UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
14/764,280	07/29/2015	Kirill Mendelev	84312707	2826

146568 7590 11/18/2019
MICRO FOCUS LLC
500 Westover Drive
#12603
Sanford, NC 27330

EXAMINER	
SALEHI, HELAI	

ART UNIT	PAPER NUMBER
2433	

NOTIFICATION DATE	DELIVERY MODE
11/18/2019	ELECTRONIC

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

Notice of the Office communication was sent electronically on above-indicated "Notification Date" to the following e-mail address(es):

software.ip.mail@microfocus.com

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

Ex parte KIRILL MENDELEV, LU ZHAO, DAVID JOHN BABCOCK, and RONALD JOSEPH SECHMAN

Appeal 2018-007823
Application 14/764,280
Technology Center 2400

POTHIER, *Administrative Patent Judge.*

DECISION ON APPEAL

STATEMENT OF THE CASE

Pursuant to 35 U.S.C. § 134(a), Appellant[1,2] appeals from the Examiner's decision to reject claims 1–18. We have jurisdiction under 35 U.S.C. § 6(b). We reverse.

[1] Throughout this opinion, we refer to the Final Action (Final Act.) mailed November 30, 2017, the Appeal Brief (Appeal Br.) filed May 3, 2018, the Examiner's Answer (Ans.) mailed May 25, 2018, and the Reply Brief (Reply Br.) filed July 24, 2018.

[2] We use the word Appellant to refer to "applicant" as defined in 37 C.F.R. § 1.42(a). Appellant identifies the real party in interest as EntIT Software LLC. Appeal Br. 3.

CLAIMED SUBJECT MATTER

The claims are directed to scans used to identify security vulnerabilities in web applications. Spec. ¶ 1. The Specification explains:

> a dynamic security scan may identify security vulnerabilities by performing actual attacks such as a buffer overflow attack or a structured query language (SQL) injection attack on a web application and recording the results. However, dynamic security scans are black-box testing procedures that have no knowledge of the underlying source code of the web applications. Accordingly, the effectiveness of a dynamic security scan is typically determined based on the speed of the scan and the number of vulnerabilities that are identified.

Id. ¶ 9. In the above scan technique, "the number of vulnerabilities identified may depend more on the quality of the web application rather than the quality of the scan." *Id.* ¶ 10. The invention in this application determines an attack surface "coverage[,] using runtime analysis and static code analysis" (*id.* ¶ 11), which "allows the user to determine whether the web application is actually free of security vulnerabilities and whether the dynamic security scan is properly configured" (*id.* ¶ 12).

Claim 1 is reproduced below:

> A system comprising:
>
> a processor to:
>
> *perform a static code analysis of a web application to identify a plurality of data entry points for the web application*, wherein the plurality of data entry points are used to determine an attack surface size for the web application;
>
> *initiate runtime monitoring for a dynamic security scan of the web application*;
>
> *detect, by the runtime monitoring, invocation of a statement at one of the plurality of data entry points,* wherein the invocation is logged as an invocation entry

> that comprises invocation parameters and a timestamp; and
>
> determine an attack surface coverage of the dynamic security scan using the invocation entry and the attack surface size.

Appeal Br. 18 (Claims App'x) (emphases added).

REFERENCES

The prior art relied upon by the Examiner is:

Name	**Reference**	**Date**
Calendino	US 2010/0169974 A1	July 1, 2010
Podjarny	US 2011/0161486 A1	June 30, 2011

REJECTION OVER CALENDINO AND PODJARNY

The Examiner rejected claims 1–18 under 35 U.S.C. § 103(a) as unpatentable over Calendino and Podjarny. Final Act. 2–19. For independent claim 1, the Examiner finds Calendino teaches a processor that "perform[s] static code analysis of a web application to identify a plurality of data entry points for the web application" and "detect[s], by the runtime monitoring, invocation of a statement at one of the plurality of data entry points." *Id.* at 3–4 (citing Calendino ¶¶ 48, 50, 59). Similar findings are made for independent claims 7 and 12. *Id.* at 10–12, 15–17. The Examiner adds in the Answer that "[t]he run-time analyzer 320 does identify [a] plurality of data entry points for a web application by a static code analysis. The 'application inputs' [in Calendino] are a plurality of data entry points for a web application that are identified through a static code analysis." Ans. 21.

