



wSAST

CODE REVIEW TOOL OF THE FUTURE (MOSTLY BECAUSE IT IS NEVER FINISHED)

Whoami?



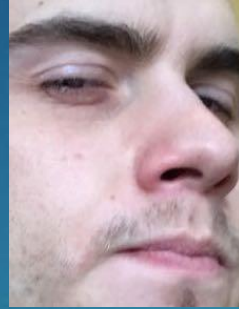
- ▶ Peter Winter-Smith (@peterwintrsmith on Twitter)
- ▶ Most well known as Peter Wiener from Burp/Portswigger Labs
- ▶ Previously: AppSec consultant (web, client, native) with code focus
- ▶ Now: Offensive tool developer
- ▶ One previous public project: Crystal Anti-Exploit Protection 2012
(source: <https://github.com/peterwintersmith/crystalaep>)

What the...



- ▶ Code analysis platform with multiple language support
- ▶ Provides code searching, graphing, dataflow and static SAST
- ▶ Lowers code to an IL resembling simplified Java with some extensions to increase flexibility
- ▶ Plugin architecture that allows anything in the IL format to be a “source” or “sink”
- ▶ Designed to support real consultant needs rather than only dev teams (e.g. no compilation, partial source)
- ▶ Designed for constrained platforms – single or multi-threaded parsing, analysis, scanning, and memory/time guardrails

Why the...



- ▶ I've had an interest in parsing and code analysis for the past 10 years
- ▶ First attempt in 2013 failed miserably – couldn't even write a proper EBNF processor; read a few books on parsing and they were too complicated
- ▶ Revisited them several years later with increased maturity and focus, found some better books, and second time around the subject made more sense
- ▶ Made employment changes in 2020 for multiple reasons, one of which was to allow me to focus more on developing this tool
- ▶ Goal is to produce a cheap, partially open source, community centered alternative to expensive/restricted SAST platforms, with some added extras to help in initial assessment phases



Technical Overview

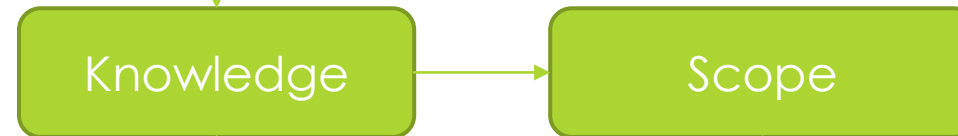
Technical - High Level Design

- ▶ Multiple major components, including:

- ▶ Code Processors:



- ▶ Context Analyzers:



- ▶ Semantic Analyzers:



- ▶ Symbolic Execution:



- ▶ Static Evaluation

Analysis Phases – Code Processing

- ▶ Multiple major components, including:

- ▶ Code Processors:



- ▶ Context Analyzers:



- ▶ Semantic Analyzers:



- ▶ Symbolic Execution:



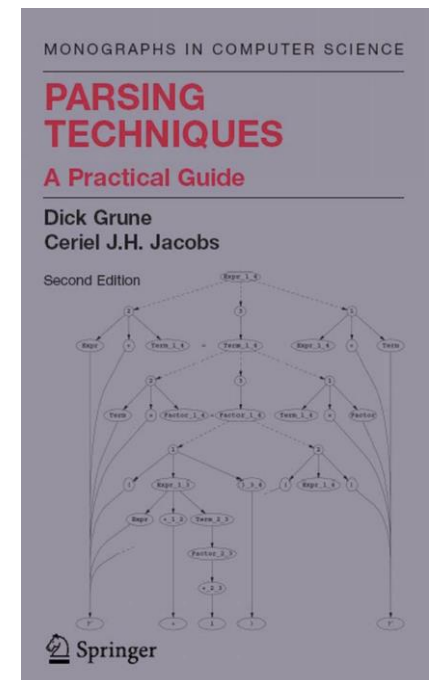
- ▶ Static Evaluation

Parsers

Problem 6.6: *Project:* The GRDP parser of Section 6.6.3 cannot handle left recursion, but it seems reasonable that that can be remedied as follows. Upon calling the routine for a non-terminal L we first suppress all left recursion; this gives us the set of lengths of segments (if present) that match L non-recursively. Then we call the routine for L again, now feeding it these lengths to use for the left-recursive call, so it can collect more; etc. In the end no more new matches are found, and the collected lengths can be returned. Turn this into a complete algorithm.



- ▶ Idea for wSAST born out of experimentation with parsers
- ▶ Read numerous books on parsing, ended up writing own framework based on an idea in Grune book
- ▶ The framework supports $LL(*)$ parser generation – it works roughly like this:
 - ▶ EBNF grammar is processed into an AST and annotated with information on left recursion (which productions and rules are LR)
 - ▶ Code is generated to handle different rule types ((non-)terminals, optionals, repetitions, etc.)
 - ▶ For a given production all non-LR rules are matched, then the entire list of matches are fed into the LR rule matchers and these are added to the set of matches
 - ▶ This process recurs until all matches are exhausted at which point a complete set of possible parse trees have been found



Translators

- ▶ A parse tree must be converted into an intermediate language called WSIL which is understood by wSAST
- ▶ WSIL is similar to Java, but with a few changes to support some concepts from C++/C# (see: WSIL.ebnf)
- ▶ The parsers implement an ILanguageProvider interface and must invoke a callback to provide updates including the WSIL tree
- ▶ A helper class is provided to help build WSIL trees and a validator to ensure they're correctly formed
- ▶ References to the original parse tree and associated source files can be stored within the tree; ambiguous parses can be recorded and resolved later if needed

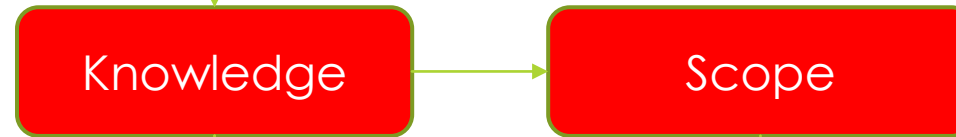
Analysis Phases – Context Analysis

- ▶ Multiple major components, including:

- ▶ Code Processors:



- ▶ Context Analyzers:



- ▶ Semantic Analyzers:



- ▶ Symbolic Execution:



- ▶ Static Evaluation

Knowledge Analyzer

- ▶ The KnowledgeDB (KDB) is the authoritative database representing the entire codebase under analysis
- ▶ WSIL trees are walked in a single phase by the Knowledge Analyzer using a recursive tree walk which builds the KDB in tandem
- ▶ Elements of the KnowledgeDB (KDB) are CodeObjects and have types such as NAMESPACE, CLASS, METHOD, BLOCK, VARIABLE etc.
- ▶ Each CodeObject has an associated path, WSIL node reference, scope information, and internally a search function to handle proper searching
- ▶ The KDB exposes methods for code searching, accessing certain properties of code objects like type parameters and code block exits

Scope Analyzer

- ▶ The Scope Analyzer walks a WSIL tree in a single pass attaching a UnitScope annotation to each associated CodeObject KDB entry
- ▶ The UnitScope contains information such as:
 - ▶ The code unit to which the CodeObject belongs (often source file)
 - ▶ The resolved and unresolved imports
 - ▶ The namespaces visible to the CodeObject
 - ▶ A scope field that anchors the visibility of the object (in case of name conflicts)
- ▶ The scope information is used by the KDB when performing searches, to ensure only visible CodeObjects are identified, as well as to clear variables that fall out of scope during analysis

