Know your tools: quirks and flaws of integrating SAST into your pipeline

Artem Bychkov
(artem.bychkov@huawei.com)

Overview of the talk

What is in scope

- Technology-neutral intro into SAST tools:
  - how they work under the hood
  - what limitations they have
- Discussion of common grey areas, misunderstandings and mistakes when using SAST tools

What is out of scope

- Academic/scientific discussion of SAST and related problems
- Other types of security testing
- General DevSecOps process organization

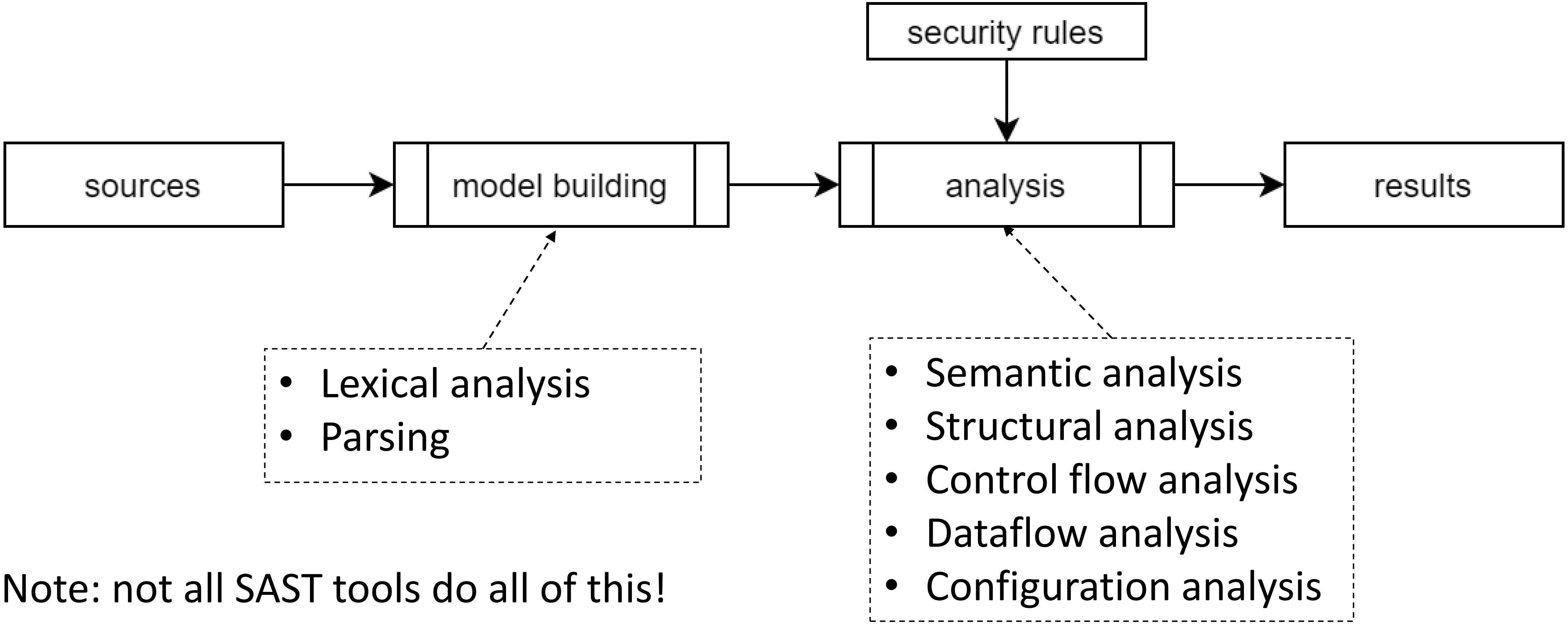
About me

- 14 years in security
- 8 years in AppSec
- Currently: principal security engineer at Advanced Software Technology Lab
  @ Huawei Moscow Research Center
- Previously: security consultant and in-house security engineer focusing on AppSec and offensive disciplines
- Industries worked in: financial, telecom, transportation, oil&gas, retail

What is SAST?

