Finding good bugs in very good code

Fraser Brown, Dawson Engler, Deian Stefan, Craig Disselkoen
We like finding bugs
0. Write checking tool
0. Write checking tool
1. Take large codebase that isn’t aggressively checked
0. Write checking tool
1. Take large codebase that isn’t aggressively checked
2. Look for null pointer dereferences
0. Write checking tool
1. Take large codebase that isn’t aggressively checked
2. Look for null pointer dereferences
3. Send bug reports
0. Write checking tool
1. Take large codebase that isn’t aggressively checked
2. Look for null pointer dereferences
3. Send bug reports
...
0. Write checking tool
1. Take large codebase that isn’t aggressively checked
2. Look for null pointer dereferences
3. Send bug reports
...
...
We want to find good bugs
We want to find good bugs in very good code
This talk: security bugs in Chrome, Firefox, SQLite
Chrome and Firefox browsers run fuzzers
Chrome and Firefox browsers run fuzzers
Chrome and Firefox browsers run fuzzers

- Chrome has **25,000** machines fuzzing **24/7**
Chrome and Firefox browsers run fuzzers

- Chrome has 25,000 machines fuzzing 24/7
- As a result, Chrome testing hits ~75% line coverage of browser
Chrome and Firefox browsers run fuzzers

- Chrome has 25,000 machines fuzzing 24/7
- As a result, Chrome testing hits ~75% line coverage of browser
- Firefox has an entire team devoted to fuzzing
Chrome and Firefox browsers run fuzzers

- Chrome has 25,000 machines fuzzing 24/7
- As a result, Chrome testing hits ~75% line coverage of browser
- Firefox has an entire team devoted to fuzzing
- Firefox runs 7+ different fuzzers (2016)
Chrome and Firefox browsers run fuzzers *with sanitizers*
Chrome and Firefox browsers run fuzzers *with sanitizers*

Look for bugs in code while it executes
Chrome and Firefox browsers run fuzzers *with sanitizers*

- Chrome’s 24/7 fuzzers use:
Chrome and Firefox browsers run fuzzers *with sanitizers*

- Chrome’s 24/7 fuzzers use:
  - ASAN: heap and stack buffer overflows, etc
Chrome and Firefox browsers run fuzzers *with sanitizers*

- Chrome’s 24/7 fuzzers use:
  - ASAN: heap and stack buffer overflows, etc
  - MSan: uninitialized memory bugs
Chrome and Firefox browsers run fuzzers *with sanitizers*

- Chrome’s 24/7 fuzzers use:
  - ASAN: heap and stack buffer overflows, etc
  - MSan: uninitialized memory bugs
  - UBSan: undefined behavior bugs
Chrome and Firefox run static checkers
Chrome and Firefox run static checkers

Check code without running it
Chrome and Firefox run static checkers
Chrome and Firefox run static checkers

- Chrome: Clang C++, unix, and core static checkers
Chrome and Firefox run static checkers

- Chrome: Clang C++, unix, and core static checkers
- Firefox:
Chrome and Firefox run static checkers

- Chrome: Clang C++, unix, and core static checkers
- Firefox:
  - Automatic static checkers on every patch
Chrome and Firefox run static checkers

- Chrome: Clang C++, unix, and core static checkers
- Firefox:
  - Automatic static checkers on every patch
  - Clang checkers
Chrome and Firefox run static checkers

- Chrome: Clang C++, unix, and core static checkers
- Firefox:
  - Automatic static checkers on every patch
  - Clang checkers
  - Coverity checkers
Chrome and Firefox run static checkers

- Chrome: Clang C++, unix, and core static checkers
- Firefox:
  - Automatic static checkers on every patch
  - Clang checkers
  - Coverity checkers
  - Infer checkers
Chrome and Firefox browsers have bug bounty programs
SQLite:
SQLite:

- Part of Chrome and Firefox
SQLite:

- Part of Chrome and Firefox
- 700x more test code than database code
SQLite:

- Part of Chrome and Firefox
- 700x more test code than database code
- 100% branch test coverage
SQLite:

- Part of Chrome and Firefox
- 700x more test code than database code
- 100% branch test coverage
- Re-tests with sanitizers for three different compilers on:
  - 32- and 64-bit builds
  - Big- and little-endian architectures
Few bugs with our existing checking systems
New approach: Static analysis + symbolic execution
New approach: Static analysis + symbolic execution