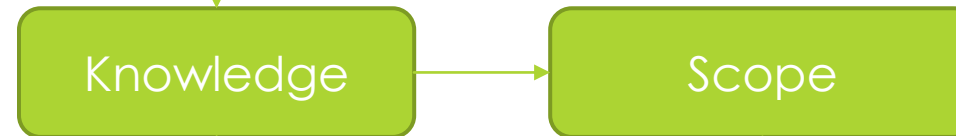
Analysis Phases – Semantic Analysis

- ▶ Multiple major components, including:

- ▶ Code Processors:



- ▶ Context Analyzers:



- ▶ Semantic Analyzers:



- ▶ Symbolic Execution:



- ▶ Static Evaluation

Type Analyzer

- ▶ The Type Analyzer walks each tree in the entire codebase, annotating the WSIL tree nodes with type information
- ▶ Whenever a well-known type (like a literal), or a defined type (like a variable) is used, the pass annotates the associated identifier or expression with the type as a string (e.g. "int4", "boolean", "MyClass", "unresolved:String")
- ▶ Expressions are annotated by looking up type information known thus far in the KDB and propagating throughout the remainder of the expression
- ▶ Other sources of type information such as method return values are also used and propagated
- ▶ Generic types are specialised with usage and added to the KDB here

Global CF Analyzer

- ▶ The Global Control Flow Analyzer phase then processes all method calls, attempting to connect the WSIL “call” expression nodes to the tree elements for the methods they invoke
- ▶ To do this a best fit approach is used - taking into consideration:
 - ▶ Parameter types and counts
 - ▶ Types and inheritance of instance methods
 - ▶ Types and inheritance of parameter types (for closer matching)
 - ▶ Variable parameter counts
 - ▶ Generic class/method type specializations
- ▶ This wires up constructor calls too when a WSIL “new” call is encountered
- ▶ Generic types encountered are also specialized with usage here

Global CF + Type Analyzer Phases

- ▶ The global CF analysis and type analysis phases are interdependent and continue until no new information is obtained (no new control flow paths, no new type information propagated)
- ▶ This is necessary because type information influences control flow (for example, if we learn that a local variable is a class instance, we can resolve methods on it if called)...
- ▶ ... and control flow influences type information because we can then propagate that method return types, etc.

Local Control Flow Analyzer

- ▶ The Local Control Flow Analyzer represents the final phase of analysis; it annotates WSIL statement nodes with information representing how they influence control flow (e.g. if/else, switch, goto, try/catch)
- ▶ This phase is a tree walk which descends statement blocks linking them together to form a graph over the WSIL tree
- ▶ Each non-control-flow-altering statement is assigned a “basic block” or “throwable basic block” type and is linked to the statement following
- ▶ Control flow influential statements (such as if/else) are linked to the block of “basic blocks” based on their separate statement blocks
- ▶ The process tracks special statements (such as loop break/continue, throwable basic blocks) and wires them up when such statement types are encountered

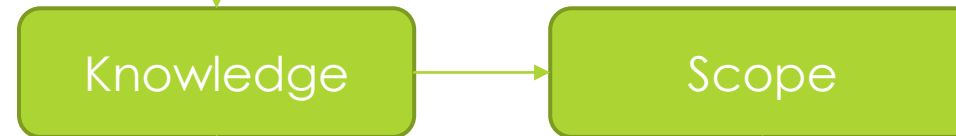
Analysis Phases – Dataflow Analysis

- ▶ Multiple major components, including:

- ▶ Code Processors:



- ▶ Context Analyzers:



- ▶ Semantic Analyzers:



- ▶ Symbolic Execution:



- ▶ Static Evaluation

Entrypoint Analysis

- ▶ Once the analysis phases are completed, before dataflow analysis can proceed it is necessary to find the set of all entrypoints into a codebase
- ▶ This is usually a superset of all real entrypoints into a codebase due to missing code
- ▶ Entrypoint analysis finds all functions and free code blocks with no callers (or preceding statements), recursively walking backwards from all methods eliminating candidates until a minimal set is identified
- ▶ Recursive functions which have no single caller as an entrypoint are picked randomly as entrypoints

Dataflow Engine (1)

- ▶ When all of the analysis phases are completed, it is possible to invoke the Dataflow Engine to simulate execution
- ▶ The dataflow engine walks the local control flow graph from the set of entrypoints determined, including entering called functions and constructors
- ▶ The engine switches class contexts (creating necessary this pointer, member variables, outer class variables) as needed when calling across class boundaries, and walks inheritance hierarchies as needed

Dataflow Engine (2)

- ▶ For each “step” of “execution” any registered source or sink callbacks are invoked; these are passed all execution state information thus far, the node on which they’ve been invoked, all variables and their state information etc.
- ▶ These can create new variables, set variables as traced, assign arbitrary data to variables
- ▶ Dataflow state and variables are cloned whenever updated, and merged at the convergence of control flow points to prevent incorrect findings

Sources + Sinks + Subscribers

- ▶ These are registered with the engine and can implement custom checks
- ▶ Sources are attached to the execution state, and sinks are attached to variables
- ▶ Sources and sinks can record variables, create new ones, set traced status, validate previously encountered sinks, examine code path and WSIL tree context, and even attach arbitrary data to variables
- ▶ Whenever a source reaches a matching sink the sink can choose to create a “source-to-sink-match” object and pass it to any registered subscribers
- ▶ Subscribers report findings and are provided with source, sink and any other context needed such as recorded variables



Functionality Overview

Usage – High Level

- ▶ Command-line driven dataflow or static scans
- ▶ Interactive mode to allow REPL-style search, graphing, iterative scans

```
[*] wSAST v0.1-alpha (release date 18-12-2023) (c) 2023 Peter Winter-Smith (@peterwintsmith)
Licensed to: 0.1-alpha (Company: 0.1-alpha)
Expiration: 17-Mar-2024

activate
  Activates the software for use.
  [activation-id] - The activation ID for the license

dataflowscan
  Performs source-to-sink analysis of the specified sources; generates a report of results.
  <sources>      - Paths (comma separated) to scan for source files; !regex excludes path.
  <project>      - Name of project.
  [config]       - Path to wSAST configuration XML (default .\config.xml).
  [languages]    - Languages to scan (default: all).
  [filter]       - Entrypoint name regexes to scan from (comma separated; multiple means OR, ! prefix means AND NOT).

staticscan
  Perform static syntax tree analysis of the specified sources; generates a report of results.
  <sources>      - Paths (comma separated) to scan for source files; !regex excludes path.
  <project>      - Name of project.
  [config]       - Path to wSAST configuration XML (default .\config.xml).
  [languages]    - Languages to scan (default: all).
  [filter]       - Entrypoint name regexes to scan from (comma separated; multiple means OR, ! prefix means AND NOT).

interactive
  Perform interactive analysis of the specified sources.
  <sources>      - Paths (comma separated) to scan for source files; !regex excludes path.
  <project>      - Name of project.
  [config]       - Path to wSAST configuration XML (default .\config.xml).
  [languages]    - Languages to scan (default: all).
```


Usage – Interactive

- ▶ Graph class relationships, calls, local control flow and WSIL AST
- ▶ Search KnowledgeDB or AST to locate specific methods, classes, variables.
- ▶ List calls to and from particular methods, classes
- ▶ Find paths between methods
- ▶ Search is regex based and inclusive or exclusive