Among other arguments, Appellant contends run-time analyzer 320 "perform[s] a run-time analysis, not a static code analysis, on an executing application," which differs the recited "processor to: perform a static code analysis of a web application" in claim 1. Appeal Br. 8–9; *id.* at 10 (stating "[a] runtime analysis is not a static code analysis."); Reply Br. 1–2. Appellant further argues analyzer 320 does not identify the recited "data entry points for the web application" in claim 1 (Appeal Br. 9) and does not detect statement invocations at one of the identified data entry points (*id.* at 10). In the Reply Brief, Appellant also asserts the rejection relies on run-time analyzer 320 to teach both the recited processor to "perform a static code analysis" and "detect . . . an invocation of a statement" limitation. Reply Br. 2–3.

ISSUE

Under § 103, has the Examiner erred in rejecting claim 1 by finding that Calendino and Podjarny would have taught or suggested a processor to:

(1) "perform a static code analysis of a web application to identify a plurality of data entry points for the web application" and

(2) "detect, by the runtime monitoring, invocation of a statement at one of the plurality of data entry points"?

ANALYSIS

Based on the record before us, we find error in the Examiner's rejection of independent claims 1, 7, and 12. We begin with claim 1. Because Podjarny was not relied upon to teach the above-identified disputed recitations (see Final Act. 5), we confine our discussion to Calendino.

First, the Examiner's mapping to above-noted limitations is troublesome. Specifically, the Examiner has mapped the *same* passages in Calendino to *both* the recited "perform a static code analysis of a web application to identify a plurality of data entry points for the web application" function *and* "detect, by the runtime monitoring, invocation of a statement at one of the plurality of data entry points" function in claim 1. *Compare* Final Act. 3–4 (stating "run-time analyzer 320 monitors all potential inputs to web application 314 and citing Calendino ¶¶ 48, 50, 59), *with id.* at 4–5 (stating "run-time analyzer 320 monitors various inputs to web application 314" and citing Calendino ¶ 48).

However, "[w]here a claim lists elements separately, 'the clear implication of the claim language' is that those elements are 'distinct component[s]' of the patented invention." *Becton, Dickinson & Co. v. Tyco Healthcare Grp., LP*, 616 F.3d 1249, 1254 (Fed. Cir. 2010) (quoting *Gaus v. Conair Corp.*, 363 F.3d 1284, 1288 (Fed. Cir. 2004)); *see also CAE Screenplates Inc. v. Heinrich Fiedler GmbH*, 224 F.3d 1308, 1317 (Fed. Cir. 2000) ("In the absence of any evidence to the contrary, we must presume that the use of . . . different terms in the claims connotes different meanings."). Consistent with these guiding principles, the disputed functions in claim 1 should be treated as distinct functions. Yet, as Appellant indicates (*see* Reply Br. 2–3), the Examiner's mapping does not seem to treat these separately recited elements in claim 1 as distinct.

The Specification also describes "static code analysis" and "runtime monitoring" separately. See Spec. ¶¶ 17 (discussing "[s]tatic code analysis instructions 122 may perform static code analysis on source code of a web application"), 20 (discussing "[r]untime monitoring instructions 124 may initiate runtime monitoring of an executing web application during a dynamic security scan.”). Thus, the “perform” and “detect” functions in claim 1, when broadly and reasonably construed in light of the disclosure, should be interpreted as separate and distinct.

Second, we agree with Appellant that Calendino’s run-time analyzer 320 does not perform “a static code analysis” as claim 1 requires. Appeal Br. 8; *id.* at 10 (stating the run-time analyzer is performed “without a static code analysis”); Reply Br. 2 (stating the run-time analyzer 320 “perform[s] a dynamic analysis, and not a static code analysis.”) That is, we agree run-time analyzer 320 does not perform “a static code analysis” when considering this phrase’s broadest reasonable construction “in light of the specification as it would be interpreted by one of ordinary skill in the art” during prosecution. *In re Am. Acad. of Sci. Tech Ctr.*, 367 F.3d 1359, 1364 (Fed. Cir. 2004) (citation omitted).

Specifically, the Specification does not define the phrase “static code analysis,” but addresses “static code analysis” separately from “runtime monitoring” (e.g., a run-time analysis performed by run-time analyzer 320) as previously mentioned. See Spec. ¶¶ 11, 17, and 20. Moreover, when considering how the phrase “static code analysis” in the claims would be interpreted by an ordinary skilled artisan, we conclude run-time analyzer 320’s functions do not perform “static code analysis.” For example and consistent with the Specification, Veracode indicates a static analysis “is performed in a non-runtime environment” and typically “will inspect program code for all possible run-time behaviors and seek out coding flaws," whereas dynamic analysis "adopts the opposite approached and is executed while a program is in operation."[3] Another source states:

> Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program . . . In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution.[4]

Thus, the broadest reasonable construction of "a static code analysis" in light of the Specification as it would be interpreted by an ordinary skilled artisan would be an analysis of code performed in a fixed or non-runtime environment.