Check code without running it
New approach: Static analysis + symbolic execution

“Run” program over all possible paths and values
New approach: Static analysis + symbolic execution
New approach: Static analysis + symbolic execution

Symbolic: checking:
- UC-Klee
- Woodpecker

Combined static and symbolic:
- Chopper
- Dowser
- Deadline
- ... more!
New approach: Static analysis + symbolic execution
New approach: Static analysis + symbolic execution

- Static analysis identifies many potential errorsites ($)$
New approach: Static analysis + symbolic execution

- Static analysis identifies many potential errorsites ($)

- Symbolic execution jumps directly to candidate errorsite and executes ($$$$$)
New approach: Static analysis + symbolic execution

- Static analysis identifies many potential errorsites ($)
- Symbolic execution jumps directly to candidate errorsite and executes ($$$$$)
New approach: Static analysis + symbolic execution

- Static analysis identifies many potential errorsites ($)
  - Programmer-written static extension (avg. 175 LOC)
- Symbolic execution jumps directly to candidate errorsite and executes ($$$$$)
New approach: Static analysis + symbolic execution

- Static analysis identifies many potential errorsites ($)
  - Programmer-written static extension (avg. 175 LOC)
- Symbolic execution jumps directly to candidate errorsite and executes ($$$$$$)
  - Programmer-written symbolic checkers (avg. 40 LOC)
New approach: Static analysis + symbolic execution
New approach: Static analysis + symbolic execution

: ModuleID = 'undefbc'
source_filename = "undef.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

LLVM IR File(s)
New approach: Static analysis + symbolic execution

LLVM IR File(s)

Static extension
New approach: Static analysis + symbolic execution

: ModuleID = 'undef.bc'
source_filename = 'undef.c'
target datalayout = 'e-m:e-i64:64-f80:128-n8:16:32:64-S128'
target triple = 'x86_64-pc-linux-gnu'

LLVM IR File(s)  Static extension  Suspicious path

Alloc x => Uninit x
Store y x => Init x
Load x => Error x
...

Alloc x
Store y z
Load x
New approach: Static analysis + symbolic execution

LLVM IR File(s) → Static extension → Suspicious path → Symbolic checker

: ModuleID = 'undef.bc'
  source_filename = "undef.c"
  target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
  target triple = "x86_64-pc-linux-gnu"

Alloca x => Uninit x
Store y x => Init x
Load x => Error x
...

Alloca x
Store y z
Load x

V = Load shadow x
If isSet V
Then Bug
Else No Bug
New approach: Static analysis + symbolic execution

: ModuleID = 'undef.bc'
source_filename = "undef.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

LLVM IR File(s) → Static extension → Suspicious path → Symbolic checker

Alloc x => Uninit x
Store y x => Init x
Load x => Error x
...

Alloc x
Store y z
Load x

V = Load shadow x
If isSet V
Then Bug
Else No Bug

Bug
Heap out-of-bounds bug, CVE 2019-XXXX
const int stride = input + 1;
...
a = my_malloc( (sizeof(int)+12)*stride );

if( a==0 ){  
    return -1;
}
...
memset(a, 0, sizeof(int)*(stride) );
Heap out-of-bounds bug, CVE 2019-XXXX

const int stride = input + 1;
...
a = my_malloc((sizeof(int)+12)*stride);

if( a==0 ){
    return -1;
}
...
memset(a, 0, sizeof(int)*stride);
Heap out-of-bounds bug, CVE 2019-XXXX

const int stride = input + 1;
...

a = my_malloc( (sizeof(int)+12)*stride );

if( a==0 ){
  return -1;
}
...

memset(a, 0, sizeof(int)*(stride) );
Heap out-of-bounds bug, CVE 2019-XXXX

const int stride = input + 1;
...
a = my_malloc( (sizeof(int)+12)*stride );
if( a==0 ){
  return -1;
}
...
memset(a, 0, sizeof(int)*(stride) );

32-bit multiplication
64-bit multiplication
Heap out-of-bounds bug, CVE 2019-XXXX

```c
const int stride = input + 1;
...
a = my_malloc((sizeof(int)+12)*stride);
if( a==0 ){
  return -1;
}
...
memset(a, 0, sizeof(int)*(stride));
```

