Software Security: Vulnerability Analysis
Symbolic Execution
```
int parse(FILE *fp) {
    char cmd[256], *url, buf[5];
    fread(cmd, 1, 256, fp);
    int i, header_ok = 0;
    if (cmd[0] == 'G')
        if (cmd[1] == 'E')
            if (cmd[2] == 'T')
                if (cmd[3] == ' ')
                    header_ok = 1;
    if (!header_ok) return -1;
    url = cmd + 4;
    i = 0;
    while (i<5 && url[i]!='\0' && url[i]!='\n') {
        buf[i] = tolower(url[i]);
        i++;
    }
    buf[i] = '\0';
    printf("Location is %s\n", buf);
    return 0; }
```

- How many inputs (of length 256) are needed on expectation to cover past line 10 using random fuzzing?
Smallest number of test cases for coverage

- Smallest number of cases needed $\ll$ expected number via random fuzzing
- … so random fuzzing is a non-ideal brute force technique.
- How to reach the smallest number?
Path Exploration

1: int parse(FILE *fp) {
2:   char cmd[256], *url, buf[5];
3:   fread(cmd, 1, 256, fp);
4:   int i, header_ok = 0;
5:   if (cmd[0] == 'G')
6:     if (cmd[1] == 'E')
7:       if (cmd[2] == 'T')
8:         if (cmd[3] == ' ')  
9:           header_ok = 1;
10:  if (!header_ok) return -1;
11:  url = cmd + 4;
12:  i=0;
13:  while (i<5 && url[i]!='\0' && url[i]!='\n') {
14:    buf[i] = tolower(url[i]);
15:    i++;
16:  }
17:  buf[i] = '\0';
18:  printf("Location is %s\n", buf);
18:  return 0; }

- Can you find a value for cmd to make the execution follow the outlined path?
Path Exploration

- Let’s try to find an expression for all values of \textit{cmd} that yield the highlighted path.
  - Consider \textit{cmd} as a \textbf{symbolic variable}
  - write down a \textbf{formula} to describe all the values of \textit{cmd} for following the highlighted path.

**SPECIFY INPUT as symbolic variable:**

\begin{itemize}
  \item \textbf{cmd}: cmd0 | cmd1 | cmd2 | cmd3 | cmd4 | cmd5 | cmd6 | cmd7 | cmd8 | cmd9
  \item \textbf{example}: ‘G’ | ‘E’ | ‘T’ | ‘ ’ | ‘h’ | ‘t’ | ‘t’ | ‘p’ | ‘:’ | ‘/’
\end{itemize}

(we’re considering input of length 10 just for this example)
if (!header_ok)
header_ok = 1;

(spec0 == 'G')

(cmd1 == 'E')

!(cmd2 == 'T')

(cmd0 == 'G') && (cmd1 == 'E') && !(cmd2 == 'T')

if (!header_ok)
if (cmd[0] == 'G')
if (cmd[1] == 'E')
if (cmd[2] == 'T')
if (cmd[3] == '')
header_ok = 1;

Dawn Song
Symbolic Execution

- Represent input as **symbolic values**
- Perform each operation on symbolic variables **symbolically**
- Registers and memory values dependent on inputs become **symbolic expressions**
- Certain conditions for conditional jump become **symbolic expressions** as well

- **Path predicate** *(highlighted path)*:
Using a Solver

- Is there a value for cmd such that
  \((\text{cmd0} == 'G') \&\& (\text{cmd1} == 'E') \&\& !(\text{cmd2} == 'T'))\)?

- Give the symbolic formula to a solver
  - Z3, STP, Yices

- In this case, the solver returns Yes
  - e.g. cmd0 = 'G', cmd1 = 'E', cmd2 = 'x', cmd3 = 'x', …, cmd9 = 'x'

What to do next?
Path Exploration

1: f(unsigned int input) {
2:      unsigned int len, s;
3:      char *buf;
4:      len = input + 3;
5:      if (len < 10)
6:         then s = len;
7:      else if (len % 2 == 0)
8:         then s = len;
9:      else s = len + 2;
10:     buf = malloc(s);
11:     read(fd, buf, len);
12: ...}

- Can you write down a formula to describe all the values of `input` for following the red path?
Quiz: Path Exploration

