Laborator II: Tehnici de Descoperire a Vulnerabilităților

Tabelă de Conținut

Vulnerabilități

Recapitulare

- Proces
- Executabil
- Vulnerabilitate
- Exploatarea executabilelor
- Suprafață de atac
- Vectori uzual de atac la executabile

Suprascrierea Stivei

- **Funcționare**: citirea unor date de lungime prea mare într-un buffer neîncăpător, alocat pe stivă
- **Impact**: suprascrierea unor variabile, a valorii vechi a stivei sau chiar a adresei de retur

```
char last_name[20];

printf ("Enter your last name: ");
scanf ("%s", last_name);
```

Suprascrierea Întregilor

- **Funcționare**: depășirea limitelor valorilor ce pot fi stocate într-un întreg, în cazul procesării (de exemplu, înmulțire cu altă valoare) acestuia
- **Impact**: suprascrieri de *buffers*, alte vulnerabilități logice

```
// Excerpt from OpenSSH 3.3
nresp = packet_get_int();
if (nresp > 0) {
  response = xmalloc(nresp*sizeof(char*));
  for (i = 0; i < nresp; i++) response[i] = packet_get_string(NULL);
}
```

Atacuri Cu Șirurilor de Formatare

- **Funcționare**: folosirea unor funcții specifice (de exemplu, `printf`, cu un șir de formatare provenit de la utilizator)
- **Impact**: suprascrieri de *buffers*, vizualizarea memoriei procesului

```
int main(int argc, char **argv){
  char buffer[128];
  char api_key[32];

  [...]

  snprintf(buffer, 128, argv[1]);
}
```


STILL WAITING


nu voi mai scrie cod vulnerabil
nu voi mai scrie cod vulnerabil
nu voi mai scrie cod vulnerabil
nu voi mai scrie cod vulnerabil
nu voi mai scrie cod vulnerabil
nu voi mai scrie cod
nu voi mai scrie cod
nu voi mai scrie cod
nu voi mai scrie cod

Tehnici de Descoperire a Vulnerabilităților

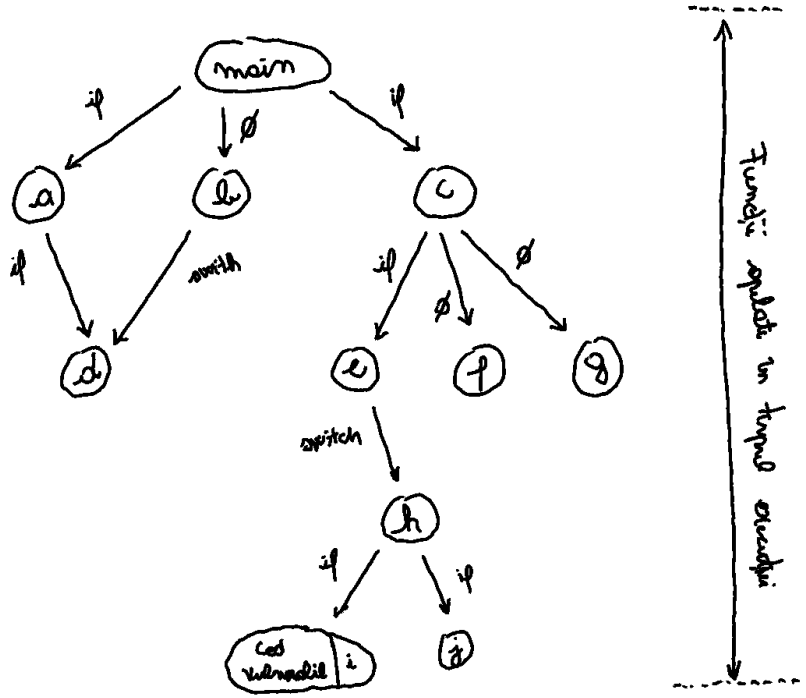
Tehnică de Descoperire a Vulnerabilităților

- **Tehnică de Descoperire a Vulnerabilităților**: Metodă prin care se folosesc secvențe de intrări (parte a suprafeței de atac) care, odată transmise procesului, acesta se comportă incorect.
- Sunt folosite:
  - Extern de atacatori și bug bounty hunters
  - Intern, de cercetători de securitate

Fuzzing

- Reprezintă generarea (cu ajutorul unui algoritm) de intrări ce vor fi oferite unui proces.
- Rezultatul este o parcurgere în lățime a grafului format din secvențele de instrucțiuni executate (engl. *control-flow graph*).

