Fuzz testing
(“fuzzing”)

Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

Fuzzing

- **What is fuzzing ?**
  - Testing software with invalid and possibly malicious data, usually generated in semi-automatic manner.
- **What is the goal of fuzzing?**
  - Evaluate program response to invalid input, rather than “common case” inputs used for plain functional testing.
- **Optimal response to invalid inputs:**
  - a grafecul failure — in line with the “Fail Safely” design principle. Nothing “unintended” or “bad” happens!
- **Vulnerable responses to invalid input** may include (possibly a combination of):
  - program crashes, memory corruption (e.g. buffer overflows). failure to detect the error in input

Fuzz testing

- **Deriving inputs — essential techniques**
  - **Randomisation**: generate random inputs, or introduze randomness during generation:
  - **Mutation**: mutate given inputs according to some criteria
  - **Grammar-based generation**: use a grammar to generate inputs
  - Hybrid approaches combining these are common.
- **Fuzz-testing process**
  - **Black-box**: generate inputs and monitor execution result, blindly.
  - **White-box**: guide input generation according to feedback from execution + information regarding program structure.

Random input

```
$ head -c 15 /dev/urandom | xargs ping
ping: cannot resolve ?c?D?\fN\016?=?;?: Unknown host
```

- No context of the software at stake or the type of input.
- Easy to implement, but will typically expose only shallow bugs

Mutation-based input generation

- Start from valid inputs e.g. inputs for normal functional testing or concrete execution.
- Mutate them according to some strategy for instance:
  - Applying randomisation, e.g., random bit flips.
  - More generally, applying mutation rules
  - Mutation fragments may be domain-specific, e.g., contain shell-code, SQL injection, etc.
- Ability to expose bugs: dependent on starting inputs and mutation expressiveness for the context at stake.
- Example tools next:
  - radamsa
  - The ZAP fuzzer
  - zzuf

Example tools — radamsa

```
$ echo 192.168.106.103 | radamsa --count 10 --seed 0
-107.167.106.103
192.168.8407971865571866.-9?5154737306362663942413194069
191.1A1.1A1.106.1
192.129.18.106.103
192.168.0.103
192.170141183460.106.1802311213346089.104
-3402823669209.106.168.106.16.103
192093846346337460765704.192.65704.-1.?-18446744073709518847
192.106.0
191.168.106.103
$ echo 192.168.106.103 | radamsa --count 1 --seed 0 | xargs ping
ping: invalid option -- 1
```

- Radamsa: a mutation-based input generator
- Mutates given inputs, randomly applying pre-defined mutation rules and patterns.

Example tools — radamsa (2)

```
$ ./radamsa --list
Mutations (-m)
  ...
  bd: drop a byte
  bf: flip one bit
  bi: insert a random byte
  ...
  sr: repeat a sequence of bytes
  sd: delete a sequence of bytes
  ld: delete a line
  ...
  ls: swap two lines
  ...
  num: try to modify a textual number
  xp: try to parse XML and mutate it
  ...
Mutation patterns (-p)
  od: Mutate once
  nd: Mutate possibly many times
  bu: Make several mutations closeby once
```

- Example mutations and mutation patterns (listed with `radamsa --list`)

ZAP fuzzer

Select part of the input to “fuzz with”, in this case the “1” value that is part of the HTTP request header

```
GET
http://localhost:8081/vulnerabilities/sqli/?id=1&Submit=Submi
t HTTP/1.1
Proxy-Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.
...l+xml,applicati...
...nerabilities/sq...
```

Select “fuzz set” of replacements for the chosen input, in this case strings likely to trigger SQLi, if a vulnerability of this kind exists

Files: [ ] Find Next Find Previ

- [x] Active SQL Injection
- [x] MS SQL Injection i
- [x] MS SQL Ninja Injection (Blind)
- [x] MySQL Injection (Blind)
- [x] MySQL Injection 101
- [x] MySQL/MS SQL Common Injection
- [x] Oracle SQL Injection
- [x] Passive SQL Injection
- [x] SQL Injection
- [ ] URI Exploits
- [ ] User Agents