- Can you write down a formula to describe all the values of \( \text{len} \) for following the red path?
- Let’s do it step by step:
  - (1) \( \text{len} == \text{input} + 3 \)
  - (2) \( \neg (\text{len} < 10) \)
  - (3) \( \neg (\text{len} \% 2 == 0) \)
- Final formula:
  - \((\text{len} == \text{input} + 3) \land \neg (\text{len} < 10) \land \neg (\text{len} \% 2 == 0)\)
Quiz: Bug Finding

1: f(unsigned int input) {
2:   unsigned int len, s;
3:   char *buf;
4:   len = input + 3;
5:   if (len < 10)
6:     then s = len;
7:   else if (len % 2 == 0)
8:     then s = len;
9:   else s = len + 2;
10:  buf = malloc(s);
11:  read(fd, buf, len);
12: ...

- Do you see a bug?
Bug Finding

```
1:f(unsigned int input){
2:   unsigned int len, s;
3:   char *buf;
4:   len = input + 3;
5:   if (len < 10)
6:     then s = len;
7:   else if (len % 2==0)
8:     then s = len;
9:     else s = len + 2;
10:  buf = malloc(s);
11:  read(fd, buf, len);
12:...}
```

- Can you add an assertion in the code to check for the bug?
Quiz: Bug Finding

1:
2: unsigned int len, s;
3: char *buf;
4: len = input + 3;
5: if (len < 10)
6: then s = len;
7: else if (len % 2 == 0)
8: then s = len;
9: else s = len + 2;
10: assert(s >= len);
11: buf = malloc(s);
12: read(fd, buf, len);
13: ...

- How do you find a value to trigger the bug?
1: f(unsigned int input) {
2:     unsigned int len, s;
3:     char *buf;
4:     len = input + 3;
5:     if (len < 10)  
6:         then s = len;
7:     else if (len % 2 == 0)  
8:         then s = len;
9:     else s = len + 2;
10:    assert(s >= len);
11:    buf = malloc(s);
12:    read(fd, buf, len);
13:    ...
}

- Can you write down the formula representing inputs that will make the program execution follow the red path and violate the assertion? : ________________
Constraint-based Automatic Test Case Generation

- Look inside the box
  - Use the code itself to guide the fuzzing
- Assert security/safety properties
- Explore different program execution paths to check for security properties
- Steps:
  1. Find inputs that will go down different program execution paths
  2. For a given path, need to check whether an input can trigger the bug, i.e., violate security property
Summary: Symbolic Execution for Bug Finding

- Symbolically execute a path
  - Create the formula representing: path constraint ^ assertion failure
  - Give the solver the formula
    - If returns a satisfying assignment, a bug found
- Reverse condition for a branch to go down a different path
  - Give the solver the new path constraint
  - If returns a satisfying assignment
    - The path is feasible
    - Found a new input going down a different path
- Related work & tools
  - DART, CUTE, KLEE, SAGE, BitBlaze, BAP, S2E
Automatic Test Case Generation for Vulnerability Discovery

- Two factors
  - Explore program execution space to reach vulnerability point
  - Detect vulnerability when reaching vulnerability point
Static Analysis
Static analysis goals

- Bug finding
  - Identify code that the programmer wishes to modify or improve

- Correctness
  - Verify the absence of certain classes of errors
Syntactic Analyzer

- Syntactic analysis checking for patterns
  - Flawfinder, RATS, etc.
  - strcpy(), strcat(), gets(), sprintf(), and the scanf()
Example: Check for missing optional args

- Prototype for open() syscall:
  ```c
  int open(const char *path, int oflag, /* mode_t mode */...);
  ```

- `fd = open("file", O_CREAT);`

  Typical mistake:

- Result: file has random permissions
Example: Chroot protocol checker

- Goal: confine process to a “jail” on the filesystem
  - chroot() changes filesystem root for a process
- Problem
  - chroot() itself does not change current working directory
Does this program ever crash?

```c
int parse(FILE *fp) {
    char cmd[256], *url, buf[5];
    fread(cmd, 1, 256, fp);
    int i, header_ok = 0;
    if (cmd[0] == 'G')
        if (cmd[1] == 'E')
            if (cmd[2] == 'T')
                if (cmd[3] == ' ')
                    header_ok = 1;
    if (!header_ok) return -1;
    url = cmd + 4;
    i=0;
    while (i<5 && url[i]!="\0" && url[i]!="\n") {
        assert(i>=0 && i<252 && i<5);
        buf[i] = tolower(url[i]);
        i++;
    }
    assert(i>=0 && i<5);
    buf[i] = '\0';
    printf("Location is %s\n", buf);
    return 0; }
```

Dawn Song
Does this program ever crash?

