Advanced Systems Security: Symbolic Execution

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Problem

- Programs have flaws
  - Can we find (and fix) them before adversaries can reach and exploit them?
  - Then, they become “vulnerabilities”
Vulnerabilities

- A program **vulnerability** consists of three elements:
  - A flaw
  - Accessible to an adversary
  - Adversary has the capability to exploit the flaw

- Which one should we focus on to find and fix vulnerabilities?
  - Different methods are used for each
Is It a Flaw?

• Problem: Are these flaws?
  ‣ Failing to check the bounds on a buffer write?
  ‣ `printf(char *n);`
  ‣ `strcpy(char *ptr, char *user_data);`
  ‣ `gets(char *ptr)`
    • From man page: “Never use `gets()`.”
  ‣ `sprintf(char *s, const char *template, ...)`
    • From gnu.org: “**Warning:** The `sprintf` function can be **dangerous** because it can potentially output more characters than can fit in the allocation size of the string `s`.”

• Should you fix all of these?
Is It a Flaw?

- Problem: Are these flaws?
  - `open(filename, O_CREAT);`
    - Need `O_EXCL` to ensure only new file is created
  - `stat, then open`
    - To check access/ownership before open
  - `mkdir, then chmod`
    - Change perms after making a directory
  - `x = x + n; (all ints)`
    - Yes, this is just addition…
  - `strncpy(filename, user_data, n), then open(filename)`
- Should you fix all of these?
Finding Flaws

• Researchers have explored program analysis methods to find flaws that may lead to vulnerabilities

• Dynamic analysis
  ‣ **Dynamic program analysis** is the analysis of computer software that is performed by running the program
  ‣ Can perform dynamic analysis easily on binaries

• Static analysis
  ‣ **Static program analysis** is the analysis of computer software that is performed without running the program
  ‣ Generally, use the source code, but not required
Dynamic Analysis

• Given the code, find whether the code may reach an execution state that fails an invariant
  ‣ Choose sequences of inputs to provide to the program
    • The choice of sequences can include random inputs (*fuzzing*)
  ‣ See what happens – does it violate an invariant?
    • Does it crash?
      ‣ Keep trying – *must try all paths on all inputs*
        • Not much in the way of feedback usually
  • Not as much theory here
    ‣ Can abstract conditions under which result holds
Dynamic Analysis – Limits

- Have both access control policy and program system calls
- Inherently results in **false negatives**
  - Attacks require special conditions
    - Current working directory, links, …
    - Can’t run all cases…
- Still, many **false positives**
  - Program code might defend itself
  - Manual audits impractical
  - In our study, **only 13% of adversary-accessible name resolutions are actually vulnerable**

```c
01 /* filename = /var/mail/root */
02 /* First, check if file already exists */
03 fd = open(filename, filg);
04 if (fd == -1) {
05     /* Create the file */
06     fd = open(filename, O_CREAT|O_EXCL);
07     if (fd < 0) {
08         return errno;
09     }
10 }
11 /* We now have a file. Make sure we did not open a symlink. */
12 struct stat fdbuf, filebuf;
13 if (fstat (fd, &fdbuf) == -1)
14     return errno;
15 if (lstat (filename, &filebuf) == -1)
16     return errno;
17 /* Now check if file and fd reference the same file, file only has one link, file is plain file */
18 if (fdbuf.st_dev != filebuf.st_dev
19 || fdbuf.st_ino != filebuf.st_ino
20 || fdbuf.st_nlink != 1
21 || filebuf.st_nlink != 1
22 || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
23     error ("At must be a plain file with one link"), filename);
24     close (fd);
25     return EINVAL;
26 }
27 /* If we get here, all checks passed.
28 Start using the file */
29 read(fd, ...)
```
Static Analysis

- Given the code, find whether the code may reach an execution state that fails an invariant
  - grep – find existence of a dangerous command
    - Syntactic
  - Examine possible executions
    - Semantic
      - “Run program in aggregate” – maintain an approximation of the possible values that can reach a statement
      - “Run in a non-standard way” – evaluate each function separately and stitch their executions together
- There is a lot of theory and experience on this!
Static Analysis

- “Run program in aggregate” – maintain an approximation of the possible values that can reach a statement
- “Run in a non-standard way” – evaluate each function separately and stitch their executions together

```
1: void expand(char *arg, unsigned char *buffer) { 8
2:    int i, ac;
3:    while (*arg) { 9
4:        if (*arg == '\\') { 10*
5:            arg++;
6:            i = ac = 0;
7:        } else if (*arg >= '0' && *arg <= '7') {
8:            do {
9:                ac = (ac << 3) + *arg++ - '0';
10:               i++;
11:            } while (i < 4 && *arg >= '0' && *arg <= '7');
12:            buffer++ = ac;
13:        } else if (*arg == '.') {
14:            *buffer++ = '.';
15:        } else if (*arg == '1') {
16:            i = *arg++;
17:        } else if (*arg++ == '-') {
18:            *buffer++ = '[';
19:            arg = -2;
20:        }
21:        continue;
22:    }
23:    while (i < ac) *buffer++ = i++;
24:    arg++ = /* Skip ']' */
25:    } else
26:    *buffer++ = *arg++;
27: }
28: }
29: }
30: ...
31: int main(int argc, char* argv[]) {
32:    int index = 1;
33:    if (argc > 1 && argv[index][0] == '-') {
34:        ...
35:    }
36:    ...
37:    expand(argv[index++], index);
38:    ...
39: }
```