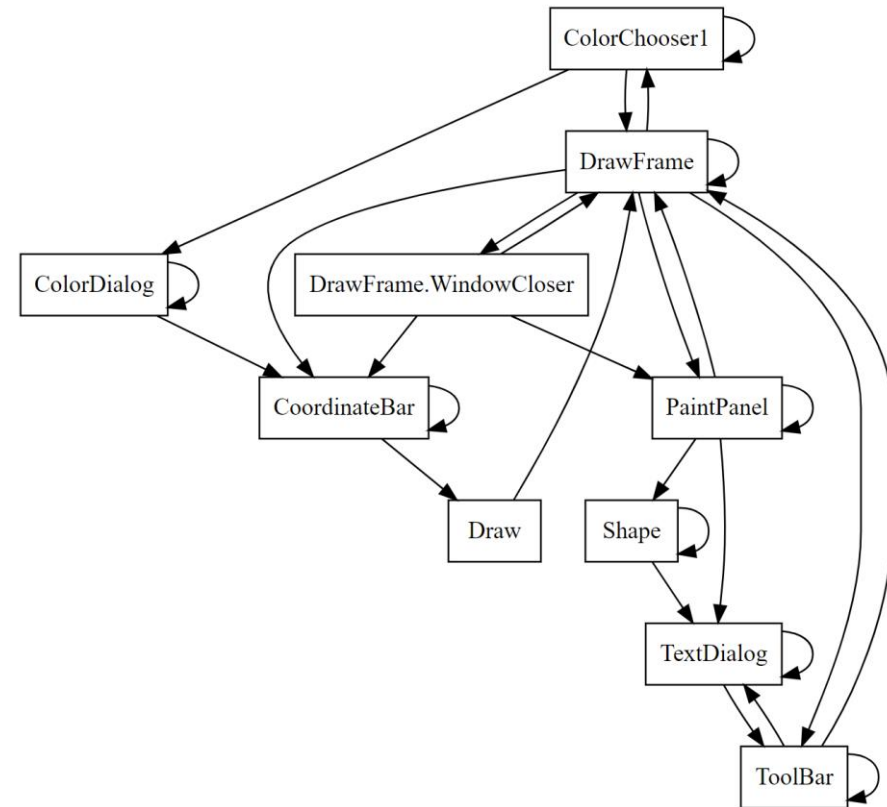
```
[*] Interactive code analysis mode
Command> help

graph      Create graphviz DOT output from knowledge
graph <type> [...]
  <type>      - classes, calls, local, ast
  [--filter=] - comma separated regex of KDB entries (multiple means OR, ! prefix means AND NOT)
  [--inclusive] - specifies that either source OR destination of a link can match filter
  [--filter-root] - filter applies only to root code object (only: calls)
  [--highlight=] - comma separated regex (multiple means OR, ! prefix meant AND NOT)
  [--entrypoints] - highlight entrypoints
  [--filename=] - output file (in project directory)
  [--tokens] - print tokens in graph labels (warning: this can be inaccurate for languages other than WSIL) (only: local)
  [--locs] - print code filename and line/column info (only: ast)
  [--ignorecase] - filters are case insensitive
kdbsearch  Search knowledge base entries
kdbsearch <filter> [...]
  <filter>      - comma separated regex of KDB entries (multiple means OR, ! prefix means AND NOT)
  [types]      - [g]lobal, [n]amespace, [i]nterface, [c]lass, [m]ethod, [c]onstructor, [b]lock, [v]ariable
  [--ignorecase] - filters are case insensitive
astsearch  Search AST
astsearch <filter> [...]
  <filter>      - comma separated regex (multiple means OR, ! prefix means AND NOT)
  [types]      - [t]ype, [l]abel, [v]alue
  [--filter=] - comma separated regex of KDB entries (multiple means OR, ! prefix means AND NOT)
  [--ignorecase] - filters are case insensitive
calls      List calls to and from specific methods
calls <direction> <path> [...]
  <direction> - to, from
  <filter>      - comma separated regex of KDB method entries (multiple means OR, ! prefix means AND NOT)
  [--depth=] - depth of calls to trace (default: 99999)
  [--locs=] - number of surrounding lines of code to print (default 0)
  [--ignorecase] - filters are case insensitive
paths      Find paths connecting functions
paths <filter-start> <filter-end> [depth]
  <filter-start> - comma separated regex of KDB entries (multiple means OR, ! prefix means AND NOT)
  <filter-end> - comma separated regex of KDB entries (multiple means OR, ! prefix means AND NOT)
  [--depth=] - depth of calls to trace (default: 99999)
  [--locs=] - number of surrounding lines of code to print (default 0)
  [--ignorecase] - filters are case insensitive
dfscan     Reload config and perform dataflow scan
dfscan [--filter=] - comma separated regex of KDB entries (multiple means OR, ! prefix means AND NOT)
sscan      Reload config and perform a static code scan
sscan [--filter=] - comma separated regex of KDB entries (multiple means OR, ! prefix means AND NOT)
clear      Clear console.
exit       Exit to console.

Command>
```