Parcurgerea CFG la Fuzzing



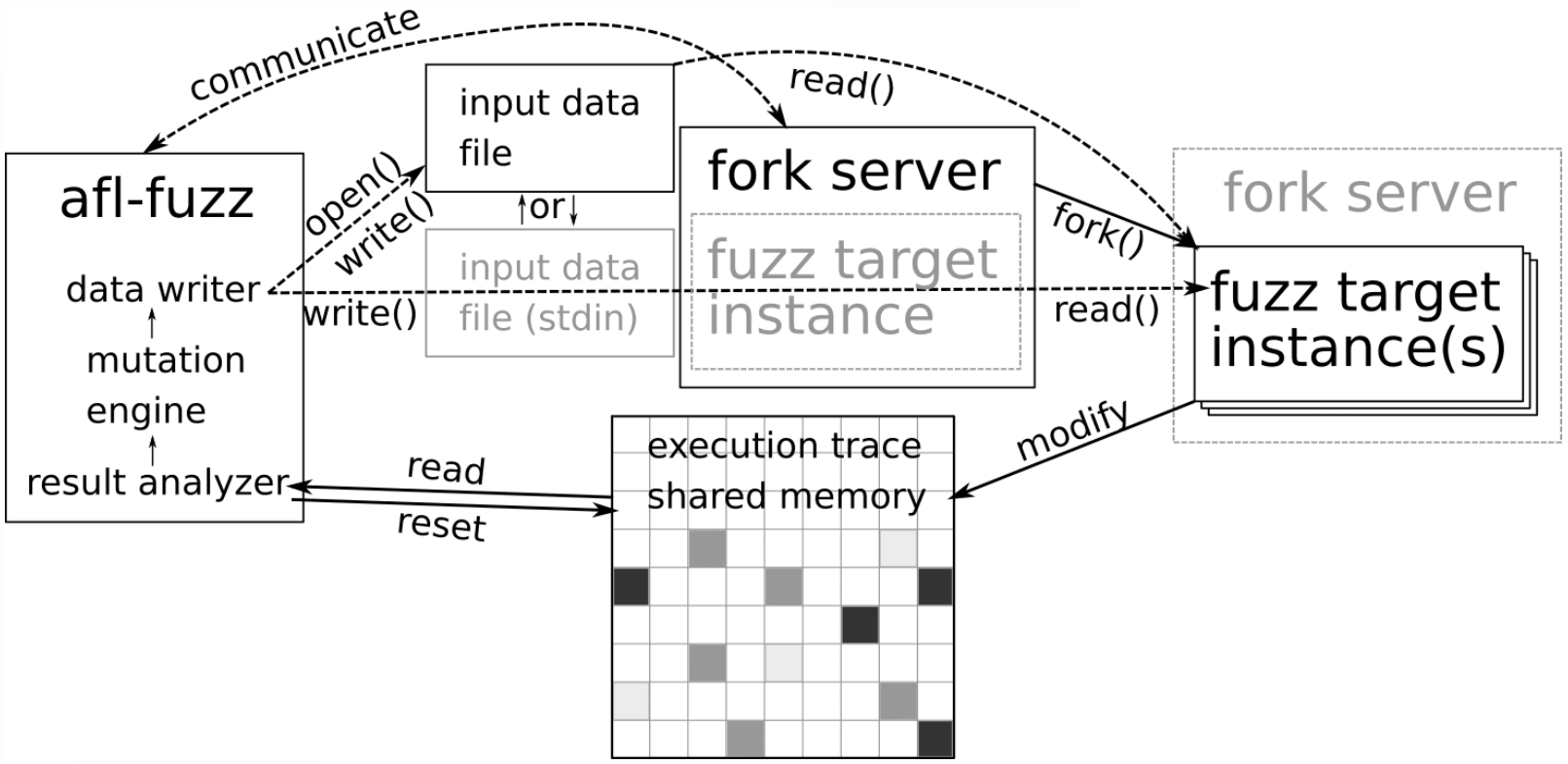
Pentru ca fuzzing-ul să găsească o vulnerabilitate în funcția i(), calea main() → c() → e() → h() → i() trebuie găsită. ⇒

