Software Security VI: Symbolic Execution

Chengyu Song

Slides modified from Dawn Song
Administrivia

• Lab1
  • Due: **Today** Oct 23 11:59pm

• Lab2
  • Due: Monday Nov 13 11:59pm

• Exploit stack buffer overflow
Finding vulnerabilities

- Fuzzing
  - Input generation
  - Feedback
  - Error detection
Limitations of fuzzing

- Some branch predicates are very hard to satisfy based on random mutation/generation
  - Magic number
  - Checksum
  - ...

Example

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
    }...
}
```
Quiz: coverage (1)

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len & 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```

- # of lines, branches, paths?
Quiz: coverage (2)

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```

- # of lines = 10, branches = 3, paths = 4
- # of input to full coverage?
Quiz: coverage (3)

```c
foo(unsigned input)
{
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```

- # of lines = 10, branches = 3, paths = 4
- # of input to full coverage, lines = 3, branches = 4, paths = 4
Quiz: coverage (4)

What is the expected number of inputs required to cover the highlighted line, using random test-case generation? Assuming unsigned is 32 bits.
We can evaluate the efficiency of an input generation technique using the following formula:

\[
\frac{\text{minimum # of inputs}}{\text{expected # of inputs}}
\]

A technique is efficient if the minimum value is close to expected value.

A technique is NOT efficient if minimum \(<<\) expected value.

There are many cases where minimum \(<<\) expected for fuzzing.
Symbol reasoning vs. fuzzing

- Fuzzing: individual input in the input space

```c
unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)

s = len;
if (len % 2 == 0)

s = len + 2;
buf = malloc(s);
read(fd, buf, len);
```

Set of all inputs
Symbol reasoning vs. fuzzing

- Symbol reasoning: example input in each input sets
Explicit vs. symbolic representation (1)

- Explicit representation

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>-3</th>
<th>-1</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

- Symbolic representation

\[ x > -4 \land x < 4 \land x \mod 2 = 1 \land y = x + 3 \]
Explicit vs. symbolic representation (2)

- Explicit representation

<table>
<thead>
<tr>
<th>x</th>
<th>-7</th>
<th>-5</th>
<th>-3</th>
<th>-1</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-4</td>
<td>-2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

- Symbolic representation

\[
x > -8 \land x < 8 \land x \mod 2 == 1 \land y == x + 3
\]
Explicit vs. symbolic representation (3)

• Explicit representation

```
+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
| x   | ... | -5   | -3   | -1   |  1   |  3   |  5   | ... |
+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
| y   | ... | -2   |  0   |  2   |  4   |  6   |  8   | ... |
+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

• Symbolic representation

\[
x \mod 2 == 1 \ \&\& \ y == x + 3
\]
Pros. vs. Cons.

• Advantages
  • Can be exponentially smaller than explicit representation of finite sets
  • Can represent infinite sets (e.g. regular expressions)
  • Generic algorithms (e.g. same algorithms for a certain type of formulas)

• Trade-offs
  • Performing basic operations may be expensive
  • Specialized algorithms are required
  • Difficult to predict size of representation
Satisfiability

• A formula is **satisfiable** if there is a way to assign values to variables and make the formula true
  
  • \((x > 0 \land x < 20 \land x = y + y)\) is satisfied by \((x:10, y:5)\)

• A formula is unsatisfiable if every assignment of values to variables makes the formula false
  
  • \((x > 0 \land x < 20 \land x = y + y \land x \% 2 = 1)\) is unsatisfiable
Solvers

- A solver determines if a formula is satisfiable
  - A SAT solver is a solver for propositional logic
  - An SMT solver is a solver for formulas in a first-order logic
Popular solvers

- Z3: https://github.com/Z3Prover/z3
- CVC4: https://github.com/CVC4/CVC4
- Yices2: http://yices.csl.sri.com/
- STP: http://stp.github.io/
Path as formulas

Write a formula for the values of `len` and `input` that execute the colored path.
Path as formulas (cont.)

Write a formula for the values of len and input that execute the colored path.

```c
if (input < UINT_MAX - 2)
    false
else
    true

unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)
    1
else
    false

s = len;
if (len & 2 == 0)
    false
else
    true

s = len + 2;
s = len;

buf = malloc(s);
read(fd, buf, len);

input < UINT_MAX - 2
&& len == input + 3
&& ! (len < 10)
& & ! (len & 2 == 0)
```
Path predicates

- A path predicate encodes the constraints that must be satisfied for a program path to be executed
- To construct a path predicate
  - Rename variables to have unique occurrences (symbolize)
  - Assignments become equalities
  - Branches are themselves, or negated
  - Sequence is conjunction
- Feasibility of a path == satisfiable of the path predicates
From path to bug

• Can you spot the bug involving the integer variables?

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```
From path to bug (cont.)

- Can you add an assertion to catch the bug?

```c
foo(unsigned input){
  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else {
      assert(len < UINT_MAX - 1);
      s = len + 2;
    }
    buf = malloc(s);
    read(fd, buf, len);
    ...
  }
}
```
CFG changes

```c
if (input < UINT_MAX - 2) {
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
        s = len;
    else
        s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
}
```

```c
if (input < UINT_MAX - 2) {
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
        s = len;
    else
        s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
}
```
Path to assertion violation

input < UINT_MAX - 2

&& len == input + 3

&& !(len < 10)

&& !(len % 2 == 0)

&& !(len < UINT_MAX - 1)
Violating input

• Is the path predicates satisfiable?

\[
\text{input} < \text{UINT\_MAX} - 2 \quad \land \quad \text{len} = \text{input} + 3 \quad \land \quad !(\text{len} < 10) \\
\quad \land \quad !(\text{len} \% 2 = 0) \quad \land \quad !(\text{len} < \text{UINT\_MAX} - 1)
\]

• Yes! When \(\text{input} = \text{UINT\_MAX} - 3\)
Summary

• How to use symbolic execution for bug finding
  • Augment a program with appropriate assertions
  • Symbolically execute a path
    • Create formula representing path constraint and assertion failure
    • Solve constraints with a solver
    • A satisfying assignment, if found, is an input triggering a bug
In practice ...

- Symbolic execution sounds great, but does it really work?
  - Not as expected
  - Symbolically execute a path is much slower than native execution
    (hundreds to thousands times slower)
  - Solving large formula is very slow
  - Most bugs are still found by fuzzing

- Hybrid approach
  - Invoke symbolic execution when fuzzer stuck at a branch
For next class ...

- Software Security VII: static analysis