- 32-bit multiplication WRAPS TO SMALL
- 64-bit multiplication STAYS BIG
Heap out-of-bounds bug, CVE 2019-XXXX

```c
const int stride = input + 1;
...
a = my_malloc( (sizeof(int)+12)*stride );
if( a==0 ){
    return -1;
}
...
memset(a, 0, sizeof(int)*(stride) );
```

Small allocation

Very large out-of-bounds write
Static extension (heap out-of-bounds)
Static extension (heap out-of-bounds)

```c
x = malloc (y);

x[y - 1] = 0;
```
Static extension (heap out-of-bounds)

\[ x = \text{malloc}(y); \]

\[ x[y - 1] = 0; \]
Static extension (heap out-of-bounds)

```c
x = malloc (y);

x[y - 1] = 0;
```

Dependency between $y$ and $y - 1$ => mark suspicious path
Static extension (heap out-of-bounds)

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]
Static extension (heap out-of-bounds)

\[ x = \text{malloc} \left( y \right); \]

\[ x[y - 1] = 0; \]

Save \text{sizeof} (x) = y
Static extension (heap out-of-bounds)

\[ x = \text{malloc}(y); \]

\[ x[y - 1] = 0; \]

Save \( \text{indexSize}(x) = y - 1 \)
Static extension (heap out-of-bounds)

\[ x = \text{malloc}(y); \]

\[ x[y - 1] = 0; \]

\[ \text{indexSize}(x) = y - 1 \]
\[ \text{sizeOf}(x) = y \]
Symbolic checker (heap out-of-bounds)
Symbolic checker (heap out-of-bounds)

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]
Symbolic checker (heap out-of-bounds)

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]

\[ y - 1 > y \quad \Rightarrow \quad \text{report bug} \]
Symbolic checker (heap out-of-bounds)

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]
Symbolic execution engine examines all possible values along path

\[ x = \text{malloc} \ (y); \]
\[ x[y - 1] = 0; \]

Can \( y - 1 > y \) ?
“Constraints” express lines of code as logical formulas

```
a && b && c || d || e ....
```
“Constraints” express lines of code as logical formulas

\[ a \land \land b \land \land c \lor d \lor e \ldots \]

\[ a = \text{true} \]
\[ b = \text{true} \]
\[ c = \text{true} \]
\[ d = \text{false} \]
\[ e = \text{true} \]
“Constraints” express lines of code as logical formulas

\[ a \land \neg a \land b \land c \lor d \lor e \ldots \]
“Constraints” express lines of code as logical formulas

a && not a && b && c || d || e ....

UNSAT
“Constraints” express lines of code as logical formulas

```
malloc (y)
x [y -1]
```

Suspicious path
“Constraints” express lines of code as logical formulas

Suspicious path

```
...  
malloc (y)  
x [y -1]  
```

Constraints

```
x = 0xdeadbeef

tmp = y - 1
...
```
"Constraints" express lines of code as logical formulas

```
void main() {
  ...  
  malloc(y)  
  x[y-1]  
  x = 0xdeadbeef  
  tmp = y - 1  
  ...  
  y - 1 > y  
}
```

Suspicious path

Constraints

Bug constraints
“Constraints” express lines of code as logical formulas

```
... 
malloc(y) 
... 
x[y-1]
```

Suspicious path

```
x = 0xdeadbeef 
tmp = y - 1 
... 
```

Constraints

```
y - 1 > y 
... 
```

Bug constraints

```
SAT
or
UNSAT
```

SMT Solver
Symbolic checker for heap out-of-bounds

x = malloc (y);

x[y - 1] = 0;
1. Symbolic engine translates line

\[ x = \text{malloc} \ (y); \]
\[ x[y - 1] = 0; \]
1. Symbolic engine translates line

\[
x = \text{malloc} \ (y);
\]

\[
x[y - 1] = 0;
\]

\[
X = \text{new concrete location (e.g., 0xdeadbeef)}
\]
2. Symbolic checker examines line

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]
1. Symbolic engine translates line

\[
x = \text{malloc} \ (y);
\]

\[
x[y - 1] = 0;
\]
1. Symbolic engine translates line

```c
x = malloc (y);
x[y - 1] = 0;
```

1. tmp = y - 1
1. Symbolic engine translates line

```
x = malloc (y);
x[y - 1] = 0;
```

1. tmp = y - 1
2. ptr = x + sizeof(x) * tmp
1. Symbolic engine translates line

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]

