound nor complete

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

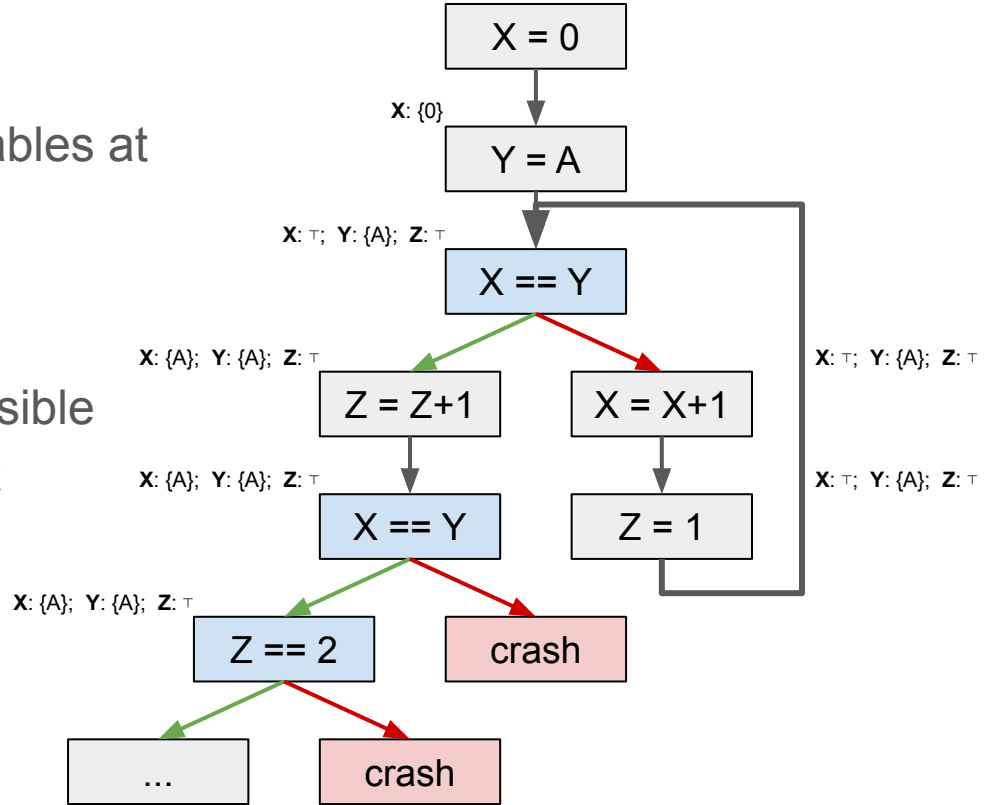


Data flow analysis

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

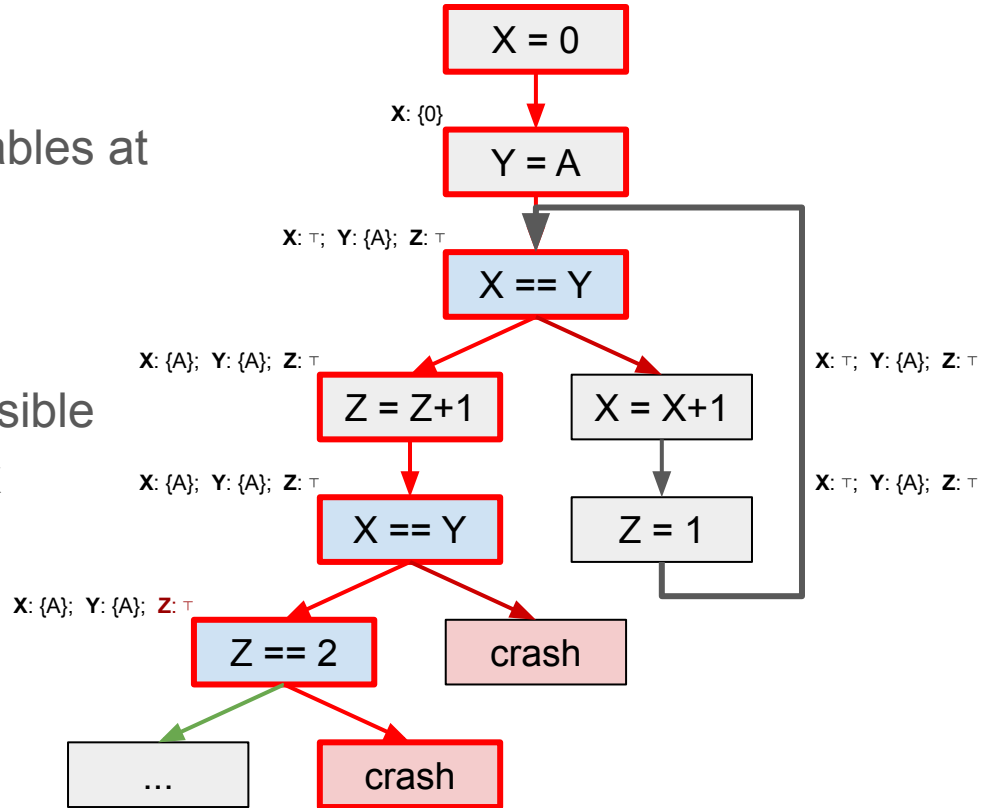


Data flow analysis

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()`

`free()`

`bzero()`


```

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()`

`free()`

`bzero()`