Intrarea generată aleator trebuie să respecte condițiile în cele 4 cărora probabilitate individuală e $p_i \in (0,1]$ ⇒

$P_{găsire\ vuln.} = \prod_{i \in \{main, c, e, h\}} p_i$ (mică în valoare) ⇒

Fuzzing-ul va găsi mai rapid vulnerabilități din funcțiile superioare ale CFG-ului.

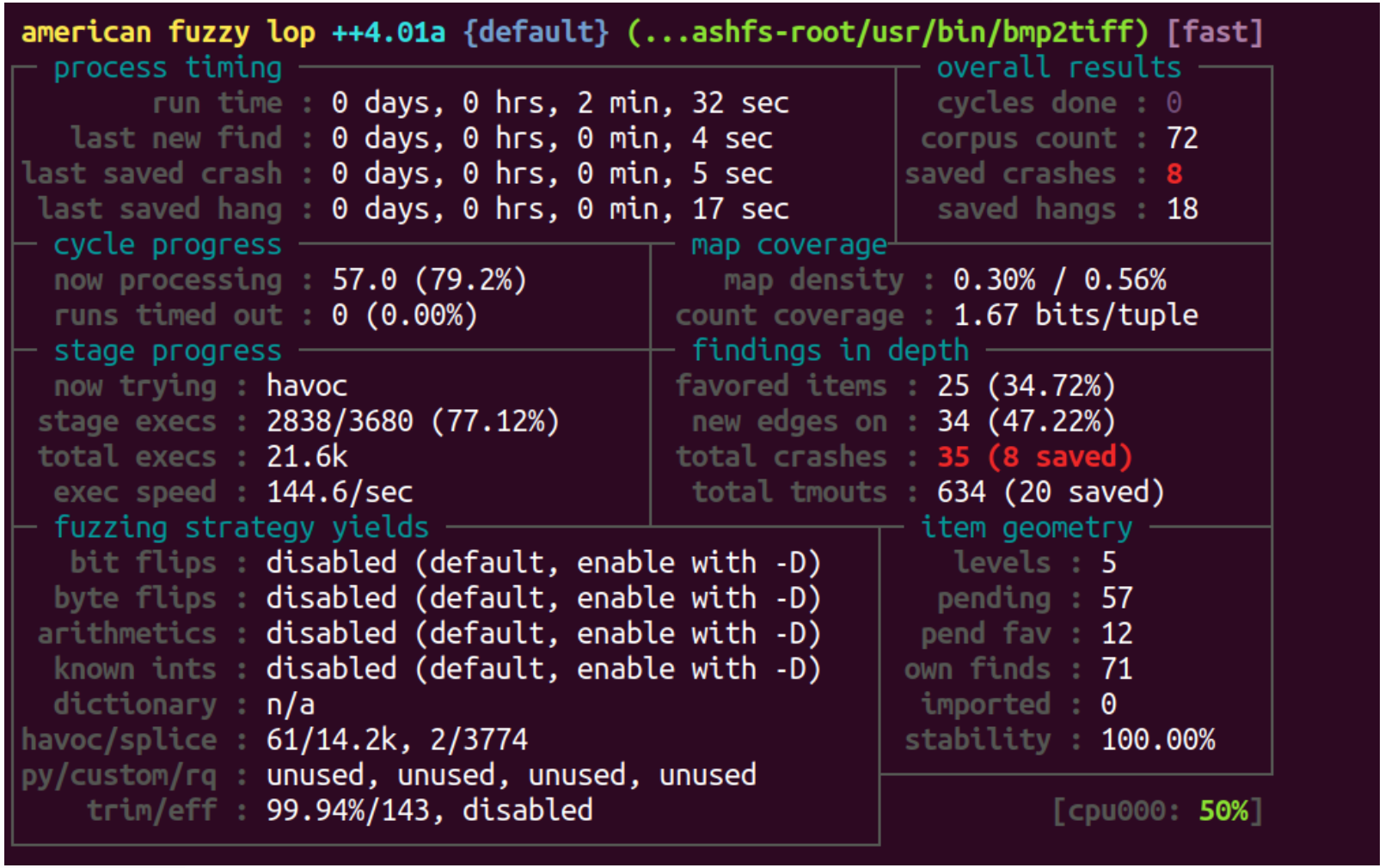
Arhitectura unui *Fuzzer*



Tipuri de *Fuzzers*

- Cunoașterea codului sursă: *blackbox*, *whitebox* și *graybox*
- Cunoașterea formatului de intrare: *smart* și *dumb*
- Cunoașterea stării programului: *stateless* și *stateful*
- În funcție de suprafața de atac: pentru GUI, de rețea, de fișiere etc.

AFL++


american fuzzy lop ++4.01a {default} (...ashfs-root/usr/bin/bmp2tiff) [fast]
process timing
run time : 0 days, 0 hrs, 2 min, 32 sec
last new find : 0 days, 0 hrs, 0 min, 4 sec
last saved crash : 0 days, 0 hrs, 0 min, 5 sec
last saved hang : 0 days, 0 hrs, 0 min, 17 sec
overall results
cycles done : 0
corpus count : 72
saved crashes : 8
saved hangs : 18
cycle progress
now processing : 57.0 (79.2%)
runs timed out : 0 (0.00%)
map coverage
map density : 0.30% / 0.56%
count coverage : 1.67 bits/tuple
stage progress
now trying : havoc
stage execs : 2838/3680 (77.12%)
total execs : 21.6k
exec speed : 144.6/sec
findings in depth
favored items : 25 (34.72%)
new edges on : 34 (47.22%)
total crashes : 35 (8 saved)
total tmouts : 634 (20 saved)
fuzzing strategy yields
bit flips : disabled (default, enable with -D)
byte flips : disabled (default, enable with -D)
arithmetics : disabled (default, enable with -D)
known ints : disabled (default, enable with -D)
dictionary : n/a
havoc/splice : 61/14.2k, 2/3774
py/custom/rq : unused, unused, unused, unused
trim/eff : 99.94%/143, disabled
item geometry
levels : 5
pending : 57
pend fav : 12
own finds : 71
imported : 0
stability : 100.00%
[cpu000: 50%]

