UFR IM²AG
UNIVERSITÉ Grenoble Alpes
Grenoble INP Ensimag

Software security, secure programming

Fuzzing

Master M2 Cybersecurity

Academic Year 2021 - 2022

Outline

Fuzzing (or how to *cheaply* produce *useful* program inputs)

A concrete fuzzer example: AFL (with a short demo)

Making the fuzzing smarter: (Dynamic) Symbolic Execution

Conclusion

Fuzzing a software ?

A (pretty old !) testing method for software (and hardware !) . . .

$\hookrightarrow$ an application to software security = vulnerability detection

Main principle

run the program in order to detect “unsecure behaviors”
(from simple crashes to complex security property violations)

Fuzzing a software ?

A (pretty old !) testing method for software (and hardware !) ...

↪ an application to software security = vulnerability detection

Main principle

run the program in order to detect “unsecure behaviors”
(from simple crashes to complex security property violations)

Several ways to find “good” input values

black-box vs white-box fuzzing, public vs unknown input format, etc.

- (pseudo)-random values, (pseudo)-random mutations of given inputs
- human expertise, (non) typical use-cases
- code or input space coverage techniques
- goal oriented input selection:
  - target critical functionnalities or suspicious pieces of code
  - try to invalidate code assertions or security properties
  - etc.

In the following

A quick tour on . . .

"the most commonly used fuzzing techniques for vulnerability detection"

- random fuzzing
- grammar based fuzzing
- genetic based fuzzing (with an overview on AFL)
- *smart fuzzing*, or symbolic and dynamic-symbolic execution

Random (or brute-force or blind) fuzzing

```
random_fuzzing (pgm P) {
   while (true) {
     create a random input i
// either from scratch or randomly mutating an existing one
     run P with input i
     if the execution "succeeds"
          (i.e., crash, security breach, etc.)
         store the input i
   }
}
```

Random (or brute-force or blind) fuzzing

```
random_fuzzing (pgm P) {
   while (true) {
     create a random input i
// either from scratch or randomly mutating an existing one
     run P with input i
     if the execution "succeeds"
          (i.e., crash, security breach, etc.)
         store the input i
   }
}
```

Pros:

- very efficient generation scheme !
- no initial knowledge required
- pure black-box

Cons:

- no control over the execution sequences produced ...
- easily stuck by checksums, robust parsers, etc.

Grammar-based fuzzing

Drive the input generation using a grammar `G` of the nominal pgm input
(to ensure that these input won't be immediately rejected ...)

```
grammar_based_fuzzing (pgm P, grammar G) {
   while (true) {
   create a random input i belonging to L(G)
       run P with input i
       if the execution "succeeds"
          (i.e., crash, security breach, etc.)
         store the the input i
   }
}
```

Grammar-based fuzzing

Drive the input generation using a grammar `G` of the nominal pgm input
(to ensure that these input won't be immediately rejected ...)

```
grammar_based_fuzzing (pgm P, grammar G) {
   while (true) {
   create a random input i belonging to L(G)
       run P with input i
       if the execution "succeeds"
          (i.e., crash, security breach, etc.)
         store the the input i
   }
}
```

Pros:

- may cover complex input domains (file format, protocol)
- may overcome checksums and first-level parsing barriers

Cons:

- required some knowldge about the nominal pgm inputs
  (publicly available, reverse-engineering, learning, . . .)
- how much "unexpected" are the input produced ?

Genetic-based fuzzing

Use a fitness function to measure execution “relevance”

```
genetic_fuzzing (pgm P, input set Init) {
   CIS = Init /* Current (finite) Input Set */
   while (true) {
      randomly mutate/combine some inputs of CIS
      for each i of CIS
      run P with input i and compute its "score"
      if the execution "succeeds"
       store the the input i
   update CIS with the highest score inputs
  }
}
```

Genetic-based fuzzing

Use a fitness function to measure execution "relevance"

```
genetic_fuzzing (pgm P, input set Init) {
   CIS = Init /* Current (finite) Input Set */
   while (true) {
      randomly mutate/combine some inputs of CIS
      for each i of CIS
      run P with input i and compute its "score"
      if the execution "succeeds"
       store the the input i
   update CIS with the highest score inputs
  }
}
```

Pros:

- a mix between random and controled fuzzing
- still an efficient generation scheme

Cons:

- needs to design a good fitness function w.rt. the intended objective (coverage, pattern oriented, property oriented, etc.)
- some code instrumention usually required (for the fitness function)
- may still be stuck by checksums, robust parsers, etc. (local maximum of fitness function)

More details on basic fuzzing techniques

see D. Song slides . . .