Figure 1: Code snippet from MINIX’s tr, representative of the programs checked in this paper: tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are numbered on the right hand side.
Static Analysis – Limits

- Analyze program to find potentially vulnerable name resolution calls
  - Due to complexity of checks, mainly limited to TOCTTOU
- Deficiencies
  - False positives due to adversary inaccessibility
  - Our runtime study found only around 5% of name resolutions were accessible to adversaries
Symbolic Execution

- Analysis of a program using symbolic values rather than concrete
  - Similar to dynamic analysis in that programs are “executed” sequentially
    - Must evaluate all paths to find all flaws
    - But, not for all values
  - Similar to static analysis in that a symbolic value represents all possible values
    - Maintain symbolic values representing all possible
    - But, build model incrementally with execution
      - Eliminate unnecessary paths, take preferred paths
Symbolic Execution

• Example
  ‣ \( y = \text{read()} \) – what are the possible values of input? “\( s \)”
  ‣ \( y = 2 \times y \)
  ‣ if \( (y == 12) \) – what value of input leads to failure?
    ‣ fails()
  ‣ print("OK")

• Path constraint to “fail” – “\( y == 12 \)”
  ‣ So “\( y = 2 \times s = 12; s = 6 \)” is test case for failure
Symbolic Execution

- Versus Dynamic Analysis...

**Testing:** Multiple test sets are required to execute both paths on each branch point.

**Symbolic Execution:** Test input are specified as symbols. Case-splitting is performed at runtime to execute all paths.

---

**Input data**

- `data1=\textit{XB}`
- `data1=\textit{AB}`

---

**Process1**

- **Does data1 include \textit{X}?**
  - yes
  - no

- **Is length of data1 3?**
  - yes
  - no

**Process2**

- This part is not tested

**Process3**

- All paths are tested
• 1 – KLEE constructs symbolic command line arguments
  ▶ Initially there are no path constraints on any
  ▶ Need some notion of the system “environment” to constrain possible values (adversary access?)

• 2 – When KLEE hits conditional predicates (*), it uses a constraint solver to see which branches are legal
  ▶ KLEE forks execution along all legal paths
  ▶ Adds path conditions for each path option

• 3 – KLEE must pick path to execute next
  ▶ Want to choose paths that are likely to lead to flaws

1: void expand(char *arg, unsigned char *buffer) {
  8
2:   int i, ac;
  9
3:   while (*arg) {
  10
4:     if (*arg == '\\') {
  11*
5:       arg++;
  6:       i = ac = 0;
  7:     }
  8:     do {
  9:       ac = (ac << 3) + *arg++ - '0';
 10:      i++;
 11:     } while (i<4 && *arg>=0x0 && *arg<=7);
 12:     *buffer++ = ac;
 13:   } else if (*arg == '\0')
 14:     *buffer++ = *arg++;
 15:   } else if (*arg == '1') {
 16:     arg++;
 17:   i = *arg++;
 18:   if (*arg++ != '-') {
 19:     *buffer++ = '1';
 20:     arg -= 2;
 21:     continue;
 22:   }
 23:   ac = *arg++;
 24:   while (i <= ac) *buffer++ = i++;
 25:   arg++; /* Skip '}' */
 26: } else
 27:   *buffer++ = *arg++;
 28: }
 29: }
30: ...
31: int main(int argc, char* argv[]) {
  1
32:   int index = 1;
  2
33:   if (argc > 1 && argv[index] == '-') {
  3*
34:     ...
  4
35:   }
  5
36:   ...
  6
37:   expand(argv[index++], index);
  7
38:   ...
39: }

Figure 1: Code snippet from MINIX’s tr, representative of the programs checked in this paper: tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are numbered on the right hand side.
4 – KLEE identifies “dangerous operations” and checks if any values satisfy path constraints and cause dangerous operation

- Need to define such dangerous operations

5 – KLEE generates concrete values that when rerun will cause the “dangerous operation” to reveal the flaw

- Then KLEE continues running this path looking for other flaws – and runs other paths as well

6 – KLEE would run until all paths are explored

Figure 1: Code snippet from MINIX’s `tr`, representative of the programs checked in this paper: tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are numbered on the right hand side.
### KLEE Findings