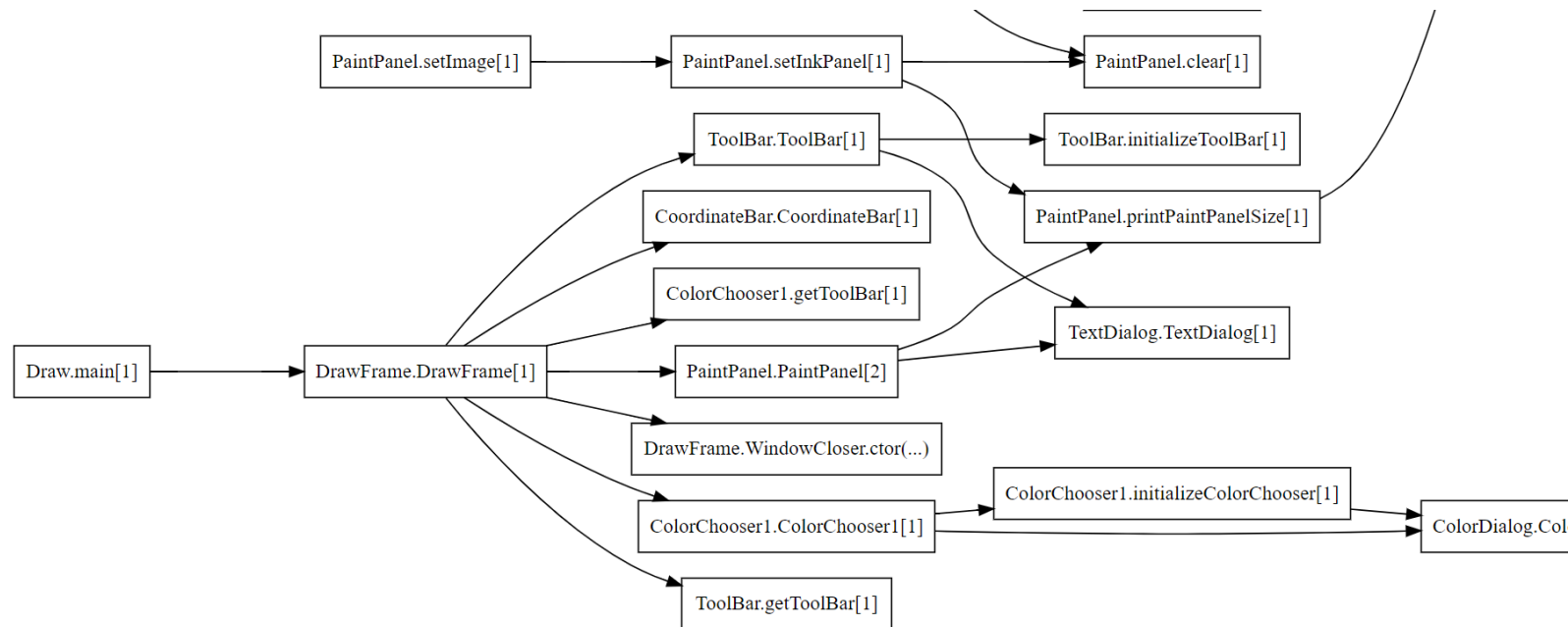
Example – Graphing Classes

- Relationships between classes can be easily observed



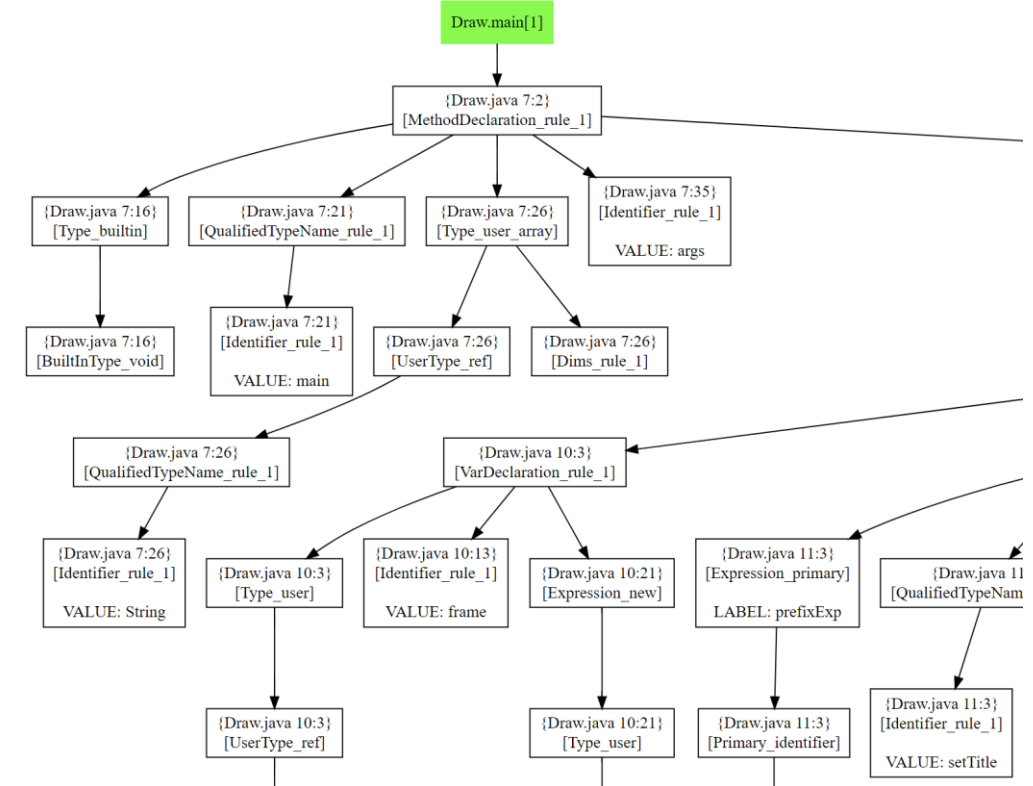
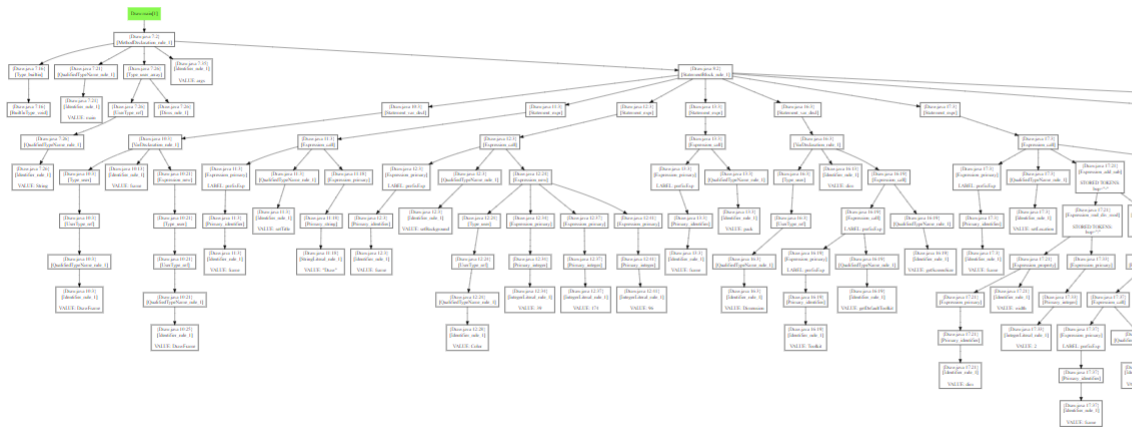
Example – Graphing Calls