1. \( \text{tmp} = y - 1 \)
2. \( \text{ptr} = x + \text{sizeof}(x) \times \text{tmp} \)
3. \( \text{mem}[\text{ptr}] = 0 \)
2. Symbolic checker examines line

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]
2. Symbolic checker examines line

\[ x = \text{malloc} \ (y); \]
\[ x[y - 1] = 0; \]

Know: \( \text{sizeOf}(x) = y \)
Know: \( \text{indexSize}(x) = y - 1 \)
2. Symbolic checker examines line

\[ x = \text{malloc} \ (y); \]

\[ x[y - 1] = 0; \]

know \( \text{sizeof}(x) = y \)
know \( \text{indexSize}(x) = y - 1 \)
assert \( y - 1 > y \)
3. Query SMT solver

```c
x = malloc (y);

x[y - 1] = 0;
```
3. Query SMT solver

\[
x = \text{malloc}(y); \\
x[y - 1] = 0;
\]

SAT

\[
y = 0 \\
x = 0x\text{deadbeef}
\]
Results of the heap out-of-bounds checker
Results of the heap out-of-bounds checker

- 22 bugs
Results of the heap out-of-bounds checker

- 22 bugs
- High-severity pattern in SQLite
  - Patched within 7hrs
  - Backported
  - Bounty
  - CVE
Results of the heap out-of-bounds checker

- 22 bugs
- High-severity pattern in SQLite
  - Patched within 7hrs
  - Backported
  - Bounty
  - CVE
- 3 security audits
Results of the heap out-of-bounds checker

- 22 bugs
- High-severity pattern in SQLite
  - Patched within 7hrs
  - Backported
  - Bounty
  - CVE
- 3 security audits
- 1 audit of Firefox checking tools
Results of the heap out-of-bounds checker

- 22 bugs
- High-severity pattern in SQLite
  - Patched within 7hrs
  - Backported
  - Bounty
  - CVE
- 3 security audits
- 1 audit of Firefox checking tools
- 4 false positives
Three checkers
Three checkers

- Heap out-of-bounds
Three checkers

- Heap out-of-bounds
- Uninitialized memory
Three checkers

- Heap out-of-bounds
- Uninitialized memory

```c
int x;
int y = x + 5;
```
Three checkers

- Heap out-of-bounds
- Uninitialized memory
- Simple stack out-of-bounds
Three checkers

- Heap out-of-bounds
- Uninitialized memory
- Simple stack out-of-bounds

```c
int x[5];
return x[6];
```
Challenges
Challenge: Unknown state
int foo (int * buff, int x) {

    int val = buff[x];
}

Challenge: Unknown state
Challenge: Unknown state

```c
int foo (int * buff, int x) {
    int val = buff[x];
}
```
Challenge: Unknown state
Solutions: Unknown state

1. Target specific errors instead of general correctness
Ask bad questions get bad answers

```c
int foo (int * buff, int x) {
    int val = buff[x];
}
```
Solutions: Unknown state

1. Target specific errors instead of general correctness
const int stride = input + 1;
...
a = my_malloc((sizeof(int)+12)*stride);

if( a==0 ){
  return -1;
}
...
memset(a, 0, sizeof(int)*(stride));
const int stride = input + 1;
...
a = my_malloc( (sizeof(int)+12)*stride );

if( a==0 ){
    return -1;
}
...
memset(a, 0, sizeof(int)*(stride) );
Solutions: Unknown state

1. Target specific errors instead of general correctness
2. All paths are internally consistent
Constraint solver rejects infeasible paths

if (p) ....
if (!p) ....
Constraint solver rejects infeasible paths

if (p) ....
if (!p) ....
Solutions: Unknown state

1. Target specific errors instead of general correctness
2. All paths are internally consistent
3. Tool is simple to support checker-specific tricks
Results so far (3mos, 1 person):
Results so far (3mos, 1 person):

- 3 checkers (2 out-of-bounds, 1 uninitialized memory)
- Every checker has found at least one bountied browser bug
- 2 CVEs
- High severity, exploitable Chrome pattern (~13 instances)
- 4 medium-severity bugs
- 4 low-severity bugs
- 12+ patches (+3 already patched, +5 mystery patches)
- 22+ patched functions
- 2 security audits
- 48 reported bugs, 38+ confirmed bugs, 18 false positives