Outline

Fuzzing (or how to *cheaply* produce *useful* program inputs)

A concrete fuzzer example: AFL (with a short demo)

Making the fuzzing smarter: (Dynamic) Symbolic Execution

Conclusion

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, Golang, RUST, OCaml, ...
- (any) binary code (with QEMU)

governing principles

- speed
- reliability
- ease-of-use
- availabililty and code sharing . . .

`lcamtuf.coredump.cx/afl/`

Fuzzing algorithm

branch coverage-oriented mutation-based fuzzing

Repeat until a time budget is reached:

1. pick an input from a queue
2. mutate it
3. run it
4. if "coverage increases" put the new input in the queue

Detailed algo:
`https://www.comp.nus.edu.sg/~mboehme/paper/CCS16.pdf`

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:

```
     // identifies the current basic block
cur_location = <compile-time-random-value> ;
     // mark (and count) a tuple hit
sh_mem[cur_location ^ prev_location]++ ;
    // 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 pahs ...

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
  $\rightarrow$ 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
  $\rightarrow$ reduce their size to speed-up fuzzing
  e.g., remove the size of variable lengths blocks

$\Rightarrow$ favorite seed = fastest and smallest input execersizing a tuple

Mutation strategy

no relationships between mutations and program states

- deterministic (sequentially):
  - flip bits (<> lengths)
  - add/substract small integers
  - insert known interesting integers (0, 1, INT_MAX, etc.)
- non deterministic:
  insertion, deletion, arithmetics, etc.

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

Crash unicity

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

AFL

a crach 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 . . .

Outline

Fuzzing (or how to *cheaply* produce *useful* program inputs)

A concrete fuzzer example: AFL (with a short demo)

Making the fuzzing smarter: (Dynamic) Symbolic Execution

Conclusion

Hunting in the corner cases

Random/Grammar/Genetic fuzzing techniques not always efficient enough to find “good” test inputs ?

Example: which input allow to activate the vulnerability(ies) below ?

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
  // assert (x+10 != 0)
  int *t = (int *) malloc((x+10) * sizeof(int)) ;
  z = twice(y);
  if (x == z) {
         // assert (y <= x +10) ;
         // assert (y > 0) ;
         t[y] = 0 ;
    }
  }
```

Hunting in the corner cases

Random/Grammar/Genetic fuzzing techniques not always efficient enough to find “good” test inputs ?

Example: which input allow to activate the vulnerability(ies) below ?

```
int twice(int v) {
    return 2 * v;
}

void test(int x, int y) {
  // assert (x+10 != 0)
  int *t = (int *) malloc((x+10) * sizeof(int)) ;
  z = twice(y);
  if (x == z) {
         // assert (y <= x +10) ;
         // assert (y > 0) ;
         t[y] = 0 ;
    }
  }
```

A random-based search may not succeed . . .
Is it possible to improve the technique ?

$\Rightarrow$ An (old !) answer: symbolic execution . . .

Symbolic Excecution

King, 76

Objective:

run a program paths (as in test execution) but mapping variables to symbolic values (instead of **concrete ones**)

- each symbolic execution allows to reason on a set of concrete executions (all the ones following **the same path** in the CFG)
- allow to decide if a CFG path is feasable or not (and with wich input values ?)
- allow to explore a **(finite !)** set of paths in the CFG . . .

Symbolic Excecution

King, 76

Objective:

run a program paths (as in test execution) but mapping variables to symbolic values (instead of **concrete ones**)

- each symbolic execution allows to reason on a set of concrete executions (all the ones following **the same path** in the CFG)
- allow to decide if a CFG path is feasable or not (and with wich input values ?)
- allow to explore a **(finite !)** set of paths in the CFG ...