- SAST stands for Static Application Security Testing
- Based on static code analysis with security knowledge mixed in
- Static code analysis is performed without executing the code
- Both sources and compiled binaries may be analyzed statically. We focus exclusively on source code analysis in this talk.

Overview of a general SAST process



Note: not all SAST tools do all of this!

Model correctness – why is this important?

- Incomplete or incorrect model will prevent certain most useful analysis types
- Vulnerable component may not be analyzed at all

Build model – what can go wrong?

- SAST tool may have not resolved all symbols
  - and it's really a great feature if it reports missing symbols!
- SAST tool may not support certain language version or features
  - if parsing fails gracefully and silently, SAST coverage suffers
- Using different project configs for build and SAST environments
  - leads to possible inconsistent results

A primer on model building – 1

Suppose we have a simple project:

```
TEST_PROJECT
    __init__.py
    test.py
```

test.py

```python
import rand

print(f"Hello world! Yor secure random number is:\
 {rand.secure_random()}")
```

'test.py' imports 'rand' module, which currently is not present in the project import path. Its source code is also simple:

rand.py

```python
import random

def secure_random():
    return random.randint(0, 100)
```

A primer on model building – 2

Now, let’s run different tools and linter

```
sast_demo :> pylint test.py
************* Module test
test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test.py:2:0: E0401: Unable to import 'rand' (import-error)

------------------------------------------------------------------
Your code has been rated at -10.00/10 (previous run: -13.33/10, +3.33)
```

```
[redacted]>sourceanalyzer -b test01 test.py

[redacted]>sourceanalyzer -b test01 -show-build-warnings
Item 'secure_random' not found in package 'rand' at [redacted]\test_project\test.py:4:6
The Python frontend was unable to resolve the following import:
        rand at [redacted]\test_project\test.py:1.
```

```
sast_demo :> bandit test/test.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.6.9
Run started:2021-02-22 09:10:22.575241

Test results:
        No issues identified.

Code scanned:
        Total lines of code: 2
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 0.0
        Total issues (by confidence):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 0.0
Files skipped (0):
```

A primer on model building – 3

Now, let's fix broken imports and run our tools again

```
sast_demo :> pylint test/test.py
************* Module test.test
test/test.py:1:0: C0114: Missing module docstring (missing-module-docstring)

-----------------------------------
Your code has been rated at 5.00/10
```

```
[redacted]>sourceanalyzer -b test02 test.py

[redacted]>sourceanalyzer -b test02 -show-build-warnings

[redacted]>sourceanalyzer -b test02 -show-files
rand.py
test.py
```

```
[redacted]>sourceanalyzer -b test02 -scan
                                                                  ]
[redacted]

[DE9900FC556FA17FE3CED793376FBA68 : high : Insecure Randomness : semantic ]
rand.py(4) : randint()
```

*bandit's output does not change

A primer on model building – 4

Finally, let's check what happens if we explicitly add file to bandit's scan:

```
sast_demo :> bandit test/test.py test/rand.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.6.9
Run started:2021-02-22 09:23:43.244421
```

```
Test results:
>> Issue: [B311:blacklist] Standard pseudo-random ge
   Severity: Low   Confidence: High
   Location: test/rand.py:4
   More Info: https://bandit.readthedocs.io/en/lates
3       def secure_random():
4           return random.randint(0, 100)
5


--------------------------------------------------


Code scanned:
        Total lines of code: 5
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0.0
                Low: 1.0
```

A note on language features support – 1

Few years ago, I had mix of the following constructions in one Scala project:

Insecure string interpolation:

```
val insecure_query: String = s"SELECT * from users where userid=${userid}";
```

Secure custom string interpolation from Anorm library:

```
val secure_query: String = SQL"SELECT * from users where userid=${userid}";
```

… and our SAST tools did not report SQLi in the first case.

Turned out, our tool did not support all features of the supported Scala version.