- Call flows can be graphed, filtered as required and highlighted



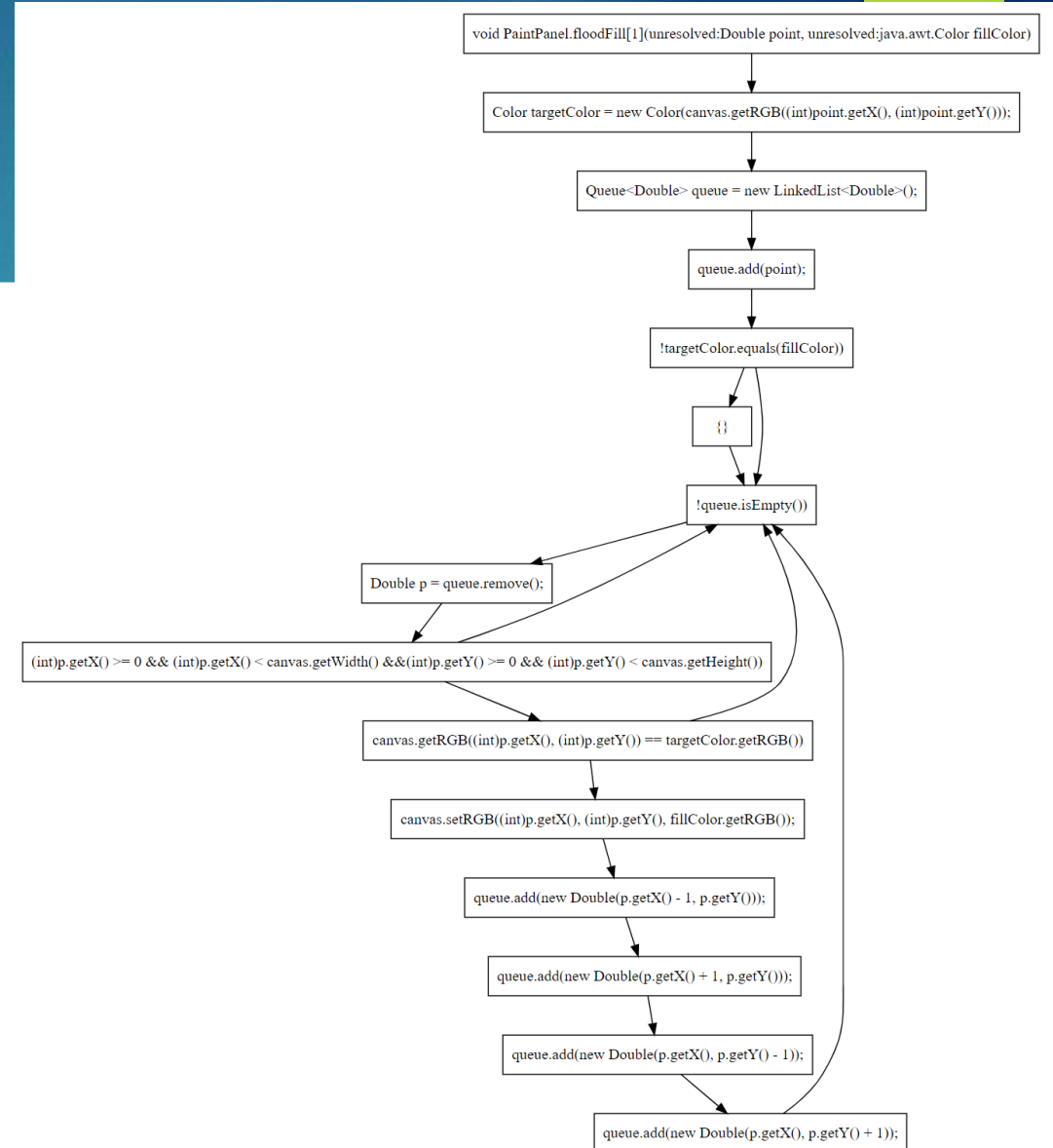
Example – Graphing AST

- WSIL AST graph is useful for debugging if writing a language plugin



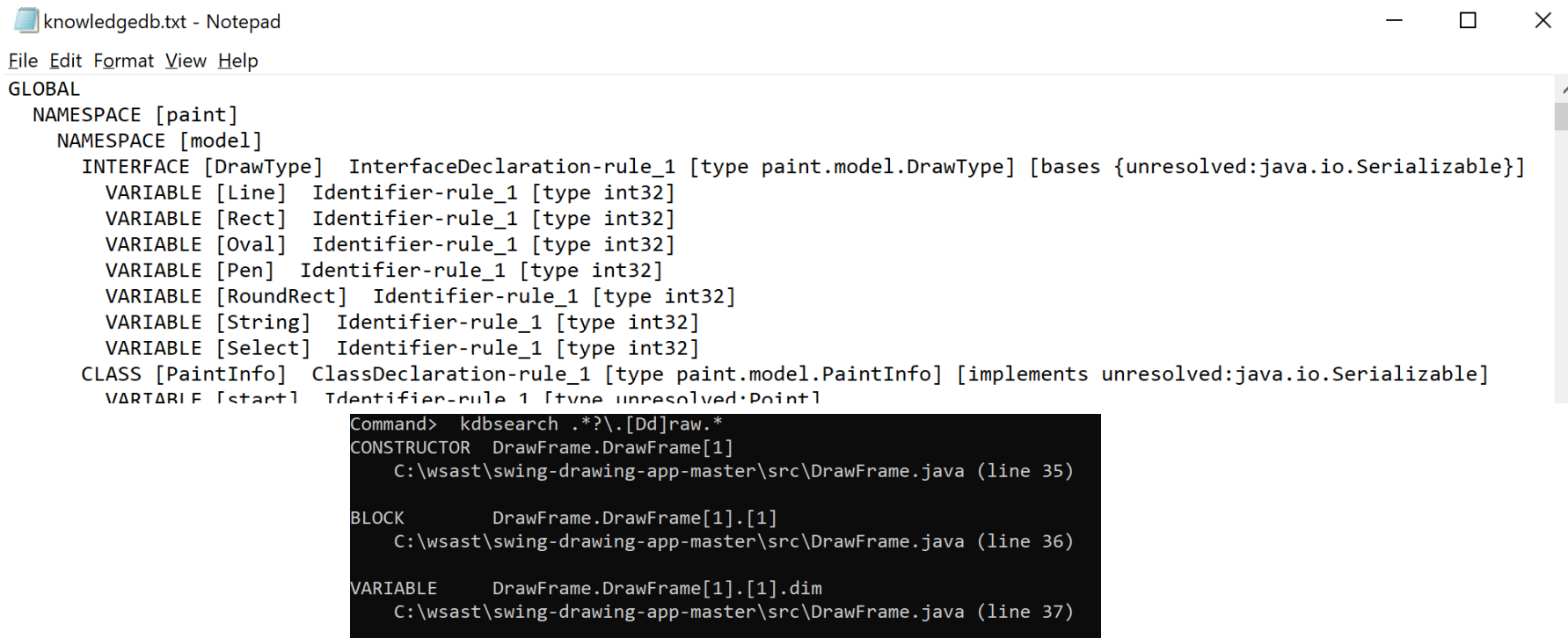
Graphing - Local

- ▶ Local control flow graph is not very useful in practice but fun to examine
- ▶ Likely to be useful if an IDE plugin is ever created



Search – Knowledge DB

- Everything about the code is kept in the Knowledge DB, which is dumped into a text file in the project directory, but can also be searched in wSAST:



The screenshot shows a Notepad window titled 'knowledge.db.txt - Notepad' with a menu bar (File, Edit, Format, View, Help). The text content is as follows:

```
GLOBAL
NAMESPACE [paint]
  NAMESPACE [model]
    INTERFACE [DrawType] InterfaceDeclaration-rule_1 [type paint.model.DrawType] [bases {unresolved:java.io.Serializable}]
      VARIABLE [Line] Identifier-rule_1 [type int32]
      VARIABLE [Rect] Identifier-rule_1 [type int32]
      VARIABLE [Oval] Identifier-rule_1 [type int32]
      VARIABLE [Pen] Identifier-rule_1 [type int32]
      VARIABLE [RoundRect] Identifier-rule_1 [type int32]
      VARIABLE [String] Identifier-rule_1 [type int32]
      VARIABLE [Select] Identifier-rule_1 [type int32]
    CLASS [PaintInfo] ClassDeclaration-rule_1 [type paint.model.PaintInfo] [implements unresolved:java.io.Serializable]
      VARIABLE [start] Identifier-rule_1 [type unresolved:Point]
```

Below the Notepad window, a terminal window shows the execution of a search command:

```
Command> kdbsearch .*?.[Dd]raw.*
CONSTRUCTOR DrawFrame.DrawFrame[1]
  C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 35)

BLOCK      DrawFrame.DrawFrame[1].[1]
  C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 36)

VARIABLE    DrawFrame.DrawFrame[1].[1].dim
  C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 37)
```

- This is useful for finding interesting areas of code from specific regexes, such as auth, password, database, models etc.

Search - Calls

- Calls to and from methods with particular regex names can be generated in list form
- This can be a useful way of finding references to functions of interest (and all the ways to reach them)

```
Command> calls to .*?\.[Dd]raw.*
new      DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\Draw.java (line 10)    METHOD    Draw.main[1]

call     unresolved:java.awt.Graphics.drawImage
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 120)  METHOD    PaintPanel.paintComponent[1]

call     unresolved:java.awt.Graphics2D.drawLine
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 130)  BLOCK    PaintPanel.paintComponent[1].[1].[2].[1]

call     unresolved:java.awt.Graphics2D.drawRect
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 134)  BLOCK    PaintPanel.paintComponent[1].[1].[2]

call     unresolved:java.awt.Graphics2D.drawOval
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 141)  BLOCK    PaintPanel.paintComponent[1].[1].[2]

call     unresolved:java.awt.Graphics2D.drawString
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 149)  BLOCK    PaintPanel.paintComponent[1].[1].[2]

call     unresolved:java.awt.Graphics2D.drawLine
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 157)  BLOCK    PaintPanel.paintComponent[1].[1].[2]

call     unresolved:java.awt.Graphics2D.drawRect
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 162)  BLOCK    PaintPanel.paintComponent[1].[1].[2]

call     unresolved:java.awt.Graphics2D.drawOval
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 169)  BLOCK    PaintPanel.paintComponent[1].[1].[2]

call     unresolved:java.awt.Graphics2D.drawImage
C:\wsast\swing-drawing-app-master\src\PaintPanel.java (line 189)  METHOD    PaintPanel.setImage[1]
```

```
Command> calls from .*?\.[Dd]raw.*
C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 35)  CONSTRUCTOR DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 37)  call         unresolved:<unknown>.getScreenSize

C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 35)  CONSTRUCTOR DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 37)  call         unresolved:<unknown>.getDefaultToolkit

C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 35)  CONSTRUCTOR DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 41)  new          <missing-info>

C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 35)  CONSTRUCTOR DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 42)  call         unresolved:javafx.swing.JPanel.setLayout
```