Turning to the Examiner's mapping, the "static code analysis" is performed by *run-time* analyzer 320. Final Act. 3 (citing Calendino ¶¶ 48, 50, 59). Calendino states analyzer 320 "is a software application capable of performing run-time analysis on a separately executing application" (Calendino ¶ 47) and "monitors the execution of web application 314" (*id.* ¶ 48). Thus, Calendino's analyzer 320 performs its testing and evaluation on *executing* applications or programs (e.g., dynamic analysis) rather than on fixed code or in a non-runtime environment (e.g., static code analysis).

---

[3] DuPaul, Neil, Static Testing vs. Dynamic Testing (Dec. 3, 2013, Updated Feb. 4, 2019) (accessed November 1, 2019), https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing; *see also* OWASP™ Foundation, Static Code Analysis (accessed November 1, 2019), https://www.owasp.org/index.php/Static_Code_Analysis.

[4] Rouse, Margaret, Definition: Static Analysis (Static Code Analysis) (accessed November 1, 2019), https://searchwindevelopment.techtarget.com/definition/static-analysis.

To be sure, analyzer 320 "monitors all potential inputs to web applications" (Calendino ¶ 48) and gathered information (*id*. ¶ 49) is sent to "feedback control 322," which "refines the tests run by black box scanner 318" (*id.* ¶ 50, *cited in* Final Act. 3). In this regard, the Examiner notes analyzer 320 determines which web application inputs are and are not exercised. Ans. 20–21 (citing Calendino ¶ 48); Calendino ¶ 59, *cited in* Final Act. 3. The Examiner also states subsequent runs by black-box scanner "will cover the previously unexercised application inputs" (Final Act. 3 (citing Calendino ¶ 50)), and "the black box scan is modified" (Ans. 20). But, the Examiner has not mapped black-box scanner 318 to the "perform a static code analysis" function in claim 1. Final Act. 3–4. In any event, Calendino's analyzer 320, not black-box scanner 318, "monitors all potential inputs to web application 314." Calendino ¶ 48. Nonetheless, as previously discussed, Calendino teaches the analyzer is a *run-time* analyzer, which does not perform "a static code analysis," as previously explained.

The Examiner further states that "[t]he run-time analyzer 320 does identify [a] plurality of data entry points for a web application by a static code analysis. The 'application inputs' [in Calendino] are a plurality of data entry points for a web application that are identified through a static code analysis." Ans. 21. Yet, the Examiner cites no passage in Calendino to support run-time analyzer 302 identifies data entry points using *static code analysis*. *See id.* Nor do we find the cited paragraphs address static code analysis. Final Act. 3–4 (citing Calendino ¶¶ 48–50, 59). At best, Calendino discusses "monitor[ing] all potential inputs" during run-time analysis but fails to explain sufficiently how the "all potential inputs" are located and even more so, fails to discuss these inputs are located *using static code analysis*. See Calendino ¶ 48.

The Examiner makes similar findings regarding Calendino for independent claims 7 and 12. Final Act. 10–12, 15–17. Independent claim 7 recites "[a] non-transitory machine-readable storage medium encoded with instructions that, when executed by a processor, cause the processor to: perform static code analysis of a web application to identify a reachable portion of source code of the web application" and "detect, by the runtime monitoring, invocation of a statement in the reachable portion of the source code." Appeal Br. 20 (Claims App'x). Although claim 7 differs from claim 1 in some respects, we find the Examiner erred in rejecting claim 7 for reasons similar to those above. Notably, the above "perform" function requires performing a static code analysis of a web application" similar to claim 1. Likewise, independent claim 12 recites a method that performs "a static code analysis to identify a plurality of data entry points for the web application" like claim 1 and to identify "a reachable portion of source code of the web application" like claim 7. Appeal Br. 21 (Claims App'x).

For the foregoing reasons, Appellant has persuaded us of error in the rejection of (1) independent claim 1 and 7, (2) independent claim 12, which recites commensurate limitations to claims 1 and 7, and (3) dependent claims 2–6, 8–11, and 13–18 for similar reasons.