This led to a several cases of **false negatives**

(== missed vulnerabilities)

A note on language features support – 2

After few support tickets, support for interpolation was added, but:

Correctly reported as insecure:

```
val insecure_query: String = s"SELECT * from users where userid=${userid}";
```

Incorrectly reported as insecure:

```
val secure_query: String = SQL"SELECT * from users where userid=${userid}";
```

… turned out, only partial support for interpolation was added at the moment (no support for custom one).

This led to a several cases of **false positives**

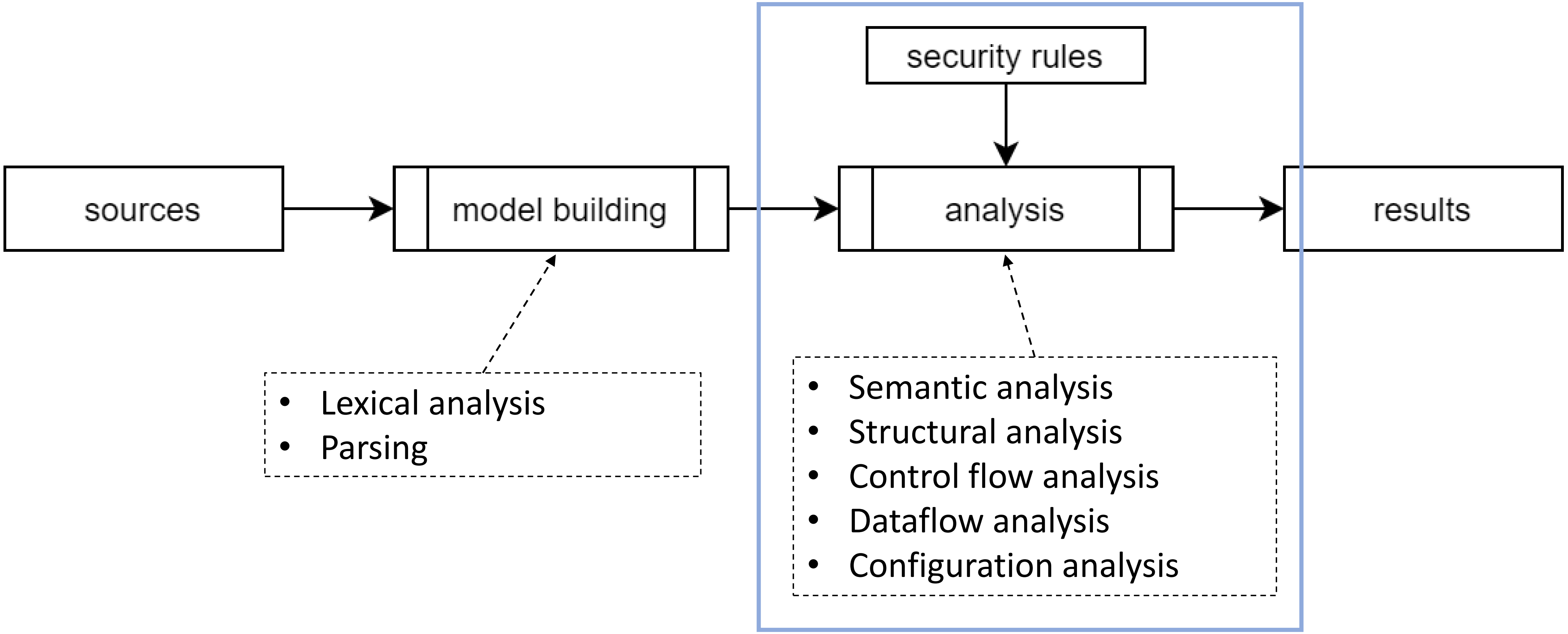
(==false vulnerabilities)

Strategy for ensuring model correctness

- Check all the warnings and errors that SAST tool emits
- If SAST is not reporting translation errors, ensure that:
  - Linter does not produce errors
  - Build process does not fail
- If SAST tool requires adding all modules to analysis explicitly, ensure to include all paths to sources and dependencies
- If possible, check that all files were included
- Write tiny test cases to check whether the tool understands new or advanced features
  of complex languages (like Scala)

Congrats! Now the project is correctly parsed!