Several test cases will be considered for execution, each replacing ‘1’ by potentially malicious input

State	Payloads
Reflected	' or '1'='1
Reflected	' or '1'='1
Reflected	' or '7659'='7659
Reflected	' or '7659'='7659
Reflected	' or 'a'='a
Reflected	' or 1=1--
Reflected	1;(load_file(char...
Reflected	1' or '1'='1
Reflected	1

Example programs - zzuf

```
zzuf -r 0.02 -s 1:3 cat ./silly_program.c

J'a|cl}de <st?i?.h>

inu`main(int avgc, char*? argw) {
   int l = 0;
  whidE("fgfgets*buf,sizeof(Buf-, f) != NULL- {
    pryntf(btf?;
  }  dclose(f);
  retezn 0;J}

#include |stdio.h

i|t main(int aRfc, ch`r** argv) {
  ahar buf[128};
```

- zzuf automates the fuzzing process by **transparently fuzzing read from files or from the network.**
    - Mutations are introduced randomly according to a specifed bit fuzzing ratio.
    - The target program runs in batch mode for a specified number of trials / seeds.
    - It has been sucessfull in uncovering bugs in real-world programs.

Example programs - zzuf (2)

- In this case zzuf transparently mutates data from the network (use of the **-n** switch).

```
$ zzuf -r 0.02 -s 1 -n curl http://www.dcc.fc.up.pt/~edrdo/QSES1819/test_zzuf.html
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   328    0   328    0     0     60      0 --:--:--  0:00:05 --:--:--     0
HT?P'1.1 200 OK
D?te: Wmd, 1"dec 2018 1=;42:36 GMt
fips PHP/54*1>?2.4.6"(CentO[)0OrenSSL/1.0.k
L?st/Modif?ed: WeD, 12 Dec 0q8$!5:40:54 GMT
Etag: "07-57bd?86197e5a"
Acce`t-Ranges: bxtus
ConteNt-Lmngth: 71
Cltent-Type: |ext.html

8html>?<rody>

ZZUF!|est(resource -- QSS 0018/2019

</body>
       </html>
```

"Fuzzed" execution

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    71  100    71    0     0    220      0 --:--:-- --:--:-- --:--:--  1145
<html>
<body>

ZZUF test resource -- QSES 2018/2019

</body>
</html>
```

Normal execution

Grammar-based input generation

- Generate inputs using a grammar.
  - Grammar rules may express possible deviations.
  - Combination with mutation: alternatively, valid inputs may be generated using a grammar, and then mutated.
  - This approach can be more systematic, is potentially able to generate more relevant inputs, and account for complex combinations of input fragments.
- Example tool illustrated next: **blab**
  - A few others of the same kind: **ABNFfuzzer** **gramfuzz**

Example tools - blab

ip_address.blab

```
output = ip_address "\n"
ip_address = octet "." octet "." octet "." octet
octet = [0-9] | [1-9][0-9] | “1” [0-9][0-9] | “2” [0-4][0-9] | “25” [0-5]
```

```
$ blab ip_address.blab -n 10 –s 0
4.4.4.104
5.148.205.94
0.237.230.95
0.140.232.252
178.81.250.6
252.252.252.8
135.159.123.250
204.5.172.8
177.188.21.213
0.78.204.240
```

- Blab: a grammar-based black-box fuzzer
- Inputs generated according to grammar. In this example the grammar generates only valid IP addresses.

Example tools - blab (2)

fuzzed_ip_address.blab

```
output = fuzzed_ip_address "\n"
fuzzed_ip_address = octet "." octet "." octet "." octet
octet = normal_octet | fuzzed_octet
normal_octet = [0-9] | [1-9][0-9] | “1” [0-9][0-9] | “2” [0-4][0-9] | “25” [0-5]
fuzzed_octet = [0-9]{3}
```