... program could crash...

feasible path!
Focusing on variable $i$, Try analyzing without approximating on input $url = "xx\0"$.

After the loop, $i$ could be many values, and only $i >= 5$ and $i < 0$ lead to a crash. Our analysis doesn’t show this…
buffer[i] = '0';

CRASH!

assert(i >= 0 && i < 5);

if (i < 5 && url[i] != '0' && url[i] != '
')

Manual testing only examines small subset of behaviors

Software

Dawn Song
yields too many false alarms
Abstract Interpretation

PROGRAM BEHAVIOR

Operating on actual values…

ABSTRACT REPRESENTATION

… we operate on a more useful abstract representation of those values.
Abstract Interpretation

\[ d_{\text{out}} = f(d_{\text{in}}) \]

**Dataflow elements**

**Transfer function**

\( d = X + 1 \) for \( X = 0 \) and \( X = 1 \).
Abstract Interpretation

\[
\begin{align*}
X = 0 & : X \leftarrow X + 1 \\
X = 1 & : \text{Is } Y = 0? \\
X = 1 & : X \leftarrow X + 1 \\
X = 1 & : \text{Is } Y = 0? \\
\end{align*}
\]

\[
\begin{align*}
d_{\text{in1}} & \rightarrow f_1 \\
d_{\text{out1}} & \rightarrow d_{\text{in2}} \\
d_{\text{in2}} & \rightarrow f_2 \\
d_{\text{out2}} & = f_2(d_{\text{in2}}) \\
d_{\text{out1}} & = f_1(d_{\text{in1}}) \\
d_{\text{out1}} & = d_{\text{in2}} \\
\end{align*}
\]
Abstract Interpretation

What is the space of dataflow elements, $\Delta$?
What is the least upper bound operator, $\sqcup$?

least upper bound operator
Example: union of possible values

\[
\begin{align*}
    d_{\text{out1}} &= f_1(d_{\text{in1}}) \\
    d_{\text{out2}} &= f_2(d_{\text{in2}}) \\
    d_{\text{join}} &= d_{\text{out1}} \sqcup d_{\text{out2}} \\
    d_{\text{join}} &= d_{\text{in3}} \\
    d_{\text{out3}} &= f_3(d_{\text{in3}})
\end{align*}
\]
\( X \neq \neg X = T \neq pos \)

\( X = \text{pos} \)

\( X = 0 \)

\( X = \text{neg} \)

\( X = \bot \)

\( X = 1 \)

\( X = \bot \)

\( \neq 0 \)

\( Y = 0 \)

\( Y \neq 0 \)

true

false

refinement lattice

Boolean formula lattice
What is the space of dataflow elements, $\Delta$?  
...the set of all intervals on the original domain.  
What is the least upper bound operator, $\sqcup$?  
...the smallest interval that contains all elements in the operand intervals.

... intuitively, this interpretation can be used to represent bounds on a variable.
Try analyzing with the “interval domain” approximation…

Ahah! Now all values of i are represented in the analysis.
However, this analysis would take much longer for a bigger loop (ex: if the condition were i<2^{32})

... therefore, might want to get to the end case faster...
Abstract Interpretation (Widening)

- Widening is a deliberate relaxation of your analysis.
- In this case, we will “widen” our intervals smartly as we iterate multiple times through the loop.
Try analyzing with the “interval domain” approximation with widening...

```
buf[i] = '\0';
```

```
assert(i>=0 && i<5);
```

```
is(i<5 && url[i]!='\0' && url[i]!='\n')?
```

```
i=[0,inf]  \(T\)
```

```
[i=0, inf]
```

```
[i=0, inf]  \(F\)
```

```
assert(i>=0 && i<5);
```

```
i=5, inf]  \(F\)
```

```
i=[5, inf]  \(T\)
```

```
i=[0,4]
```

```
CRASH!
```

```
buf[i] = '\0';
```

ahah, quick success!

... however, this widening would have yielded the same error even if the problem was fixed.

This shows one problem with this greedy yet sound reasoning.