But this is just the beginning. Let’s go to the analysis phase!



Analyzer overview: semantic

- Operates on identifiers, resolved symbols and types
- Searches for usage of specific insecure code:

```
[DE9900FC556FA17FE3CED793376FBA68 : high : Insecure Randomness : semantic ]
rand.py(4) : randint()
```

- Good semantic analyzers can detect simple cases of indirect calls to insecure code, such as:

```
from somemodule import insecure_function as secure_function

secure_function()
```

- Think about it as of “grep on steroids”

Analyzer overview: structural

- Checks for language-specific violations of safe coding practices
- Detects improper variable/functions/methods access modifiers, dead code, insecure multithreading, memory leaks, etc.
- Hardcoded secrets are also detected by this analyzer:

```
[88B4C70DFC8EBA7BC561AC6EF34CE4CD : low : Password Management : Password in Comment : structural ]
    test.py(3)
```

Analyzer overview: control flow

- Analyzing possible execution paths and control flow graphs
- Detecting flaws such:
  - Dangerous sequences of actions
  - Resource leaks
  - Race conditions
  - Improper variable/object initialization before use

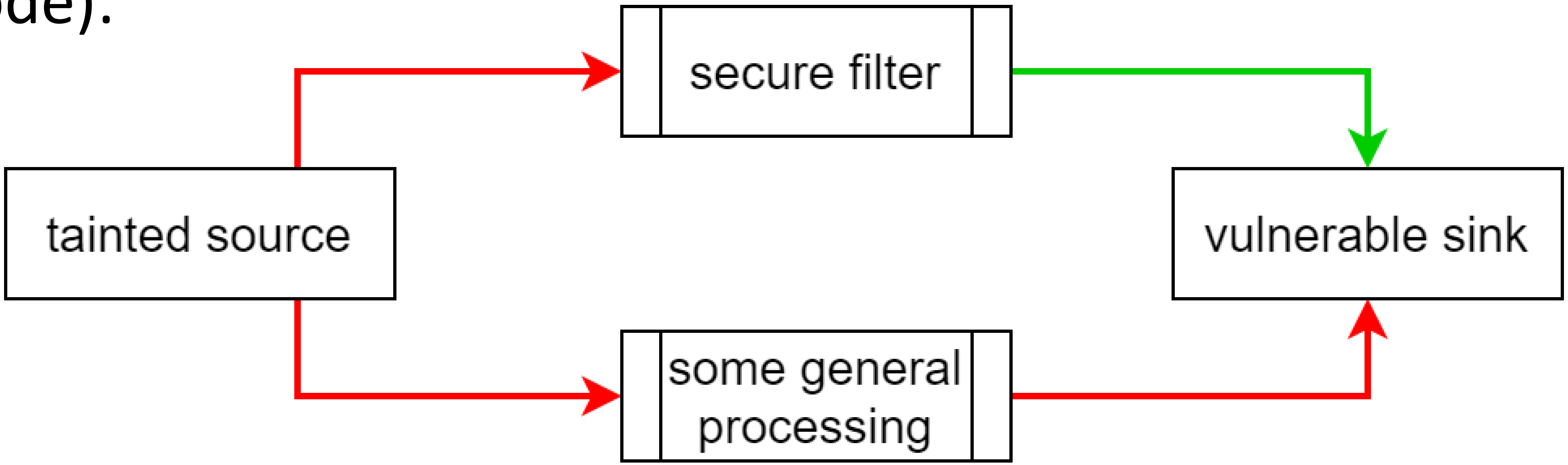
Analyzer overview: control flow – example

```
5      public void nullPointer(int id) {
6          TestClass t;
7          t = null;
8          if (id > 0) {
9              t = new TestClass(id);
10         }
11         t.process();
12     }
```

```
[28F788EBA338701471ABBF90F95F2915 : high : Null Dereference : controlflow ]
    NullPointerSample.java(7) : Assigned null : t
    NullPointerSample.java(8) : Branch not taken: (id <= 0)
    NullPointerSample.java(11) : Dereferenced : t
```

