Finding vulnerabilities
by fuzzing, dynamic and static analysis

Brandon Azad
Stanford CS155 guest lecture
April 15, 2021

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh and Zakir Durumeric at Stanford University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.
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

Intended states

Transition states

Unintended states
Classifying states

Intended states

Transition states

Unintended states

Unreachable states
Classifying states

- Intended states
- Transition states
- Unintended states
- Unreachable states

Vulnerabilities live here
Classifying states

Vulnerabilities live here

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

Vulnerabilities live here

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

Unreachable 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

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.
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

This is not 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 be successful?

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

```python
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
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.

---

**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:

```c
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_IWUSR, S_IXUSR, S_IXGRP, S_IWGRP, S_IWGRP, 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](https://github.com/google/syzkaller/blob/master/docssyscall_descriptions.md).

**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:

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

---

[https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md](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?
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

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 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

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.

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

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

4 weeks of fuzzing
87.3% coverage of the Qmage codec
5218 unique crashes
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

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

Relatively easy to get started

Fuzzing doesn’t find all types of bugs

This code parses untrusted data

Should I write a fuzzer?

Yes
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.

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
- 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 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:71
 #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
```

[Link to GitHub page](https://github.com/google/sanitizers/wiki/AddressSanitizer)
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
- Use-after-free
- Use-after-return
- Use-after-scope
- 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!

---

SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main

[https://github.com/google/sanitizers/wiki/AddressSanitizer](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

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
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!

```python
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.
Soundness vs Completeness

- Sound (over-approximate) analysis
- Possible program behaviors
- Complete (under-approximate) analysis
Is this program safe?
Yes, it is safe.
This program will not crash.
Try analyzing without approximating…

Non-termination!
Therefore, need to approximate
Try analyzing without approximating...

Non-termination!
Therefore, need to approximate
Abstraction

Concrete Domain of Integers

\begin{align*}
  x &= 5 \\
  x &= -5 \\
  x &= 0
\end{align*}

Abstract Domain of Signs

- Positive ints
- Negative ints
- Zero
Abstraction

Concrete Domain of Integers

\begin{align*}
    x &= 5 \\
    x &= -5 \\
    x &= 0
\end{align*}

Abstract Domain of Signs

\begin{align*}
    &\oplus \text{ Positive ints} \\
    &\ominus \text{ Negative ints} \\
    &\odot \text{ Zero}
\end{align*}
Abstraction

Concrete Domain of Integers

- $x = 5$
- $x = -5$
- $x = 0$
- $x = b \mathrm{?} -1 : 1$

Abstract Domain of Signs

- $\oplus$ Positive ints
- $\ominus$ Negative ints
- $\circ$ Zero
Abstraction

Concrete Domain of Integers

- $x = 5$
- $x = -5$
- $x = 0$
- $x = b ? -1 : 1$

Abstract Domain of Signs

- $\oplus$ Positive ints
- $\ominus$ Negative ints
- $\odot$ Zero
- $\top$ All integers
Abstraction

Concrete Domain of Integers

- $x = 5$
- $x = -5$
- $x = 0$
- $x = b ? -1 : 1$
- $x = y / 0$

Abstract Domain of Signs

- $\oplus$ Positive ints
- $\ominus$ Negative ints
- $\odot$ Zero
- $\top$ All integers
Abstraction

Concrete Domain of Integers

\[ x = 5 \]
\[ x = -5 \]
\[ x = 0 \]
\[ x = b \ ? \ -1 : 1 \]
\[ x = y / 0 \]

Abstract Domain of Signs

\[ \oplus \text{ Positive ints} \]
\[ \ominus \text{ Negative ints} \]
\[ \odot \text{ Zero} \]
\[ \top \text{ All integers} \]
\[ \bot \text{ No integers (undefined)} \]
Try analyzing with “signs” approximation…
Try analyzing with “signs” approximation...

```
entry
X ← 0

X = 0
Is Y = 0 ?

X = 0
Yes
X ← X + 1
X = pos
X = T
Is Y = 0 ?

X = T
Yes
Is X < 0 ?

X = T
Yes
crash

X = neg
X ← X - 1

X = T
No
exit

X = T
No
```

Might Crash
Try analyzing with “path-sensitive signs” approximation...

terminates...
... no false alarm
... soundly proved never crashes
Tainting Checkers

Unchecked data accepted from untrusted source

User input, network packets, parsed files

Unvetted data taints other data transitively

Tainted data used as an operator

- system() 
- printf() 
- malloc() 
- strcpy()

- Sent to RDBMS
- HTML Rendered

Command Injection  
Format String Manipulation  
Int/Buffer overflow  
Buffer overflow  
SQL Injection  
Cross Site Scripting Attacks
Checking for Unsanitized Integers

Warn when unchecked integers from untrusted sources reach trusting sinks

Linux: 125 errors, 24 false; BSD: 12 errors, 4 false
Example Untrusted Integer

Remote exploit, no length checks

/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
  msg = skb->data;
  ...
  memcpy(cmd.parm.setup.phone,
      msg->msg.connect_ind.addr.num,
      msg->msg.connect_ind.addr.len - 1);
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
CodeQL (Semmle)

Query language for finding patterns in large codebases

“SQL for searching code”

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

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

Manual analysis
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

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 target device)

4) Run track.py again on the target device and observe the output.

This helped the team to identify the root cause of this issue and work towards a fix.
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
Ghidra

Similar to IDA

Open source

Written by the NSA (no, really)

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, etc.

Understand and document your threat model early in the design process

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

Use a clean, consistent style throughout the codebase
Thank you!

bazad@cs.stanford.edu