#### Figure 7:

KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

<table>
<thead>
<tr>
<th>Command</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>paste -d\ \ abcdefghijklmnopqrstuvwxyz</td>
<td></td>
</tr>
<tr>
<td>pr -e t2.txt</td>
<td>KB5LEN3</td>
</tr>
<tr>
<td>tac -r t3.txt t3.txt</td>
<td>KB5LEN4</td>
</tr>
<tr>
<td>mkdir -Z a b</td>
<td>KB5LEN5</td>
</tr>
<tr>
<td>mkfifo -Z a b</td>
<td>KB5LEN6</td>
</tr>
<tr>
<td>mknod -Z a b p</td>
<td>KB5LEN7</td>
</tr>
<tr>
<td>md5sum -c t1.txt</td>
<td>KB5LEN8</td>
</tr>
<tr>
<td>ptx -F\ abcdefghijklmnopqrstuvwxyz</td>
<td></td>
</tr>
<tr>
<td>ptx x t4.txt</td>
<td>KB5LEN9</td>
</tr>
<tr>
<td>seq -f %0 1</td>
<td>KB5LEN10</td>
</tr>
</tbody>
</table>

---

```bash
pr -e t2.txt
```

```bash
tac -r t3.txt t3.txt
```

```bash
mkdir -Z a b
```

```bash
mkfifo -Z a b
```

```bash
mknod -Z a b p
```

```bash
md5sum -c t1.txt
```

```bash
ptx -F\ abcdefghijklmnopqrstuvwxyz
```

```bash
ptx x t4.txt
```

```bash
seq -f %0 1
```

```bash
t1.txt: "\t \tMD5(" |
```

```bash
t2.txt: "\b\b\b\b\b\b\b\b\b\t" |
```

```bash
t3.txt: "\n" |
```

```bash
t4.txt: "a"
```
Figure 9: Coverage of random vs. manual vs. KLEE testing for 15 randomly-chosen COREUTILS utilities. Manual testing beats random on average, while KLEE beats both by a significant margin.
KLEE Issues

• What should the “environment” be?
  ‣ Specified manually

• Maintain state of paths
  ‣ Each path has its own state

• Path explosion problem
  ‣ 10’s of thousands of states may be active

• Lots of constraint solving
  ‣ Expensive

• Choosing paths – random?
Virtual Machine Symbolic Execution Framework

Goals

- Scale to large systems
  - Selective symbolic execution
- Performance-analysis trade-off
  - Execution consistency models
- Operates on binaries, whole environment, analyze multiple paths at once

Slides from Vitaly Chipounov (EPFL)
Execution Consistency Models

- Specify the set of the paths to be analyzed
- Specify the FP/FN trade-off
• Execution Consistency Models
  ‣ Specify the set of the paths to be analyzed
  ‣ Specify the FP/FN trade-off

Consistency Models in S2E

```c
int main(argc, argv) {
  if (argc == 0) {
    ...
  }
  p = malloc(...);
  if (p == NULL) {
    ...
  }
  ...
}
```
• **Execution Consistency Models**
  ‣ Specify the set of the paths to be analyzed
  ‣ Specify the FP/FN trade-off

**Consistency Models in S2E**

```c
int main(argc, argv) {
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```
• Execution Consistency Models
  ‣ Specify the set of the paths to be analyzed
  ‣ Specify the FP/FN trade-off

Consistency Models in S2E

```c
int main(argc, argv) {
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```
• **Execution Consistency Models**
  - Specify the set of the paths to be analyzed
  - Specify the FP/FN trade-off

**Consistency Models in S2E**

```c
int main(argc, argv) {
    if (argc == 0) {
        ...
    }

    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```
S2E – SC-SE Consistency

- Strictly-Consistent, System-Level Consistency

```c
int main(argc, argv) {
    if (argc == 1) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```

System Input

Zero FN
Zero FP

Huge Number of Paths
S2E – SC-UE Consistency

- Strictly-Consistent, Unit-Level Consistency

Presence of FNs

```c
int main(argc, argv) {
    if (argc == 1) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}```
S2E – Relaxed Consistency

- Relaxed Consistency

```c
int main(argc, argv) {
    if (argc == 1) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```

Unit Input

Relax returned values

$p' \in \{\text{NULL}, p\}$

Introduces memory leak
### S2E – All Consistency Models

<table>
<thead>
<tr>
<th>Model</th>
<th>FNs w.r.t. unit</th>
<th>FP$s$ w.r.t. unit</th>
<th># system paths</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete</td>
<td></td>
<td></td>
<td></td>
<td>Valgrind</td>
</tr>
<tr>
<td>SC-SE</td>
<td></td>
<td></td>
<td></td>
<td>KLEE</td>
</tr>
<tr>
<td>SC-UE</td>
<td></td>
<td></td>
<td></td>
<td>DART</td>
</tr>
<tr>
<td>RC</td>
<td></td>
<td></td>
<td></td>
<td>RevNIC</td>
</tr>
<tr>
<td>CFG</td>
<td></td>
<td></td>
<td></td>
<td>Disassemblers</td>
</tr>
<tr>
<td>Local</td>
<td></td>
<td></td>
<td></td>
<td>DDT</td>
</tr>
</tbody>
</table>
Take Away

• Finding Flaws in Programs Is Important

• Program Analysis Techniques
  ‣ Dynamic Analysis – No False Positives – if environment is correct and understand program response – but hard to test whole program
  ‣ Static Analysis – Can eliminate False Negatives, but False Positives are common
  ‣ Symbolic Execution – Can address limitations of dynamic analysis (many test case needed) and static analysis (real values) in theory

• Any or all may be appropriate for you problem