```

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()`

`free()`

`bzero()`

Clang static analyzer

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

Built in checkers:

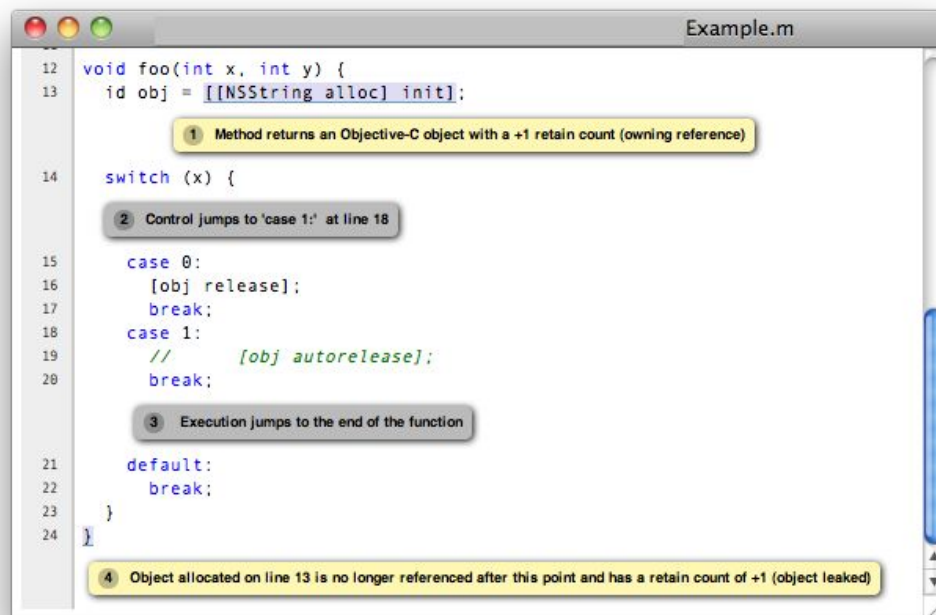
- Buffer overflows (with taint)

- RefCount errors

- `malloc()` integer overflows

- Insecure API use

- Uninitialized value use



Article

The state of static analysis in the GCC 12 compiler

April 12, 2022      [Compilers , C, C++ , Linux](#)



David Malcolm

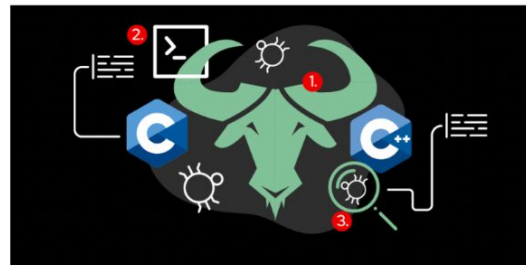


Table of contents:

Building a static analyzer into the [C compiler](#) offers several advantages over having a separate tool, because the analyzer can track what the compiler and assembler are doing intimately. As a Red Hat employee, I work on [GCC, the GNU Compiler Collection](#). Our static analyzer is still experimental but is making big strides in interesting areas, including a *taint mode* and an understanding of assembly-language code.



<https://developers.redhat.com/articles/2022/04/12/state-static-analysis-gcc-12-compiler#>

Recent Articles

[The state of static analysis in the GCC 12 compiler](#)

[Observability in 2022: Why it matters and how OpenTelemetry can help](#)

[Red Hat Summit 2022: A](#)

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

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

[🔗](#) [Code](#)

1 of 9
[Back to list](#)

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

Reverse engineering

Looking at a compiled program in order to figure out what it does and how it works