Search - Paths

- ▶ Paths between functions (or sets of functions) can be graphed; just specify the start name regex and end name regex:

```
Command> paths .*?\.[Dd]raw.* .*?\.[Ss]etBorder.*
C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 48)    CONSTRUCTOR DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\ToolBar.java (line 56)     CONSTRUCTOR ToolBar.ToolBar[1]
C:\wsast\swing-drawing-app-master\src\ToolBar.java (line 78)     call      unresolved:javafx.swing.JToolBar.setBorder

C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 48)    CONSTRUCTOR DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\ToolBar.java (line 57)     CONSTRUCTOR ToolBar.ToolBar[1]
C:\wsast\swing-drawing-app-master\src\TextDialog.java (line 113)  call      unresolved:javafx.swing.JPanel.setBorder


C:\wsast\swing-drawing-app-master\src\DrawFrame.java (line 48)    CONSTRUCTOR DrawFrame.DrawFrame[1]
C:\wsast\swing-drawing-app-master\src\ToolBar.java (line 57)     CONSTRUCTOR ToolBar.ToolBar[1]
C:\wsast\swing-drawing-app-master\src\TextDialog.java (line 117)  call      unresolved:javafx.swing.JPanel.setBorder
```

- ▶ This can be useful to find paths between particular areas of the application such as request classes and auth or database methods

Usage – Dataflow Analysis (1)

- ▶ Dataflow analysis is initiated using the “dfscan” command
- ▶ Starting with each entrypoint (or based on provided filter), code flow is followed and each loaded source/sink is executed on each node encountered
- ▶ Context on the state at point of execution is provided to the source/sink and the state of variables, associated tags, and matched sources and sinks can be read/updated
- ▶ A sink can notify “subscribers” about any findings; subscribers can output findings into any format desired

Usage – Dataflow Analysis (2)

 Command Prompt - wSAST.exe interactive --sources=C:\wsast\examples\paint-master\src\paint\MainFrame.java --project=projects\paint-master

```
[DFA]      |-> unresolved:add()
[DFA]      |-> unresolved:add()
[DFA]      |-> unresolved:add()
[DFA]      |-> unresolved:add()
[DFA] |<-
[DFA] |-> paint.MainFrame.MainFrame[1]
[DFA]      |-> unresolved:setDefaultCloseOperation()
[DFA]      |-> unresolved:setSize()
[DFA]      |-> paint.MainFrame.unsafeCopy[1]
[DFA]      | code-simple
[DFA]      | DuplicatedIfElseCondition
[DFA]      |-> unresolved:bar()
[DFA]      |-> unresolved:memcpy()
[DFA]      | MemCpySink
[DFA]      | code-simple
[DFA]      | code-simple
[DFA]      |-> unresolved:foo()
[DFA]      |-> unresolved:memcpy()
[DFA]      | MemCpySink
[DFA]      | code-simple
[DFA]      | code-simple
```

Usage – Augmenting Analysis (1)

- ▶ For cases where source code is missing (application, framework or runtime) it is possible to augment the code using WSIL
- ▶ For example, if the source code for `javax.servlet.http.HttpServletRequest` is missing then dataflow tracing will not understand the types involved, for example that `getParameter()` returns a `java.lang.String` (or even that `String` is a “string type”)
- ▶ WSIL augmentation allows code to flow naturally through missing types, and more bugs to be found; it is a write once use repeatedly scenario
- ▶ This can also be used to link disparate code written in different languages together

Usage – Augmenting Analysis (2)

```
[DFA] |-> org.cysecurity.cspf.jv1.controller.EmailCheck.processRequest[1]
[DFA]   |-> unresolved:javax.servlet.http.HttpServletResponse.setContentType()
[DFA]   |-> unresolved:javax.servlet.http.HttpServletResponse.getWriter()
[DFA]   |-> unresolved:getServletContext()
[DFA]   |-> unresolved:<unknown>.getRealPath()
[DFA]   |-> unresolved:org.cysecurity.cspf.jv1.model.DBConnect.connect()
[DFA]   |-> unresolved:javax.servlet.http.HttpServletRequest.getParameter()
[DFA]   | getParameter
[DFA]   |-> unresolved:<unknown>.trim()
[DFA]   |-> unresolved:java.sql.Connection.isClosed()
[DFA]   |-> unresolved:java.sql.Connection.createStatement()
[DFA]   |-> unresolved:java.sql.Statement.executeQuery()
[DFA]   | executeQuery
```

```
[DFA] |-> org.cysecurity.cspf.jv1.controller.EmailCheck.processRequest[1]
[DFA]   |-> unresolved:javax.servlet.http.HttpServletResponse.setContentType()
[DFA]   |-> unresolved:javax.servlet.http.HttpServletResponse.getWriter()
[DFA]   |-> unresolved:getServletContext()
[DFA]   |-> unresolved:<unknown>.getRealPath()
[DFA]   |-> unresolved:org.cysecurity.cspf.jv1.model.DBConnect.connect()
[DFA]   |-> javax.servlet.http.HttpServletRequest.getParameter[1]
[DFA]   |<-
[DFA]   | getParameter
[DFA]   |-> String.trim[1]
[DFA]   |<-
[DFA]   |-> unresolved:java.sql.Connection.isClosed()
[DFA]   |-> unresolved:java.sql.Connection.createStatement()
[DFA]   |-> unresolved:java.sql.Statement.executeQuery()
[DFA]   | executeQuery
[DFA]   | code-simple
[DFA]   |-> unresolved:java.sql.ResultSet.next()
```

```
java.wsil x
1 namespace javax.servlet.http
2 {
3     class HttpServletRequest
4     {
5         String getParameter(String name)
6         {
7             return name;
8         }
9     }
10 }
11
12 class String
13 {
14     String trim()
15     {
16         return this;
17     }
18 }
```

Usage – Static Analysis (1)

- ▶ Static analysis scans are initiated with the “sscan” command and validate the entire project WSIL AST for issues
- ▶ They lack execution context but are great for quickly identifying bugs such as potential coding errors such as duplicated expressions, calls to unsafe functions (without parameter taint awareness), etc.
- ▶ Since these scans execute very quickly and demand limited resources, it is often a good idea to run these first and review the results while the dataflow scan is being performed
- ▶ Scans can result in a greater volume of false positives but the “traced-as-dynamic” config option helps