DECISION SUMMARY

In summary:

Claims Rejected	**35 U.S.C. §**	**References**	**Affirmed**	**Reversed**
1–18	103(a)	Calendino, Podjarny		1–18
Overall Outcome				1–18

REVERSED

Notice of References Cited	Application/Control No. 14/764,280	Applicant(s)/Patent Under Patent Appeal No. 2018-007823	
	Examiner	Art Unit 2433	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-			
	B	US-			
	C	US-			
	D	US-			
	E	US-			
	F	US-			
	G	US-			
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

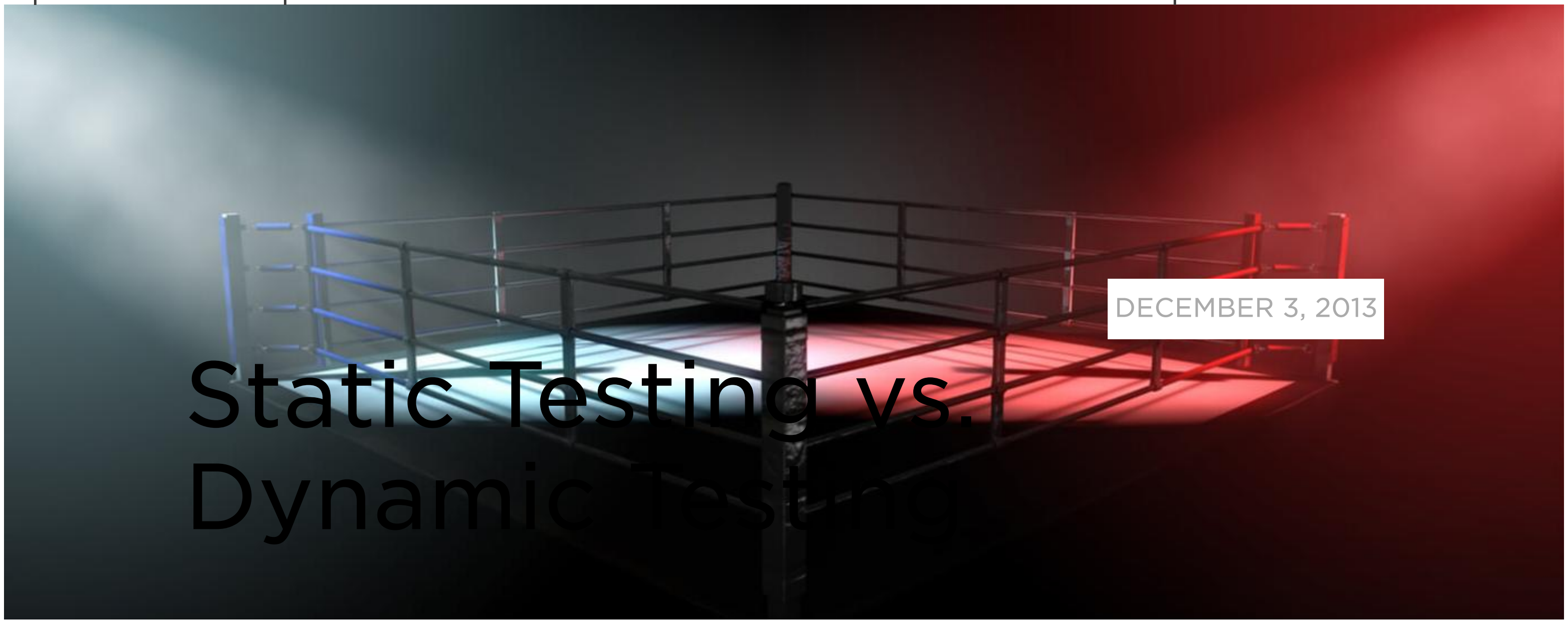
FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	DuPaul, Neil, Static Testing vs. Dynamic Testing (Dec. 3, 2013, Updated Feb. 4, 2019) (accessed November 1, 2019), https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing;
	V	OWASPTM Foundation, Static Code Analysis (accessed November 1, 2019), https://www.owasp.org/index.php/Static_Code_Analysis.
	W	Rouse, Margaret, Definition: Static Analysis (Static Code Analysis) (accessed November 1, 2019), https://searchwindevelopment.techtarget.com/definition/static-analysis.
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.



DECEMBER 3, 2013

Static Testing vs. Dynamic Testing

By Neil DuPaul (/blog/author/neil-dupaul)

SECURE DEVELOPMENT (/BLOG/CATEGORY/SECURE-DEVELOPMENT)

INTRO TO APPSEC (/BLOG/CATEGORY/INTRO-TO-APPSEC)

Share this article:

Updated: 2/4/2019

With reports of website vulnerabilities (//www.veracode.com/security/web-application-vulnerabilities) and data breaches regularly featured in the news, securing the software development life cycle (//www.veracode.com/security/secure-development-lifecycle) (SDLC) has never been so important. Enterprises must, therefore, choose carefully the correct security techniques to implement. Static (//www.veracode.com/products/binary-

Your Guide To Application Security Solutions

Learn best practices from the pros at Veracode.

