Dynamically Proving That Security Issues Exist

Andrew V. Jones, Vector Software, Inc.

NIST SwMM-RSV, July 2016

Focus of this talk

Chess & McGraw'04[1]:

> *Good static checkers can help spot and eradicate common security bugs*

Therefore (for the purposes of this talk!):

- If we find an instance of a CWE, it is a vulnerability!
- If it crashes the software, it can be a security issue
  - `SIGSEGV` $\Rightarrow$ DoS!

[1] see: `https://www.computer.org/csdl/mags/sp/2004/06/j6076.pdf`

Focus of this talk

~~Chess & McGraw'04~~[1]:

> *Good dynamic analysers can help spot and eradicate common security bugs*

Therefore (for the purposes of this talk!):

- If we find an instance of a CWE, it is a vulnerability!
- If it crashes the software, it can be a security issue
  - `SIGSEGV` $\Rightarrow$ DoS!

[1] see: `https://www.computer.org/csdl/mags/sp/2004/06/j6076.pdf`

A tale of two customers...

Customer A

- We have some testing of open source projects
- Can you find any issues?
- Display issues

Customer B

- VectorCAST performed automated test-case generation
- Can you find any issues?
- Fuzzing of test-cases
- Display issues

What did they want?

The view from the trenches

- Binary – do we have any issues? Yes or no!
- Count – how many?
- Identification – what and where are they?

So how did it work?

Crash-test generation

- Take a test that allocates a pointer
- Remove the `alloc`
- Run it
- Does it crash?
- If yes: potential weakness!

Caveat emptor

- This is white-box unit testing – not black-box “Dynamic Application Security Testing” (DAST)!
- We can only find defects in what we can deduce a test for
    - Not trying to solve the halting problem – things can slip through our net
- Aiming for soundness (if we say it is a bug, it is a bug); no chance of completeness
    - We can’t catch every bug because some are infeasible to generate unit tests for automatically

Example

Example from LIGHTTPD (v1.4.20; v1.4.18 in SATE'08)

```
int buffer_copy_string_buffer(buffer *b, const buffer *src) {
  if (!src) return -1;

  if (src->used == 0) {
    b->used = 0;
    return 0;
  }
  return buffer_copy_string_len(b, src->ptr, src->used - 1);
}
```

- **Not** detected: CPPCHECK, Facebook's INFER, UNO
- *Possible* error: LINT, CODEHAWK
- SIGSEGV: VectorCAST!

Results from SARD

- Took the **`null`** pointer issues from the Software Assurance Reference Dataset[2] ("vulnerable" C test-suite)
- Found 6 out of 7 issues
- We didn't find (`null_deref_local_flow-bad.c`):

```c
/* SNIP */

char k = 'a';
char* p = (char*)NULL;

switch (k)
{
  case 0:
    k = *p;                 /* FLAW */

/* SNIP */
```

[2] see: `https://samate.nist.gov/SARD/`

Benefits

Static analysis might not detect it

- False-positives are high – is it a real error?
- False-negatives exist – maybe they didn't show it?

Dynamic execution

- We don't claim to detect everything
  - "happy" to have false-negatives
- If we *do* find something, it is definitely an issue!
- You can fix the issue, and re-generate and re-execute that test: if the error goes away, that issue is fixed!
  - With static analysis, you might have just hidden the error under a false-negative!

Vulnerabilities of interest

Automatic identification for CWE-398 ("indication of poor code quality")

- Anything with "hard" errors
  - Use of a `null` pointer (CWE-476)
  - Buffer {under,over}flow (stack corruption) (CWE-124)
  - Divide by zero (CWE-369)
- VectorCAST supports stubbing ⇒ detection of
  - Mismatched calls – `malloc`/`free`, `fopen`/`fclose`, `pthread_mutex_lock`/`pthread_mutex_unlock` (CWE-401/404/413/415/590)
  - Bad arguments – `memcpy` (CWE-120/130)
  - Unchecked return – `malloc` (CWE-252)

What are we aiming for?