CVE-uri Descoperite cu AFL++

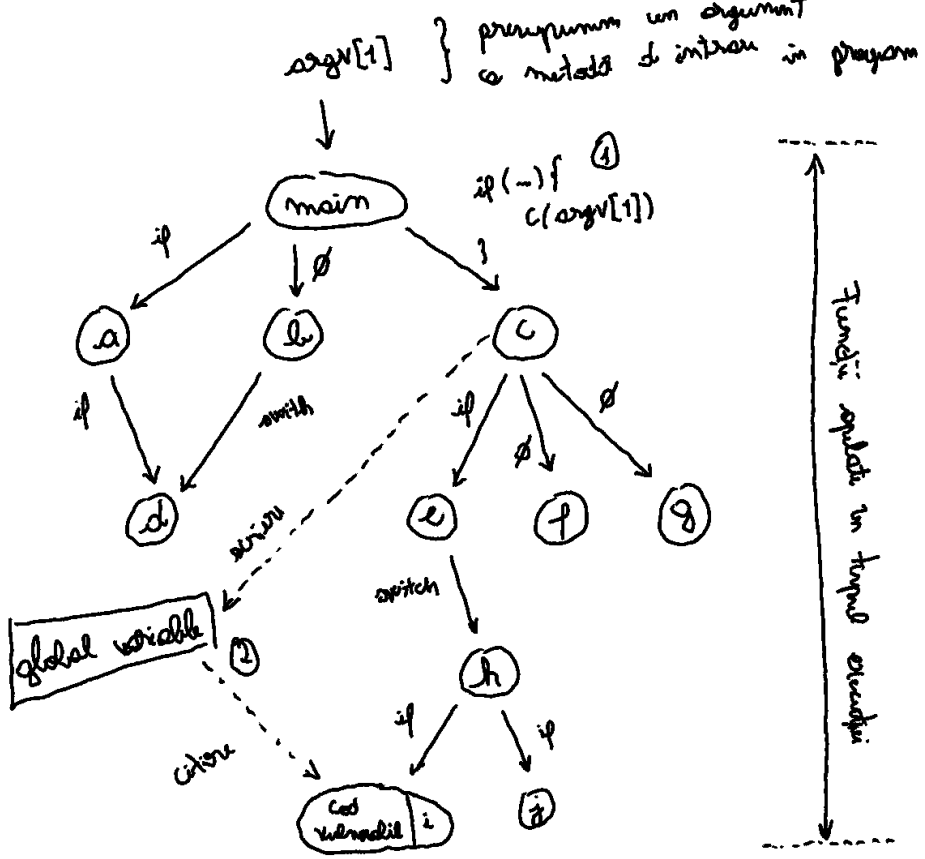
Trophies

- VLC
  - CVE-2019-14437 CVE-2019-14438 CVE-2019-14498 CVE-2019-14533 CVE-2019-14534 CVE-2019-14535 CVE-2019-14776 CVE-2019-14777 CVE-2019-14778 CVE-2019-14779 CVE-2019-14970 by Antonio Morales (GitHub Security Lab)
- Sqlite
  - CVE-2019-16168 by Xingwei Lin (Ant-Financial Light-Year Security Lab)
- Vim
  - CVE-2019-20079 by Dhiraj (blog)
- Pure-FTPd
  - CVE-2019-20176 CVE-2020-9274 CVE-2020-9365 by Antonio Morales (GitHub Security Lab)
- Bftpd
  - CVE-2020-6162 CVE-2020-6835 by Antonio Morales (GitHub Security Lab)
- Tcpdump
  - CVE-2020-8036 by Reza Mirzazade
- ProFTPd
  - CVE-2020-9272 CVE-2020-9273 by Antonio Morales (GitHub Security Lab)
- Gifsicle
  - Issue 130 by Ashish Kunwar
- FFmpeg
  - Ticket 8592 Ticket 8593 Ticket 8594 Ticket 8596 by Andrea Fioraldi
  - Ticket 9099 by Qiuhao Li
- Glibc
  - Bug 25933 by David Mendenhall
- FreeRDP
  - CVE-2020-11095 CVE-2020-11096 CVE-2020-11097 CVE-2020-11098 CVE-2020-11099 CVE-2020-13397 CVE-2020-13398 CVE-2020-4030 CVE-2020-4031 CVE-2020-4032 CVE-2020-4033 by Antonio Morales (GitHub Security Lab)
- GNOME
  - Libxps issue 3 by Qiuhao Li
- QEMU
  - CVE-2020-29129 CVE-2020-29130 by Qiuhao Li
- GNU coreutils
  - Bug 1919775 by Qiuhao Li