GET THE HANDBOOK (HTTPS://INFO.VERACODE.COM/GUIDE-TO-APPLICATION

Static and Dynamic Analyses Explained

Static analysis (//www.veracode.com/products/binary-static-analysis-sast) is performed in a non-runtime environment. Typically, a static analysis tool will inspect program code for all possible run-time behaviors and seek out coding flaws (//www.veracode.com/security/application-vulnerability), back doors (//www.veracode.com/new-taxonomy-application-backdoors-veracode), and potentially malicious code (//www.veracode.com/security/malicious-code). Dynamic analysis adopts the opposite approach and is executed while a program is in operation. A dynamic test will monitor system memory, functional behavior, response time, and overall performance of the system. This method is not wholly dissimilar to the manner in which a malicious third party may interact with an application. Having originated and evolved separately, static and dynamic

LEARN MORE (HTTPS://WWW.VERACODE.COM/PRODUCTS/BINARY-STATIC-AN

Strengths and Weaknesses of Static and Dynamic Analyses

Static analysis, with its whitebox visibility, is certainly the more thorough approach and may also prove more cost-efficient with the ability to detect bugs at an early phase of the software development life cycle. For example, if an error is spotted at a review meeting or a desk-check – both types of static analysis – it can be relatively cheap to remedy. Had the error become lodged in the system, costs would multiply. Static analysis can also unearth future errors that would not emerge in a dynamic test. Dynamic analysis, on the other hand, is capable of exposing a subtle flaw or vulnerability too complicated for static analysis alone to reveal and can also be the more expedient method of testing. A dynamic test, however, will only find defects in the part of the code that is actually executed. The enterprise must weigh up these considerations with the complexities of their own situation in mind. Application type, time, and company resources are some of the primary concerns. The level of technical debt (//www.veracode.com/blog/2011/02/application-security-debt-and-application-interest-rates/) the enterprise is willing to take on may also be measured. A

Veracode's Application Security Platform (//www.veracode.com/products) features both Static and Dynamic scanning methods (//www.veracode.com/security/web-vuln-scanner), along with a variety of other features. See how we consolidate all of these tools into one centralized platform by filling out the form below.

When to Automate Application Security Testing

While static and dynamic analysis can be performed manually they can also be automated. Used wisely, automated tools can dramatically improve the return on testing investment. Automated testing tools (//www.veracode.com/security/automated-code-testing) are an ideal option in certain situations. For example, automation may be used to test a system's reaction to a heavy volume of users or to confirm a bug fix works as expected. It also helps to automate tests that are run on a regular basis during the SDLC. As

STAY UP TO DATE ON **APPLICATION SECURITY**

SEND ME UPDATES

Enter your email address

About Veracode Dynamic Analysis

LEARN MORE (HTTPS://WWW.VERACODE.COM/PRODUCTS/DYNAMIC-ANALYS

Related Content

Live From Gartner Security & Risk Mgmt Summit: Pair Security Trainings With... (/blog/security-news/live-gartner-security-risk-mgmt-summit-pair-security-trainings-technical-controls)

JUN 18, 2019

Customer Success Story: Why CAP COM Chose Veracode (/blog/managing-appsec/customer-success-story-why-cap-com-chose-veracode)

APR 18, 2018

STAY UP TO DATE
ON **APPLICATION SECURITY**

SEND ME UPDATES

Enter your email address

FEB 01, 2018

AppSec in Review Podcast: How Developers Respond to Security Findings (/blog/managing-appsec/appsec-review-podcast-how-developers-respond-security-findings)

DEC 05, 2017

OWASP Top 10 Updated for 2017: Here's What You Need to Know (/blog/security-news/owasp-top-10-updated-2017-here%E2%80%99s-what-you-need-know)

NOV 20, 2017

By Neil DuPaul (/blog/author/neil-dupaul)

Neil is a Marketing Technologist working on the Content and Corporate teams at Veracode. He currently focuses on Developer Awareness through strategic content creation. In his spare time you'll find him doting over his lovely wife and daughter. He is a Co-Owner of CrossFit Amoskeag in Bedford NH, his favorite topic is artificial intelligence, and his favorite food is pepperoni pizza.

Static Code Analysis

From OWASP

Every **control** should follow this template.

This is a control. To view all control, please see the Control Category page.

