Software security, secure programming (and computer forensics)

Lecture 10: Dynamic Analysis and Fuzzing

Master M2 on Cybersecurity

Academic Year 2016 - 2017
Program proof

Prove some logical assertion over the pgm variables at given control locations (e.g., using weakest precondition computations)

A sound and complete technique, but a non decidable problem . . .

▶ a semi-automated approach:
  → user inputs required to give loop invariants and prove logical implications

▶ tools can be used to assist the proof construction and/or to check the correctness of a manual proof . . .

Which usage in vulnerability detection ?

▶ for small or “highly critical” pieces of code
▶ as a (long term effort) technique to prove the correctness/security of a whole software or device Ex:
  ▶ the CompCert C compiler
  ▶ XXX
Abstract Interpretation

A static analysis technique

- allow to (automatically) reason about a whole program without executing it . . .
- but at the price of approximations due to undecidability problems:
  - over-approximations $\Rightarrow$ false positives
  - under-approximations $\Rightarrow$ false negatives
- example: value-set analysis (VSA)
  abstract representation = trade-off between accuracy and efficiency
  (e.g., intervals vs polyhedra vs . . .)
- can be leveraged with use-provided assertions . . .
  (to deal with library calls, “complex” code patterns, etc.)

Long (success) story in program verification $\Rightarrow$ numerous tools available!

But:

- not so effective on binary code, simple memory model
- not go “beyond the bug” ($\neq$ exploitability analysis)
- may provide too many false positives ?
What help for “security analysis”? 
“security analysis” = vulnerability detection

A pragmatic approach:

1. annotate the code with “vulnerability checks” (e.g., frama-c -rte)
   i.e., assertions to detect integer overflows, invalid memory accesses
   (arrays, pointers), etc
2. run a VSA
   → reveals a lot of hot spots (= unchecked assertions)
3. add user-defined assertions when possible . . .
   e.g., function pre/post conditions, loop invariants, extra information . . .
   → consider proving (some of) these assertions ?
4. run the VSA again . . .

⇒ a set of potential vulnerabilities remains, to be discharged by other means,
possibly on a program slice
(false positive ? real bug but harmless w.r.t security ? real vulnerability ?)

Rk: some static analysis tools also provide bug finding facilities
(i.e., no false postives, . . . but false negatives instead)
(Dynamic) Symbolic Execution (DSE)

Run a subset of finite program executions . . .

▶ a test generation technique
▶ can be coverage-oriented or goal-oriented

Principle:
Associate a path predicate $\varphi_\sigma$ to each path $\sigma$ of the CFG:

\[
(\exists \text{ a variable valuation } v \text{ s.t. } v \models \varphi_\sigma) \iff (v \text{ covers } \sigma)
\]

($\varphi_\sigma$ is the conjunction of all boolean conditions associated to $\sigma$ in the CFG)

▶ solving $\varphi_\sigma$ indicates if $\sigma$ is feasible
▶ iteration over a finite subset of the CFG paths . . .

In practice:
▶ express $\varphi_\sigma$ in a decidable logic fragment (e.g., SMT).
▶ may need to concretize some symbolic variables (loosing completeness of the path predicate)
DSE for vulnerability analysis

- an effective and flexible test generation & execution technique
  - can be used on “arbitrary” code
dynamic allocation, complex math. functions, binary code
  - trade-off between correctness, completeness and efficiency
    (ratio between symbolic and concrete values)
  - can be used in a coverage-oriented (bug finding) or goal-oriented
    (vulnerability confirmation) way
    **Ex:** out-of-bound array access, arithmetic overflow, etc.

⇒ widely used in vuln. detection and exploitability analysis

- numerous existing tools . . .