```
$ blab fuzzed_ip_address.blab -n 10 -s 0
40.4.40.40
143.696.528.100
137.013.61.242
7.433.5.522
113.277.743.145
123.6.119.235
740.810.87.801
221.077.43.319
079.737.507.518
947.479.245.947
```

- In this variation we allow the possibility of malformed IP IP addresses.

Generate, then mutate

```
$ blab fuzzed_ip_address.blab -n 5 -s 0 | tee generated.txt
40.4.40.40
143.696.528.100
137.013.61.242
7.433.5.522
113.277.743.145
$ radamsa --count 1 --seed 22 generated.txt -p nd=10
3321759348573678331568.4.40.40
143.696.528.100
1.013.61.0
7.65535.9223372036854775803.522
113.280.743.145
```

- Generation and mutation can be combined, e.g., blab + radamsa.

Black-box fuzzing

- **Simplest approach — "black box" fuzzing**
  - Repeatedly feed the program with fuzzed inputs, without consideration for the program structure.
  - Observe program responses and assert that program fails gracefully / nothing "bad" happens (crashes, memory corruption etc).
- **Looking for bugs — possible strategies**
  - Instrument the program with runtime sanitizers to monitor abnormal program execution (undefined behavior, buffer overflows, etc)
  - Inspect exit codes (e.g. SIGSEV = 139 — segmentation fault), program output, etc

White-box fuzzing

- **Idea**
  - Monitor (instrumented) program state during execution and observe which changes to input cause new program states to be explored.
  - The information is used to generate new inputs, trying to avoid inputs that repeat the same program paths.
- The goal is to explore the state-space of the program as extensively as possible / increase code coverage.
  - The execution is automatic, but can be time-consuming given that many executions of the program under test will be triggered.
  - Tools can derive inputs randomly or (with better results) through mutations of a pre-defined set of inputs that are accepted by the program.
- Example tools:
  - **AFL**, **libFuzzer**, **SAGE**

libFuzzer / AFL

- libFuzzer, AFL
  - The fuzzers employed by Google's OSS-Fuzz project ("continuous fuzzing of open source software")
  - Employ program instrumentation/monitoring coupled with input mutation techniques that are coverage-guided.
  - The fuzzers are effective if supplied with a corpus of input samples that are representative of the program execution / likely to provide good coverage.

libFuzzer example