Last revision (mm/dd/yy): **10/16/2019**

- 1 Description
- 2 Techniques
  - 2.1 Data Flow Analysis
  - 2.2 Control Flow Graph (CFG)
  - 2.3 Taint Analysis
  - 2.4 Lexical Analysis
- 3 Strengths and Weaknesses
  - 3.1 Strengths
  - 3.2 Weaknesses
- 4 Limitations
  - 4.1 False Positives
  - 4.2 False Negatives
- 5 Important Selection Criteria
- 6 Examples
  - 6.1 RIPS PHP Static Code Analysis Tool
  - 6.2 OWASP LAPSE+ Static Code Analysis Tool
- 7 Tools
  - 7.1 OWASP Tools
  - 7.2 Open Source/Free
  - 7.3 Commercial
  - 7.4 Other Tool Lists
- 8 References
- 9 Further Reading

Description

Static Code Analysis (also known as Source Code Analysis) is usually performed as part of a Code Review (also known as white-box testing) and is carried out at the Implementation phase of a Security Development Lifecycle (SDL). Static Code Analysis commonly refers to the running of Static Code Analysis tools that attempt to highlight possible vulnerabilities within 'static' (non-running) source code by using techniques such as Taint Analysis and Data Flow Analysis.

Ideally, such tools would automatically find security flaws with a high degree of confidence that what is found is indeed a flaw. However, this is beyond the state of the art for many types of application security flaws. Thus, such tools frequently serve as aids for an analyst to help them zero in on security relevant portions of code so they can find flaws more efficiently, rather than a tool that simply finds flaws automatically.

Some tools are starting to move into the Integrated Development Environment (IDE). For the types of problems that can be detected during the software development phase itself, this is a powerful phase within the development lifecycle to employ such tools, as it provides immediate feedback to the developer on issues they might be introducing into the code during code development itself. This immediate feedback is very useful as compared to finding vulnerabilities much later in the development cycle.

The UK Defense Standard 00-55 requires that Static Code Analysis be used on all 'safety related software in defense equipment'.[0]

Techniques

There are various techniques to analyze static source code for potential vulnerabilities that maybe combined into one solution. These techniques are often derived from compiler technologies.

Data Flow Analysis

Data flow analysis is used to collect run-time (dynamic) information about data in software while it is in a static state (Wögerer, 2005).

There are three common terms used in data flow analysis, basic block (the code), Control Flow Analysis (the flow of data) and Control Flow Path (the path the data takes):

Basic block: A sequence of consecutive instructions where control enters at the beginning of a block, control leaves at the end of a block and the block cannot halt or branch out except at its end (Wögerer, 2005).

Example PHP basic block:

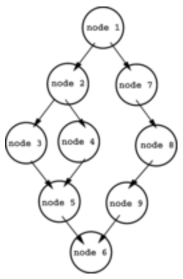
```
$a = 0;
$b = 1;

if ($a == $b)
{ # start of block
  echo “a and b are the same”;
} # end of block
else
{ # start of block
10. echo “a and b are different”;
11.} # end of block
```

Control Flow Graph (CFG)

An abstract graph representation of software by use of nodes that represent basic blocks. A node in a graph represents a block; directed edges are used to represent jumps (paths) from one block to another. If a node only has an exit edge, this is known as an ‘entry’ block, if a node only has a entry edge, this is know as an ‘exit’ block (Wögerer, 2005).

Example Control Flow Graph; ‘node 1’ represents the entry block and ‘node 6’ represents the exit block.



Taint Analysis

Taint Analysis attempts to identify variables that have been 'tainted' with user controllable input and traces them to possible vulnerable functions also known as a 'sink'. If the tainted variable gets passed to a sink without first being sanitized it is flagged as a vulnerability.

Some programming languages such as Perl and Ruby have Taint Checking built into them and enabled in certain situations such as accepting data via CGI.

Lexical Analysis

Lexical Analysis converts source code syntax into ‘tokens’ of information in an attempt to abstract the source code and make it easier to manipulate (Sotirov, 2005).

Pre tokenised PHP source code:

```
<?php $name = "Ryan"; ?>
```

Post tokenised PHP source code:

```
T_OPEN_TAG
T_VARIABLE
=
T_CONSTANT_ENCAPSED_STRING
;
T_CLOSE_TAG
```

Strengths and Weaknesses

Strengths

- Scales Well (Can be run on lots of software, and can be repeatedly (like in nightly builds))
- For things that such tools can automatically find with high confidence, such as buffer overflows, SQL Injection Flaws, etc. they are great.

Weaknesses

- Many types of security vulnerabilities are very difficult to find automatically, such as authentication problems, access control issues, insecure use of cryptography, etc. The current state of the art only allows such tools to automatically find a relatively small percentage of application security flaws. Tools of this type are getting better, however.
- High numbers of false positives.
- Frequently can't find configuration issues, since they are not represented in the code.
- Difficult to 'prove' that an identified security issue is an actual vulnerability.
- Many of these tools have difficulty analyzing code that can't be compiled. Analysts frequently can't compile code because they don't have the right libraries, all the compilation instructions, all the code, etc.