- however, not all problems solved (yet ?), e.g.:
  - “path explosion” problem on large codes
  - can be rather slow (compared with *fuzzing*)
Back to (pure) Dynamic Analysis

run an instrumented version of the target program to collect runtime information on the program behavior

Some very appealing features

- can be used on (almost) every kind of applications\(^1\): binary code, complex functions, large applications, virtual execution environment, etc.
- several execution-level applications:
  - detect assertion violations
  - profiling
  - data-flow analysis (e.g., taint analysis)
  - source-level engineering

⇒ rather well adapted for security analysis / vulnerability detection

Main requirements

- code instrumentation facilities + instrumented code execution
- find **good program inputs**!
  ⇒ makes sense within testing or fuzzing campaigns

\(^1\) as long as instrumentation is feasible, see later
More details on instrumentation techniques and tools

see Nick Sumner’s slides . . .
Fuzzing, or how to cheaply produce ”interesting” program inputs?

A major concern for dynamic analysis:

*feed the target program with good input values …*

**Fuzzing = combination of several possible strategies**

- human expertise, (non) typical use-cases
- dynamic symbolic execution
- other code or input space coverage techniques
- (pseudo)-random values, (pseudo)-random mutations of given inputs
- etc.

**Key elements**

- input generation should be fast enough to maximize the # of executions
- need a test oracle
  - from crash detection to complex dynamic property checkers

⇒ one of the most effective vulnerability detection technique to date . . .!
A trendy and powerful fuzzer: AFL

American Fuzzy Loop
A general-purpose fuzzing tool
(not specific to a set of applications, protocols, etc.)
- C, C++, Objective C
- Python, Go, RUST, OCaml, ...
- (any) binary code (with QEMU)

governing principles
- speed
- reliability
- ease-of-use
- availability and code sharing ...

lcamtuf.coredump.cx/afl/
Fuzzing algorithm

branch coverage-oriented mutation-based fuzzing

Repeat until a time budget is reached:
1. pick a input from a queue
2. mutate it
3. run it
4. if "coverage increases" put the new input in the queue

Detailed qlgo:
Code instrumentation

Lightweight instrumentation to capture:

- branch coverage
- coarse branch hits count

→ Use a 64Kb shared memory to record (src,dest) branch hits code injected at each branch point:

```c
// identifies the current basic block
cur_location = <compile-time-random-value> ;
```

```c
// mark (and count) a tuple hit
sh_mem[cur_location ^ prev_location]++ ;
```

```c
// to preserve directionality
prev_location = cur_location >> 1;
```

trade-off in the size of this memory: #collision vs efficiency (L2 cache)

Detecting new behaviors:

- maintains a global map of tuple (= branch) seen so far
- only inputs creating new tuples are added to the input queue (others are discarded)

Rk: branches are considered outside their context

→ may ignore new paths ...
Some further heuristics

► Tuple hits counted using buckets
  (1, 2, 3, 4-7, 8-15, ..., 128+)
  inputs leading to a change of bucket are added to the input queue

► Strong time limits for each executed path
  motivation: better to try more paths than slow paths ...

► Periodic queue minimization
  → select a small subset covering the same tuples mix between
  ▶ execution latency + file size
  ▶ ability to cover new tuples
  can be used as well by other external tools ...

► Trimmig input files
  → reduce their size to speed-up fuzzing
  e.g., remove the size of variable lengths blocks

⇒ favorite seed = fastest and smallest input exercising a tuple
Mutation strategy

no relationships between mutations and program states

- deterministic (sequentially):
  - flip bits (<> lengths and stepovers)
  - add/subtract small integers
  - insert known interesting integers (0, 1, INT_MAX, etc.)

- non deterministic:
  insertion, deletion, arithmetics, etc.

Dictionaries
used to retrieve/build syntax of verbose input language
(e.g., JavaScript, SQL, etc.)
Crah unicity

- faulty address is too coarse (e.g., crash in strcmp)
- call stack checksum is too slow

AFL
a crash is new if
- crash trace include a new tuple wrt existing crashes
- crash trace miss some tuple wrt existing crashes
Also provide some support for crash investigation...