```
pwm_res_t pwm_hash_password(salt_t salt, char* password, hash_t checksum) {
  MD5_CTX ctx;
  MD5Init(&ctx);
  MD5Update(&ctx, salt, sizeof(salt_t));
  MD5Update(&ctx, (unsigned char*) password, 2 + strlen(password));
  MD5Final(checksum, &ctx);
  return PWM_OK;
}
```

Crashing PWM command

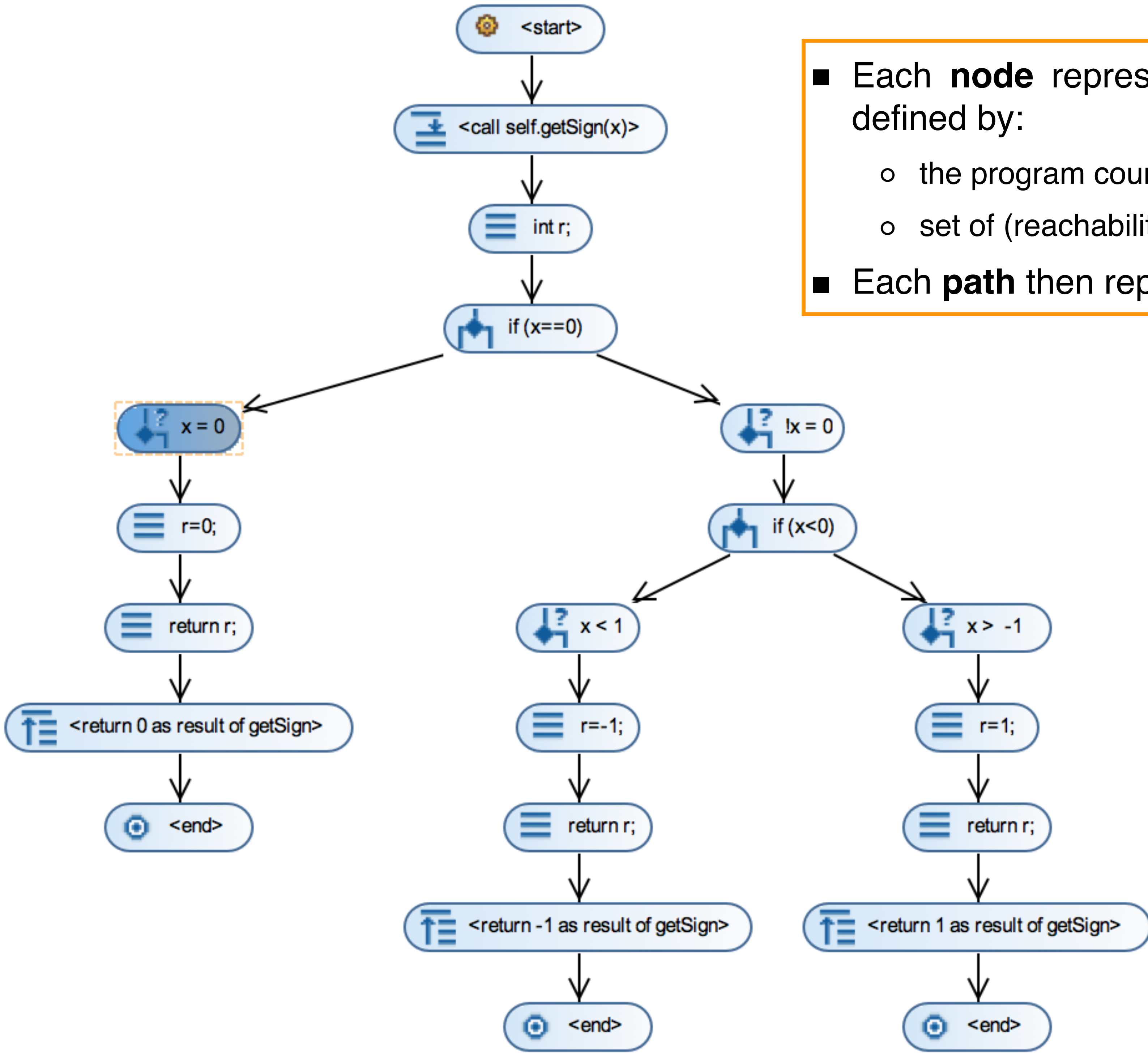
```
open password.txt Qs????????????????????????????????????????????????????
lllllllllllll??????????????????????????????????????????????????
llllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllll
llllllllllllllllllll???????????????es181
```

- Base code: a version of PWM from project 2.
- Let us introduce a bug in **`pwm_hash_password`** `shown above.`
- Sample execution: from an initial corpus of 2 input examples, libFuzzer finds the bug after one hour, generating 402 test cases along the way.

SAGE & symbolic execution

- SAGE employs **symbolic execution**.
  - Interprets a program, treating inputs as symbolic with possible constraints — actual values need not be specified for input values.
  - When a branch condition is found that depends on symbolics input, follow each branch leading to a symbolic execution tree. User-specified assertions can be checked for all possible executions.
  - May potentially explore all possible states of a program, in most cases the state-explosion problem must be curbed through state-exploration strategies.
  - A few other tools of the genre: Klee, Triton, S2E

Symbolic execution tree



- Each **node** represents a symbolic execution state and is defined by:
  - the program counter (PC)
  - set of (reachability) conditions over the symbolic inputs
- Each **path** then represents a possible execution

```
int getSign(int x) {
  int r ;
  if (x == 0)
    r = 0;
  else if (x < 0)
    r = -1;
  else
    r = 1;
  return r;
}
```

[screenshot obtained using the KeY Symbolic Execution Debugger]