Usage – Static Analysis (2)

```
Command Prompt - wSAST.exe interactive --sources=c:\wsast\examples\paint-master --project=projects\paint-master
```

```
Loading static rule 'simplesyntactic' from assembly 'CommonRulesEngine.dll' ...
  Registering static rule 'DuplicatedIfElseCondition'
  Registering static rule 'AssignmentInsideCondition'
  Registering static rule 'DuplicatedLogicalExpression'
  Registering static rule 'DuplicatedTernaryExpression'
  Registering static rule 'UnexpectedPrecedenceTernaryExpression'
  Registering static rule 'PointerLessThanZero'
  Registering static rule 'StackAllocInsideLoop'
  Registering static rule 'SuspiciousEmptyStatement'
  Registering static rule 'SuspiciousNewSimpleType'
  Registering static rule 'VariadicFunctionCallNonPODParam'
  Registering static rule 'FunctionPointerAsCondition'
  Registering static rule 'SuspiciousAllocOfStrlen'
Loading static result subscriber 'simplesubs' from assembly 'CommonRulesEngine.dll' ...
Processing static result subscriber provider 'simplesubs' ...
  Registering static result subscriber 'code-simple'
Performing static analysis of discovered code ...
[SRE] [CONSTRUCTOR] MainFrame.MainFrame[1]
[SRE] MainFrame.MainFrame[1].[1] <- SuspiciousEmptyStatement
[SRE] [METHOD      ] MainFrame.unsafeCopy[1]
[SRE] MainFrame.unsafeCopy[1].[1] <- DuplicatedIfElseCondition
[SRE] MainFrame.unsafeCopy[1].[1].[2] <- MemCpyRule
[SRE] MainFrame.unsafeCopy[1].[1] <- MemCpyRule
[SRE] [METHOD      ] MainFrame.getMainFrame[1]
[SRE] [CONSTRUCTOR] MainFrame.MenuBar.MenuBar[1]
[SRE] [METHOD      ] MainFrame.MenuBar.MenuBarHandler01.actionPerformed[1]
[SRE] [METHOD      ] MainFrame.MenuBar.MenuBarHandler02.actionPerformed[1]
```

Output – Subscribers (1)

- ▶ Outputs for dataflow scans are “subscribers” which implement a simple interface to receive results
- ▶ Notifying subscribers is initiated by a sink and does not happen automatically (since some sinks are not supposed to report, just build context)
- ▶ Lists of variables captured at the time of execution of matched sinks, paths, and issue descriptions are reported
- ▶ Subscribers can output into any format desired; so far only a simple text output is currently provided

Output – Subscribers (2)

Title: SQL Injection - executeQuery()
Source-to-Sink: getParameter -> executeQuery
Category: ALL_CATEGORIES

Source Description: Retrieves a request parameter from the query string or request body.

Sink Description: Executes the specified SQL query on the database.

```
[C:\wsast\examples\JavaVulnerableLab-master\src\main\java\org\cysecurity\csp\jv1\controller\EmailCheck.java (line 42)]  
    String email=request.getParameter("email").trim();
```

```
[C:\wsast\examples\java-wsil-aug\java.wsil (line 5)]  
    String getParameter(String name)
```

```
[C:\wsast\examples\java-wsil-aug\java.wsil (line 7)]  
    return name;
```

```
[C:\wsast\examples\JavaVulnerableLab-master\src\main\java\org\cysecurity\csp\jv1\controller\EmailCheck.java (line 42)]  
    String email=request.getParameter("email").trim();
```

```
[C:\wsast\examples\java-wsil-aug\java.wsil (line 14)]  
    String trim()
```

```
[C:\wsast\examples\java-wsil-aug\java.wsil (line 16)]  
    return this;
```

```
[C:\wsast\examples\JavaVulnerableLab-master\src\main\java\org\cysecurity\csp\jv1\controller\EmailCheck.java (line 43)]  
    JSONObject json=new JSONObject();
```

```
[C:\wsast\examples\JavaVulnerableLab-master\src\main\java\org\cysecurity\csp\jv1\controller\EmailCheck.java (line 44)]  
    if(con!=null && !con.isClosed())
```

```
[C:\wsast\examples\JavaVulnerableLab-master\src\main\java\org\cysecurity\csp\jv1\controller\EmailCheck.java (line 46)]  
        ResultSet rs=null;
```

```
[C:\wsast\examples\JavaVulnerableLab-master\src\main\java\org\cysecurity\csp\jv1\controller\EmailCheck.java (line 47)]  
        Statement stmt = con.createStatement();
```

```
[C:\wsast\examples\JavaVulnerableLab-master\src\main\java\org\cysecurity\csp\jv1\controller\EmailCheck.java (line 48)]  
        rs=stmt.executeQuery("select * from users where email='"+email+"'");
```




Common Rules Engine

Common Rules – Function (1)

- ▶ wSAST comes with built-in support for simple function rules
- ▶ These rules match a function call on several criteria:
 - ▶ Function/method name
 - ▶ Parameter types and count/return value type
 - ▶ Containing class or namespace
 - ▶ Virtual (if so walks the class hierarchy if known)
- ▶ Additionally sink function rules can contain linked parameters, so a relationship between tainted parameters can be maintained and validated
- ▶ Sources can mark parameter variables are traced; sinks can validate traced status

Common Rules – Function (2)

javax.servlet.http.HttpServletRequest.xml

```
1 <function name="javax.servlet.http.HttpServletRequest.getParameter" languages="java" report="true"  
categories="*" description="Retrieves the value of a request parameter, potentially sourced from user  
input.">  
2     <signature prefix-types=".*?HttpServletRequest.*" virtual="true" names="getParameter" param-count="1" />  
3     <param pos="1" name="name" types=".*?String" traced="false" />  
4     <return types=".*?String" virtual="true" traced="true" />  
5 </function>
```

java.io.File.xml

```
1 <function name="java.io.File.File" languages="java" report="true" categories="*" title="Arbitrary File  
Creation" description="Creates a new File object without proper validation, potentially allowing unauthorized  
file access or manipulation.">  
2     <signature prefix-types=".*?File" virtual="true" names="File" param-count="1"/>  
3     <param pos="1" name="pathName" types=".*?String" traced="true" />  
4 </function>  
5
```

Common Rules – Variable (1)

- ▶ wSAST comes with built-in support for simple variable rules
- ▶ These rules can allow variables matching a specified regex to be marked as sources or sinks
- ▶ The class or namespace containing the variable can also be validated against
 - ▶ So for example: any variable of any class where the class name contains “model”, or any variable of any class where the variable name contains “password”
- ▶ The variable access can be specified (read or write, or both)
 - ▶ Typically sources would be read, sinks written
- ▶ The variables can be marked as traced and detected by any sink

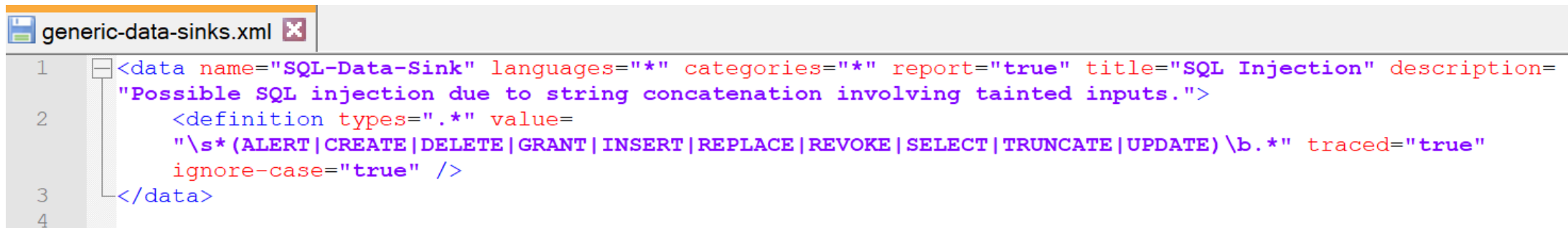
Common Rules – Variable (2)

```
generic-var-sources.xml
1 <variable name="Login-Related-Variable" languages="*" categories="*" description="Possible read of sensitive
2   login related variable." >
3   <definition types="*" virtual="false" names=".*?(auth|login|logout|verification|redirect|admin|user) .*"
4     access="read" traced="true" ignore-case="true" />
5 </variable>

generic-var-sinks.xml
16
17 <variable name="Security-Related-Variable" languages="*" categories="*" title="Possible Security Related
18   Variable" description="Possible write of sensitive security related variable." >
19   <definition types="*" virtual="false" names=".*?(backdoor|bypass|security|privilege|role|permission) .*"
20     access="write" traced="true" ignore-case="true" />
21 </variable>
```