Limitations

False Positives

A static code analysis tool will often produce false positive results where the tool reports a possible vulnerability that in fact is not. This often occurs because the tool cannot be sure of the integrity and security of data as it flows through the application from input to output.

False positive results might be reported when analysing an application that interacts with closed source components or external systems because without the source code it is impossible to trace the flow of data in the external system and hence ensure the integrity and security of the data.

False Negatives

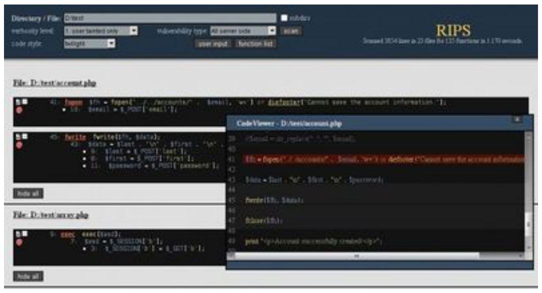
The use of static code analysis tools can also result in false negative results where vulnerabilities result but the tool does not report them. This might occur if a new vulnerability is discovered in an external component or if the analysis tool has no knowledge of the runtime environment and whether it is configured securely.

Important Selection Criteria

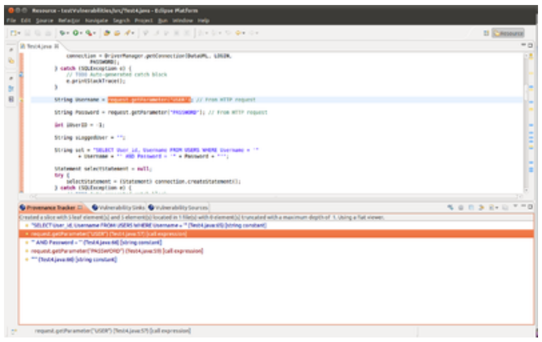
- Requirement: Must support your language, but not usually a key factor once it does.
- Types of Vulnerabilities it can detect (The OWASP Top Ten?) (more?)
- Does it require a fully buildable set of source?
- Can it run against binaries instead of source?
- Can it be integrated into the developer's IDE?
- License cost for the tool. (Some are sold per user, per org, per app, per line of code analyzed. Consulting licenses are frequently different than end user licenses.)
- Does it support Object-oriented programming (OOP)?

Examples

RIPS PHP Static Code Analysis Tool



OWASP LAPSE+ Static Code Analysis Tool


File Edit Source Refactor Navigate Search Project Run Window Help
Provenance Tracker
Vulnerability Sinks
Vulnerability Sources

Tools

OWASP Tools

Software	Language(s)
OWASP Code Crawler	.NET, Java
OWASP Orizon Project	Java
OWASP LAPSE Project	Java
OWASP O2 Platform	
OWASP WAP-Web Application Protection	PHP

Open Source/Free

Software	Language(s)	OS(es)
Agnitio (https://sourceforge.net/projects/agnitiotool/)	ASP, ASP.NET, C#, Java, Javascript, Perl, PHP, Python, Ruby, VB.NET, XML	Windows
Brakeman (https://brakemanscanner.org/)	Ruby, Rails	
Google CodeSearchDiggity (https://www.bishopfox.com/resources/tools/google-hacking-diggity/attack-tools/)	Multiple	
DevBug (http://www.devbug.co.uk)	PHP	web-based
FindBugs (http://findbugs.sourceforge.net/)	Java	
SpotBugs (https://spotbugs.github.io/) (Successor of FindBugs)	Java	
Find Security Bugs (https://find-sec-bugs.github.io/)	Java, Scala, Groovy	
FlawFinder (https://dwheeler.com/flawfinder/)	C, C++	
Microsoft FxCop (https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-3.0/bb429476(v=vs.80))	.NET	
.NET Security Guard (https://security-code-scan.github.io/)	.NET, C#, VB.net	
phpcs-security-audit (https://github.com/FloeDesignTechnologies/phpcs-security-audit)	PHP	Windows, Unix
PMD (https://pmd.github.io/)	Java, JavaScript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity, XML, XSL	
Puma Scan (https://www.pumascan.com/)	.NET, C#	
Microsoft PREFast (https://docs.microsoft.com/en-us/previous-versions/windows/embedded/ms933794(v=msdn.10))	C, C++	
RIPS Open Source (http://rips-scanner.sourceforge.net/)	PHP	any
SonarCloud (https://sonarcloud.io/about)	ABAP, C, C++, Objective-C, COBOL, C#, CSS, Flex, Go, HTML, Java, Javascript, Kotlin, PHP, PL/I, PL/SQL, Python, RPG, Ruby, Swift, T-SQL, TypeScript, VB6, VB, XML	
Splint (https://www.splint.org/)	C	
VisualCodeGrepper (https://sourceforge.net/projects/visualcodegrepp/)	C/C++, C#, VB, PHP, Java, PL/SQL	Windows