Analyzer overview: dataflow – 1

- The most powerful type of analyzer:
  tracks data flow from **taint source** (i.e., attacker-controlled inputs, like HTTP controller) to vulnerable **sink** (exploitable code):



Analyzer overview: dataflow – 2

- Detects injections, buffer overflows, format-string attacks and any other type of vulnerability relevant to known sinks
- Most advanced SAST tools may use symbolic execution with automated theorem proving/SMT solvers to improve results quality
- Downside: usually takes the most time to run
- Downside: may not be accessible in incremental mode scanning (if supported by SAST tool – check the docs!)

Analyzer overview: dataflow – 3

Example of path manipulation vulnerability discovered by the dataflow analyzer:

```
String filename = args[0];
try {
    filename = "" + (Integer.parseInt(filename) % 3);
} catch (Exception e) {
    System.out.println("Invalid input.");
}
new FileReader(filename).read(buffer);
```

```
[176CC0B182267DD538992E87EF41815F : critical : Path Manipulation : dataflow ]
EightBall.java(12) :  ->new FileReader(0)
    EightBall.java(6) : <=> (filename)
    EightBall.java(4) :  ->EightBall.main(0)
```

Analyzer overview: configurational

- Operates with known configuration files formats (and may be aware of certain frameworks' specifics)
- Detects known security misconfigurations
- Won't work if you use custom configuration or framework unknown to the tool

```
[399A248E35AE0FBB04255DE45FA9754C : low : J2EE Misconfiguration : Missing Error Handling : configuration ]
    web.xml(7)

[C2F39B963AB1AFACD8D57325EEAF747F : low : J2EE Misconfiguration : Excessive Session Timeout : configuration ]
    web.xml(7)
```

What can go wrong with analysis?

- Incomplete analysis model may lead to both false positives (FPs) and false negatives (FNs) – <u>check model building process</u>!
- Heavy use of runtime-determined behavior may make SAST an intractable problem. Examples:
  - Dynamic code import/loading
  - Use of high-order constructions, like in functional programming

<u>In such cases, add focus to dynamic testing</u>

- High level of false results, because stock rules (sets if sources and sinks) do no fit the project or its dependencies

Strategy for handling FP/FN

- Do NOT disable rules relevant for the project type (but it's OK to disable Android security rules for Spring Boot app)
- Audit project and add your custom sources, sinks and taint filters to semantic/dataflow rules
- Add exceptions to a clear false positives, or bugs non-relevant to the project
- Avoid adding too general exceptions (i.e., "ignore all defects if the file is located in the directory $(dir)")
- If adding taint-removal rule, keep track if it is 3rd party code

Custom rules for 3rd party code

- Suppose we have an XSS filter in 3rd party lib:

- Now, someone (maliciously or accidentally) removes following characters from the filter so now it may be bypassed: **“<>**

- To detect this when pulling new version of a library, consider adding unit or functional tests to complement SAST rule!

What types of defects are not good for SAST?

- New vulnerabilities, not covered by the rules
- Design/architecture flaws
- Logical vulnerabilities
- Operational vulnerabilities
- Complex multistage/trust exploiting vulnerabilities

Summing up: selecting SAST tool

- Make sure it supports your language and its features
- Prefer tools that supports your frameworks of choice
- Make sure you have solid understanding of tools specifics and mode of operation
- Prefer tools that offer customization, at least in the form of custom rules

Summing up: operating SAST tool

- Invest time to cover projects with custom rules, if stock rules does not fit
- If scan duration is critical, do not turn off heaviest and most powerful analyzers completely. Instead, consider running them on periodic schedule, rather than on each build in CI/CD pipeline.
- Do not put too much trust into “clean” results – SAST is not a magic silver bullet, and not sufficient alone. Do not forget another types of security testing!

Thank you!

Artem Bychkov
(artem.bychkov@huawei.com)