- Source of tests (pick one!)
  - Take existing tests + code coverage data
  - Symbolic execution data for test-case generation
- Source of defects (pick one!)
  - Static analysis data (from `$YOUR_FAVOURITE_SA_ENGINE`)
  - Symbolic execution data for vulnerabilities
- Generate
  - Fuzz'd tests or tests to cover vulnerabilities
- Execute tests
  - Detect vulnerabilities

What are we aiming for?

- Source of tests (~~pick one!~~)
  - Take existing tests + code coverage data
  - Symbolic execution data for test-case generation
- Source of defects (~~pick one!~~)
  - Static analysis data (from `$YOUR_FAVOURITE_SA_ENGINE`)
  - Symbolic execution data for vulnerabilities
- Generate
  - Fuzz'd tests or tests to cover vulnerabilities
- Execute tests
  - Detect vulnerabilities

Metrics

I thought this was a talk on metrics?!

"actionable intelligence"

Towards "application security"

Process[3]

1. Identify portfolio
2. Assess vulnerabilities
3. Manage risk

Some of the issues we find you might consider are "non-issues" or are mitigated against as part of your software architecture

- That's great...
- ...be wary about software re-use across projects!

[3] see: `https://www.rsaconference.com/writable/presentations/file_upload/asec-w25.pdf`

Easy metrics

An approach to ascertaining quickly Chess's "Morningstar for Software Security"[4]

- ☆☆ – "absence of obvious reliability issues"

The easy ones

- Defect density
  - Defects/SLoC
- Lines free from obvious issues (via code coverage)
  - Confidence of "defect freedom" (but not guaranteed!)
- Ratio of security tests free of defects
  - Higher ratio $\Rightarrow$ more secure

[4] see: http://www.securitymetrics.org/attachments/Metricon-2-Lee-Chess-Enterprise-Metrics.ppt

More involved metrics

- Exploit depth (from how many levels can we trigger it?)
  - Akin to a linear "attack graph"
  - More steps ⇒ high critically
- Criticality (e.g., things that crash vs. things that don't)
  - Assess the risk using CWRAF/CWSS
  - `SIGSEGV` ≫ missing `free`
- Correlation between function complexity and number of defects
  - High complexity and number of defects ⇒ higher risk
- Percentage breakdown of metrics by type/grouping
- Attack surface[5] (e.g., defect via params vs. return from stub)
  - Clearly serious if it is via a stub of `recv`!

[5] see: `http://www.cs.cmu.edu/~pratyus/tse10.pdf`

Sample metrics for `null` pointer defects

Metric	Project: LIGHTTPD	Project: ZLIB	Project: LIBXML2
Version	1.4.20	1.2.8	2.9.4
# files	89	16	84
SLoC[6]	36,605	6,726	184,179
Unique # issues	709	113	2,926
Defect density (defects/line)	1/52	1/60	1/63
Avg. # of tests per defect	11	7	12
Tests hitting defects	69%	28%	40%
Funct's with defects	44%	44%	29%
Funct's with $vg \geq 20$ and defects[7]	51%	55%	66%

[6]measured with CLOC

[7]Jones'08: "[complexity] levels greater than 20 are considered hazardous"

VECTOR software

Future metrics

- Number of vulnerabilities that are already "guarded" (e.g., if a pointer passes through some pointer test but still crashes)
  - Similar to disregarding issues if they are guarded by "intrusion protection systems"[8]
- Build a correlation to predict the vulnerability of a package[9]:
  - Extract a characteristic of the software for version n
  - Extract a vulnerability metric from the software for version n
  - Use characteristics of $n+1$ to predict vulnerabilities in $n+1$

[8] see: `http://www.securitymetrics.org/attachments/Metricon-1-Epstein-Software.ppt`

[9] see: `http://www.securitymetrics.org/attachments/Metricon-5-Massacci-Firefox-Vulnerabilities.pdf`

Take-home

Mainly: no "one size fits all" solution – use multiple tools!

- Dynamic execution can find certain vulnerabilities more definitively
- Need to always consider DP-E ratio (damage potential vs. effort)
- A number of metrics
  - Not necessarily specific to dynamic execution – also relevant to the output of a static analyser
- Future work: how can metrics be used to *predict* vulnerability

Questions?

andrew.jones@vectorcast.com