Analiza Fluxului Datelor

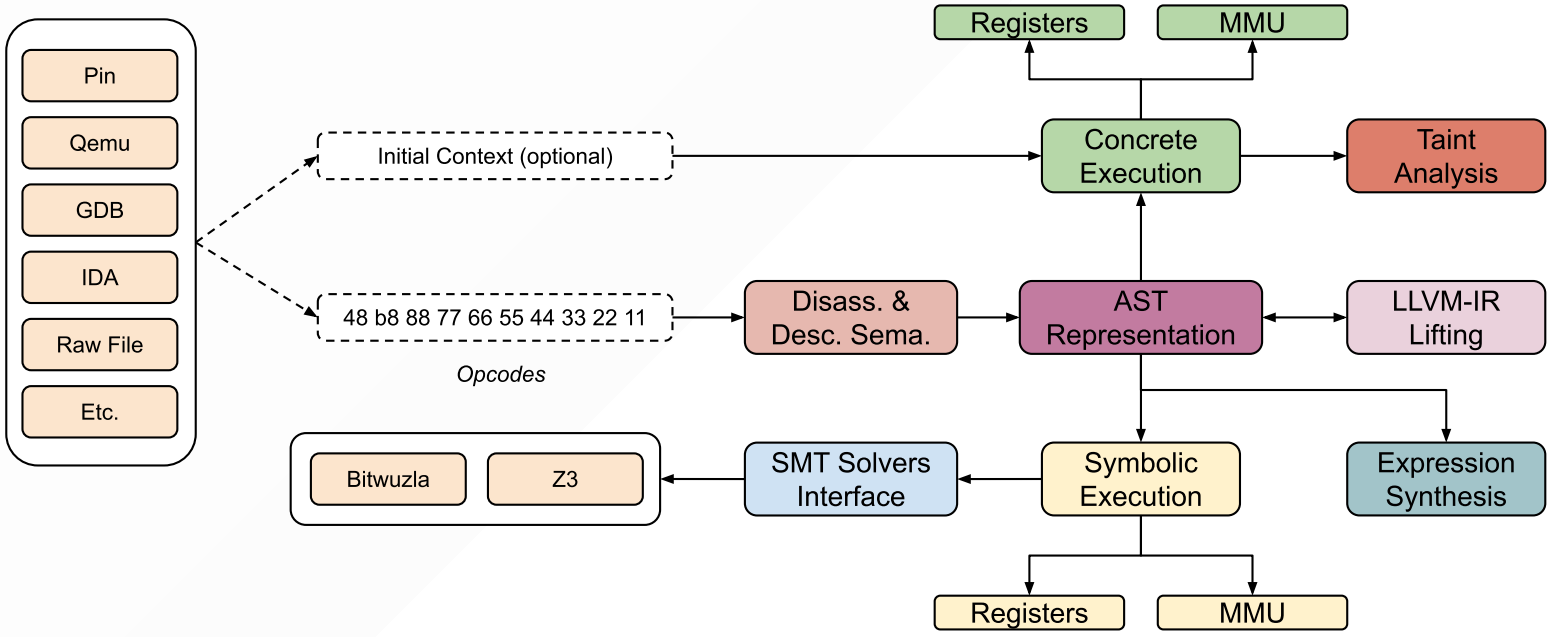
- În engleză, *taint analysis*
- Reprezintă procesul de descoperire a fluxului de date într-un proces.
- Se poate executa:
  - Static, analizând codul sursă al programului
  - Dinamic, instrumentând execuția și urmărind fluxul datelor în timp real.

Fluxul Datelor Preluate de la CFG



①, ② modalități de „curgere a datelor" în timpul execuției programului

Triton


Pin
Qemu
GDB
IDA
Raw File
Etc.
Initial Context (optional)
48 b8 88 77 66 55 44 33 22 11
Opcodes
Registers
MMU
Concrete Execution
Taint Analysis
Disass. & Desc. Sema.
AST Representation
LLVM-IR Lifting
Bitwuzla
Z3
SMT Solvers Interface
Symbolic Execution
Expression Synthesis
Registers
MMU