Commercial

Software
RIPS (https://www.ripstech.com/product/)
Fortify (https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview)
Veracode (https://www.veracode.com/)
CodeSonar (https://www.grammatech.com/)
ParaSoft (https://www.parasoft.com/)
~~Armorize CodeSecure (http://www.armorize.com/codesecure/)~~
Checkmarx Static Code Analysis (https://www.checkmarx.com/)
IBM AppScan Source (https://www.ibm.com/us-en/marketplace/ibm-appscan-source)
Coverity (https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html)
PVS-Studio (https://www.viva64.com/en/pvs-studio/)
Puma Scan Professional (https://pumascan.com/pricing/)
Klocwork (https://www.roguewave.com/products-services/klocwork/static-code-analysis)
Polyspace Static Analysis (https://www.mathworks.com/products/polyspace.html)
CodeSec (http://www.seczone.cn/2018/06/27/codesec%E6%BA%90%E4%BB%A3%E7%A0%81%E5%AE%89%E5%85%A8%E6%A3%80%E6%B5%8B%E5%B9%B3%E5%8F
Xanitizer (http://www.xanitizer.net)

Other Tool Lists

- NIST - Source Code Security Analyzers (http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html)

- Wikipedia - List of tools for static code analysis (http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)

References

[0] "Requirements for Safety Related Software in Defence Equipment" (http://www.software-supportability.org/Docs/00-55_Part_2.pdf) (pdf). Ministry of Defence. August 1, 1997. http://www.software-supportability.org/Docs/00-55_Part_2.pdf.

Further Reading

- OWASP Code Review Guide v1.1 (https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf)
- http://www.crosstalkonline.org/storage/issue-archives/2003/200311/200311-German.pdf
- http://www.ida.liu.se/~TDDC90/papers/industrial95.pdf
- http://www.php-security.org/downloads/rips.pdf
- http://www.seclab.tuwien.ac.at/papers/pixy.pdf

Retrieved from "https://www.owasp.org/index.php?title=Static_Code_Analysis&oldid=255503"

Categories: OWASP ASDR Project | Pages containing cite templates with deprecated parameters | FIXME | Control

---

DEFINITION

static analysis (static code analysis)

Posted by: Margaret Rouse WhatIs.com

Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension.

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Nevertheless, static analysis is only a first step in a comprehensive software quality-control regime. After static analysis has been done, dynamic analysis is often performed in an effort to uncover subtle defects or vulnerabilities. In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution. Static and dynamic analysis, considered together, are sometimes referred to as glass-box testing.

This was last updated in November 2006

Continue Reading About static analysis (static code analysis)

- The Multilingual Information Processing Unit at the University of Geneva (Switzerland) outlines the basics of static and dynamic analysis.

Related Terms

IronPython

IronPython is an altered version of the Python programming language that runs on top of Microsoft's .NET Framework. The language ... See complete definition

IronRuby

IronRuby is a version of the Ruby programming language developed for Microsoft Common Language Runtime (CLR), which is part of ... See complete definition

monad

A monad is: 1) A type of functor used in category theory in mathematics. 2) A kind of constructor used in functional ... See complete definition

Dig Deeper on Dynamic .NET programming languages

dynamic and static

static testing

Static routing

Statically focus a control

Dynamic and static routing

Configure backup static routes

Start the conversation

Share your comment

✓ Send me notifications when other members comment.

Add My Comment

-ADS BY GOOGLE

GitLab
10 Steps to Secure Your Next-Generation Software
Get the report
O'REILLY
10 Steps Every CISO Should Take to Secure Next-Gen Software
Cindy Blake
REPORT

CLOUD COMPUTING SOFTWARE QUALITY JAVA CLOUD APPLICATIONS

SearchCloudComputing

Licensing changes reveal Microsoft's cloud strategy

IT analysts share insights on the Microsoft licensing updates, and what they say about the on-premises giant's push for Azure ...

Microsoft JEDI contract leaves AWS to weigh its options

Losing the $10 billion JEDI cloud computing contract to Microsoft forces AWS to consider its next steps in trying to reverse the ...