Principle:

Associate a path predicate φ_σ to each path σ of the CFG:

$$(\exists \text{ a variable valuation } v \text{ s.t } v \models \varphi_\sigma) \Leftrightarrow (v \text{ covers } \sigma)$$

(φ_σ is the conjunction of all boolean conditions associated to σ in the CFG)

- solving φ_σ indicates if σ is feasible
- iterate over a (finite) subset of the CFG paths ...

In practice: express φ_σ in a decidable logic fragment (e.g., SMT).

More on Symbolic Execution . . .

- application to the previous example
- what can we do if:
  - the **path predicate** cannot be expressed in a decidable logic ?
    (e.g., non linear operations)
  - the program contains conditions on non-reversible functions ?
    (e.g., `if (x == hash(y)) ...`)
  - part of the program code is not available
    (e.g., library functions, `if (!strcmp(s1, s2) ...`)

$\rightarrow$ combine symbolic and concrete executions:
concolic execution (or Dynamic Symbolic Execution)

More on Symbolic Execution . . .

- application to the previous example
- what can we do if:
  - the **path predicate** cannot be expressed in a decidable logic ?
    (e.g., non linear operations)
  - the program contains conditions on non-reversible functions ?
    (e.g., `if (x == hash(y)) ...`)
  - part of the program code is not available
    (e.g., library functions, `if (!strcmp(s1, s2) ...`)

$\rightarrow$ combine symbolic and concrete executions:
concolic execution (or Dynamic Symbolic Execution)

$\Rightarrow$ Trade-off between:

- tractability: keep decidable decision procedures over path predicates
- scalabilty: concrete execution faster than symbolic reasonning
- completness: concretization $\Rightarrow$ loss of execution paths

see that on Martin Vechev's slides . . .

DSE for vunlnerability 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.

$\Rightarrow$ widely used in vuln. detection and exploitability analysis)

DSE for vunlnerability 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.

$\Rightarrow$ widely used in vuln. detection and exploitability analysis)

- numerous existing tools . . .
  - source-level: Klee(C/C++), JPF (Java), etc.
  - binary-level: Sage, Mayhem, Angr, BinSec, Triton, etc.

DSE for vunlnerability 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.

$\Rightarrow$ widely used in vuln. detection and exploitability analysis)

- numerous existing tools ...
  - source-level: Klee(C/C++), JPF (Java), etc.
  - binary-level: Sage, Mayhem, Angr, BinSec, Triton, etc.
- however, not all problems solved (yet ?), e.g.:
  - "path explosion" problem on large codes
  - can be rather slow (compared with *fuzzing*)

How to get more from fuzzing ?

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

[1] as long as instrumentation is feasable, see later

How to get more from fuzzing ?

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

$\Rightarrow$ rather well adapted for security analysis / vulnerability detection

---

[1] as long as instrumentation is feasable, see later

How to get more from fuzzing ?

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 feasable, see later

Outline

Fuzzing (or how to *cheaply* produce *useful* program inputs)

A concrete fuzzer example: AFL (with a short demo)

Making the fuzzing smarter: (Dynamic) Symbolic Execution

Conclusion

An effective vulnerability detection technique

(certainly still one of the most effective !)

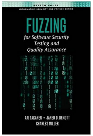
Why ?

- An "easy to go" approach: don't (always) need the source, dont (always) even need to disassemble just need to "execute" (or simply to emulate)
  $\rightarrow$ can be often implemented in a few lines of Python ...
- Cover a potentially large spectrum, e.g.,
  - AFL: fast, but detect superficial/shallow bugs only
  - DSE: slow but can find deep vulnerabilities

However

- never give you a “vulnerability free” stamp
  (but may provide you with concrete "vulnerable inputs")
- could be limited by some dynamic code protection techniques

Still a promising R&D direction . . .



A huge number of available tools, covering:

- many fuzzing techniques
- many application domains (web, protocols, file processors, OS, etc.)

Metrics to evaluate a fuzzing technique/tool

- *effectiveness:* ratio execution time vs relevance
- ability to re-execute (faulty) tests, test minimization
- feedback produced (beyond "segmentation faults")
  $\rightarrow$ exploitability indications ?

$\Rightarrow$ numerous new challenges to come:

- application domains: embedded systems, IoT, industrial systems, . . .
- (combination with other techniques: static analysis, IA, etc.

Have a look to P. Godefroid paper and **3mn video** (links on the course webpage)