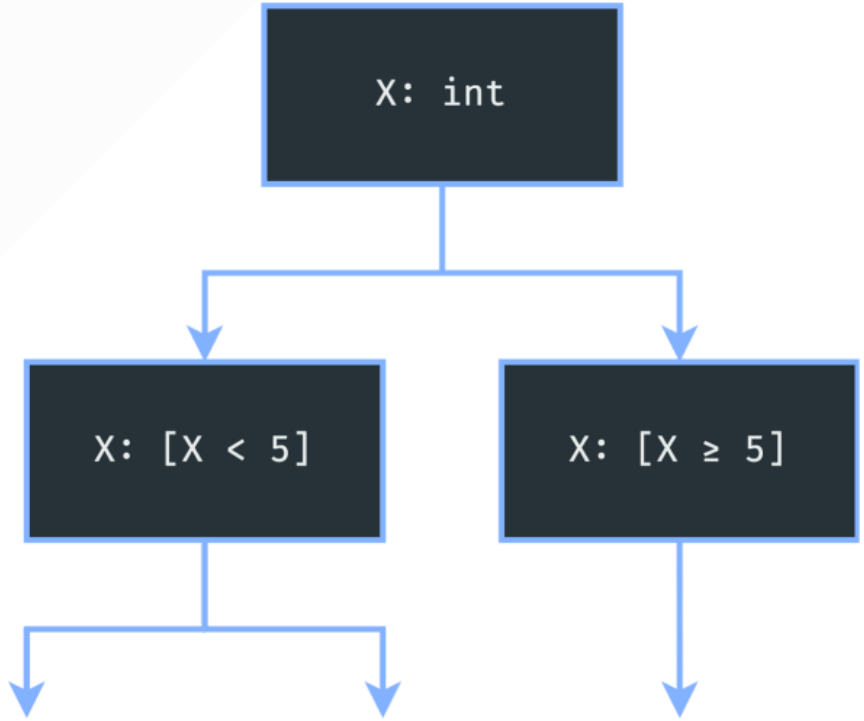
Analiza Fluxului Datelor cu Triton

```
triton::Context ctx;

ctx.setArchitecture(ARCH_X86_64);

ctx.taintRegister(ctx.registers.x86_ah);

// Execute until a chosen moment. Then stop the execution.

bool is_tainted = ctx.isRegisterTainted(ctx.registers.x86_rdx);
```

Execuție Simbolică

- Constă în utilizarea unor valori simbolice (față de cele concrete, folosite la *fuzzing*), pentru reprezentarea unor date de intrare.
- Menține căi pentru care programul a ajuns în anumite puncte, reținând în același timp și intrările corespunzătoare.
- Tipuri
  - *Offline*: Programul este rulat, se generează fișiere de *tracing*, iar analiza este efectuată pe ele.
  - *Online*: Analiza este efectuată în timpul rulării programului.

```
int x;
...
if (x < 5) {
    ①
} else {
    ②
}
```



angr

Open Source

Released as Free and Open Source Software under the permissive BSD license. Contributions are welcome.

Cross-Platform

Runs on Windows, macOS, and Linux. Built for Python 3.8+.

Symbolic Execution

Provides a powerful symbolic execution engine, constraint solving, and instrumentation.

Control-Flow Graph Recovery

Provides advanced analysis techniques for control-flow graph recovery.

Disassembly & Lifting

Provides convenient methods to disassemble code and lift to an intermediate language.

Decompilation

Decompile machine code to angr Intermediate Language (AIL) and C pseudocode.

Architecture Support

Supports analysis of several CPU architectures, loading from several executable formats.

Extensibility

Provides powerful extensibility for analyses, architectures, platforms, exploration techniques, hooks, and more.