Usually assisted by tools

Disassembler

Decompiler

Strings

Often aided by dynamic analysis

Tracing

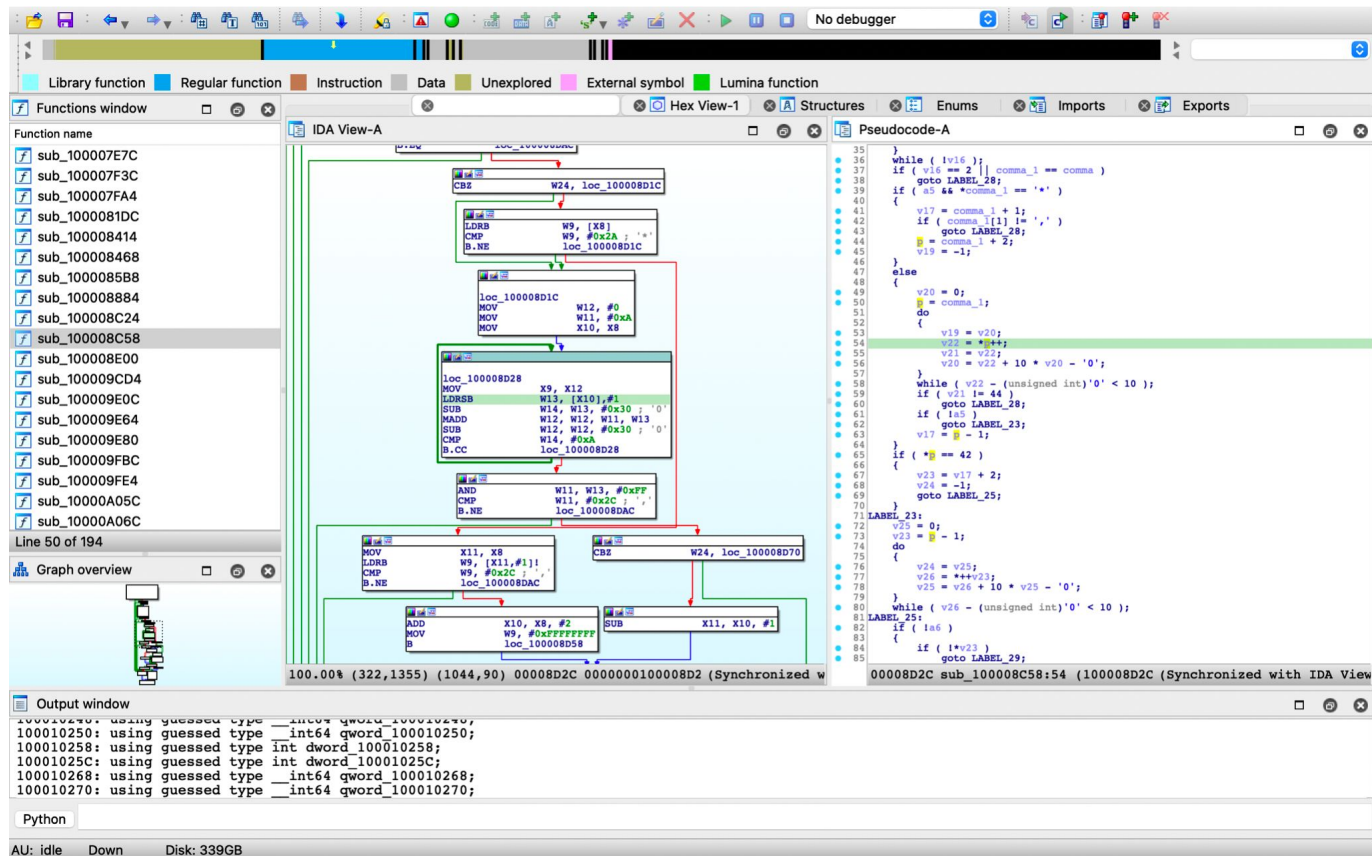
IDA Pro

Disassembly

Decompilation

Binary analysis

Scripting



The screenshot shows the Immunity Debugger interface with the following components:

- Top Menu Bar:** Edit, Analysis, Navigation, Search, Select, Tools, Window, Help.
- Left Pane:**
 - Program Tree:** Shows the loaded module structure, including sections like .bss, .data, .got.plt, .got, .dynamic, .jcr, .fini_array, .init_array, .eh_frame, .eh_frame_hdr, .rodata, .fini, .text, and .plt.
 - Symbol Tree:** Lists symbols, with `FUN_0040090c` selected under the `FUN_00400900` entry.
 - Data Type Manager:** Shows data types, with `BuiltInTypes` selected.
- Main Window:**
 - Listing: packedup:** Displays assembly code for the `packedup` function. It shows instructions like `segment_2.1`, `Loadable segment [0x400000 - 0x40097b] (disabled execute ...)`, and various `db`, `ds`, `dq`, `ddw`, and `dq` instructions.
 - Decompile: FUN_0040090c - (packedup):** Shows the decompiled C++ code. It includes a `void` function signature, variable declarations (`uint uVar1`, `int iVar2`, `ulong uVar3`, `int iVar4`, `bool bVar5`), and a loop that reads data from memory and writes it to a buffer.
- Bottom Pane:** Console - Scripting.

https://en.wikipedia.org/wiki/Ghidra#/media/File:Ghidra-disassembly,March_2019.png

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

Use a modern, memory safe language where possible: Go, Rust, Swift, etc.

Understand and document your threat model early in the design process

Design APIs and systems so that the easiest way to use them is the safe way

Treat all input from the outside adversarially, even if you trust the sender

Use a clean, consistent style throughout the codebase

Thank you!

bazad@cs.stanford.edu