Common Rules - Data

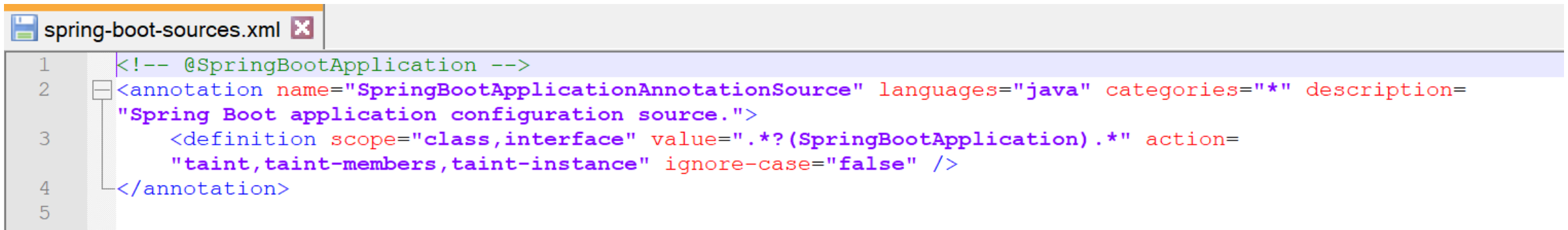
- ▶ wSAST comes with built-in support for simple data rules
- ▶ Data rules can be used to match specific literal data fragments within the code as sources or sinks (generally data rules would only be sinks)
- ▶ If the data fragment is matched and the input is traced a sink match occurs



```
1 <data name="SQL-Data-Sink" languages="*" categories="*" report="true" title="SQL Injection" description="Possible SQL injection due to string concatenation involving tainted inputs.">
2   <definition types="*" value="
3     \"s* (ALERT|CREATE|DELETE|GRANT|INSERT|REPLACE|REVOKE|SELECT|TRUNCATE|UPDATE) \b.*" traced="true"
4     ignore-case="true" />
5 </data>
```

Common Rules – Annotation

- ▶ wSAST comes with built-in support for simple annotation rules
- ▶ Annotation rules can be used to match specific class, method, parameter, variable etc. annotations and mark the resulting variable, instance or data members as traced
- ▶ The scope (code object) and action (what should be marked traced) of a matched annotation can be specified



```
spring-boot-sources.xml
1 <!-- @SpringBootApplication -->
2 <annotation name="SpringBootApplicationAnnotationSource" languages="java" categories="*" description=
  "Spring Boot application configuration source.">
3   <definition scope="class,interface" value=".*?(SpringBootApplication).*" action=
    "taint,taint-members,taint-instance" ignore-case="false" />
4 </annotation>
5
```



Extending wSAST

Technical – Dataflow Source/Sink (1)

- ▶ Sources and sinks can be provided by implementing a simple interface and adding entry to config XML; these are evaluated when the “dfscan” command is run
- ▶ Both are executed once per node on the local CFG during dataflow analysis
- ▶ Can attach and read arbitrary state to nodes, dataflow variables for extensibility
- ▶ Full path from entrypoint can be examined, matches sources attached to variables and matches sinks for an ordered list (and can go out of scope if needed)

Technical – Dataflow Source/Sink (2)

```
// WsilInterfaces, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// WsilHelper.Interfaces.ISource
using WsilHelper;

public interface ISource
{
    string GetSourceName();

    string[] GetSourceCategories();

    string GetDescription();

    bool GetMatchBeforeEval();

    bool IsMatch(WrappedDataflowEngine dfe, WrappedDataflowSource source, WrappedWsilNode node, WrappedDataflowStateIn state_in, WrappedDataflowStateOut dso);

    void DoAction(WrappedDataflowEngine dfe, WrappedDataflowSource source, WrappedWsilNode node, WrappedDataflowStateIn state_in, WrappedDataflowStateOut state_out);
}
```

```
// WsilInterfaces, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// WsilHelper.Interfaces.ISink
using WsilHelper;

public interface ISink
{
    string GetSinkName();

    string[] GetSinkCategories();

    string GetTitle();

    string GetDescription();

    bool GetMatchBeforeEval();

    bool IsMatch(WrappedDataflowEngine dfe, WrappedDataflowSink sink, WrappedWsilNode node, WrappedDataflowStateIn state_in, WrappedDataflowStateOut dso);

    void DoAction(WrappedDataflowEngine dfe, WrappedDataflowSink sink, WrappedWsilNode node, WrappedDataflowStateIn state_in, WrappedDataflowStateOut state_out);
}
```

Technical – Static Rules (1)

- ▶ For WSIL AST-only rules another interface can be implemented; these rules are evaluated when the “sscan” command is run
- ▶ These rules are executed without context (e.g. runtime state) and knowledge of variables, but can still be quite powerful as they can build up context internally
- ▶ The static rules are much faster to execute and are good for finding problem areas of code for manual review

Technical – Static Rules (2)

```
// WsilInterfaces, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// WsilHelper.Interfaces.IStaticRule
+ using ...

public interface IStaticRule
- {
    string GetRuleName();

    string GetRuleApplicableNodeType();

    string GetRuleApplicableNodeName();

    IStaticRuleResult PerformCheck(WrappedWsilNode node);
}
```

Technical – Language Support (1)

- ▶ Language support is extended by implementing a simple interface and adding entry to config XML
- ▶ The language processor must parse the original language (e.g. Java) and then use `WrappedWsilNodeBuilder` class to build IL tree
- ▶ Callbacks to notify the UI of progress

Technical – Language Support (2)

ILanguageProvider

```
// WsilInterfaces, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null  
// WsilHelper.Interfaces.ILanguageProvider  
using WsilHelper.Interfaces;
```

```
public interface ILanguageProvider  
{  
    void SetProjectName(string name);  
  
    void SetProjectPath(string path);  
  
    void SetFileExtensions(string[] extensions);  
  
    string[] GetFileExtensions();  
  
    void SetNumberOfParseThreads(int numberOfThreads);  
  
    int GetNumberOfParseThreads();  
  
    void SetFaultTolerantParse(bool tolerantParse);  
  
    bool GetFaultTolerantParse();  
  
    void SetParserCallback(ParserCallbackDelegate parserCallback);  
  
    ParserCallbackDelegate GetParserCallback();  
  
    void BeginParse(string[] paths);  
  
    void AbortParse();  
}
```

Future Plans

- ▶ Short term – add many more Java rules (GPT is excellent for this); WSIL shims for common Java frameworks including web, RPC, mobile
- ▶ Short/Mid term – add support for more CWE entries; common syntactical mistakes;
- ▶ Mid/Long term – add parsers and translators for more languages; more language constructs
- ▶ Long term – interactive IDE plugin (graphing, right click menu to create source/sink and initiate scan); paradigms for languages not yet supported (e.g. declarative, functional)

Demo

► Wish me luck...

Questions?



Contact

- ▶ Me
 - ▶ @peterwintrsmith on Twitter.
- ▶ wSAST Product & Development
 - ▶ <https://www.wsast.co.uk/>
 - ▶ @wsastsupport on Twitter
 - ▶ support@wsast.co.uk for tech support by email.