Cod Analizat cu angr

```
char *sneaky = "SOSNEAKY";

int authenticate(char *username, char *password){
  if (strcmp(password, sneaky) == 0) return 1;

  // Check the password stored into a shadow file.
}
```

Execuția Simbolică Folosind angr

```
proj = angr.Project('fauxware', auto_load_libs=False)
state = p.factory.entry_state()
manager = proj.factory.simulation_manager(state)

# Symbolically execute the program until we reach a branch
# statement for which both branches are satisfiable.
manager.run(until=lambda sm_: len(sm_.active) > 1)

input_0 = manager.active[0].posix.dumps(0)
input_1 = manager.active[1].posix.dumps(0)

[...]
```

Revizuirea Codului Sursă

- Constă într-un proces de analiză a codului sursă.
- Tipuri
  - Manual, folosind, de exemplu, experiența unui senior
  - Automat, folosind instrumente de analiză
- Exemple de categorii de instrumente
  - *Linters*
  - Interogatoare de cod

Interogarea Codului cu Joern

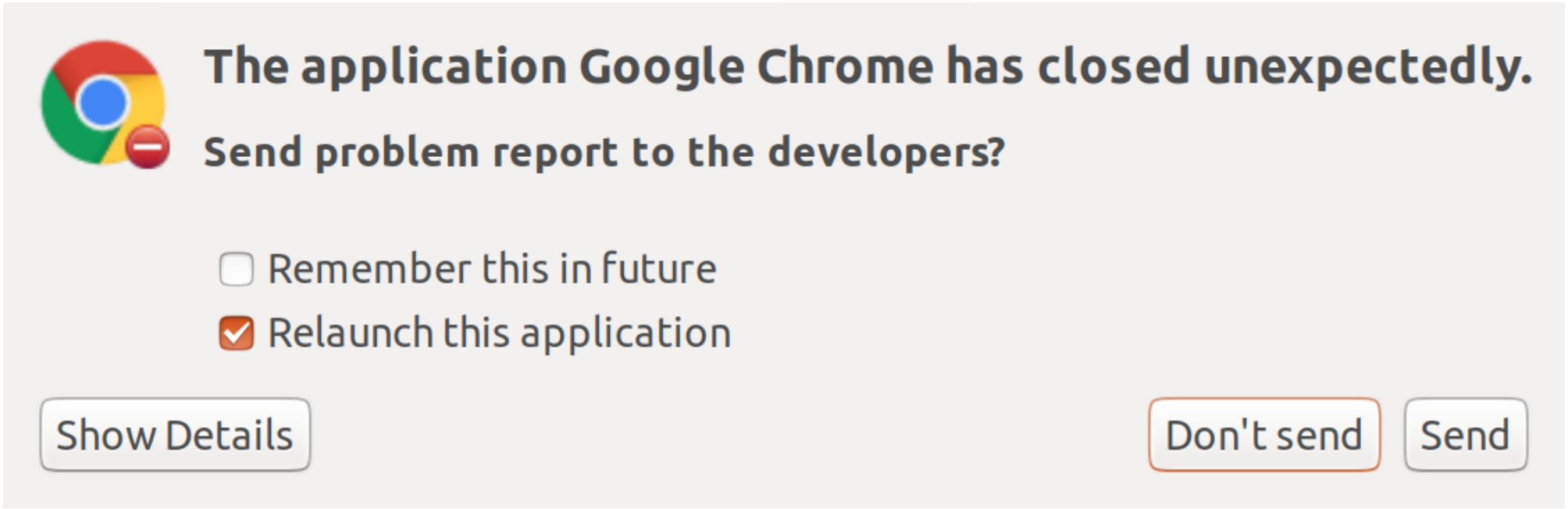
Următoarea interogare CPGQL

```
({
  cpg.method("(?i)printf").callIn
    .whereNot(_.argument.order(1).isLiteral)
}).l
```

va detecta vulnerabilitatea de mai jos

```
printf(argv[1], 4242);
```

Analiza *Crash*-urilor



Exerciții

Recomandări

- Folosiți comanda `man` pentru a primi ajutor la rularea anumitor comenzi.
- Folosiți documentația pwntools pentru a identifica metodele de care aveți nevoie.


I'M READY
memegenerator.net