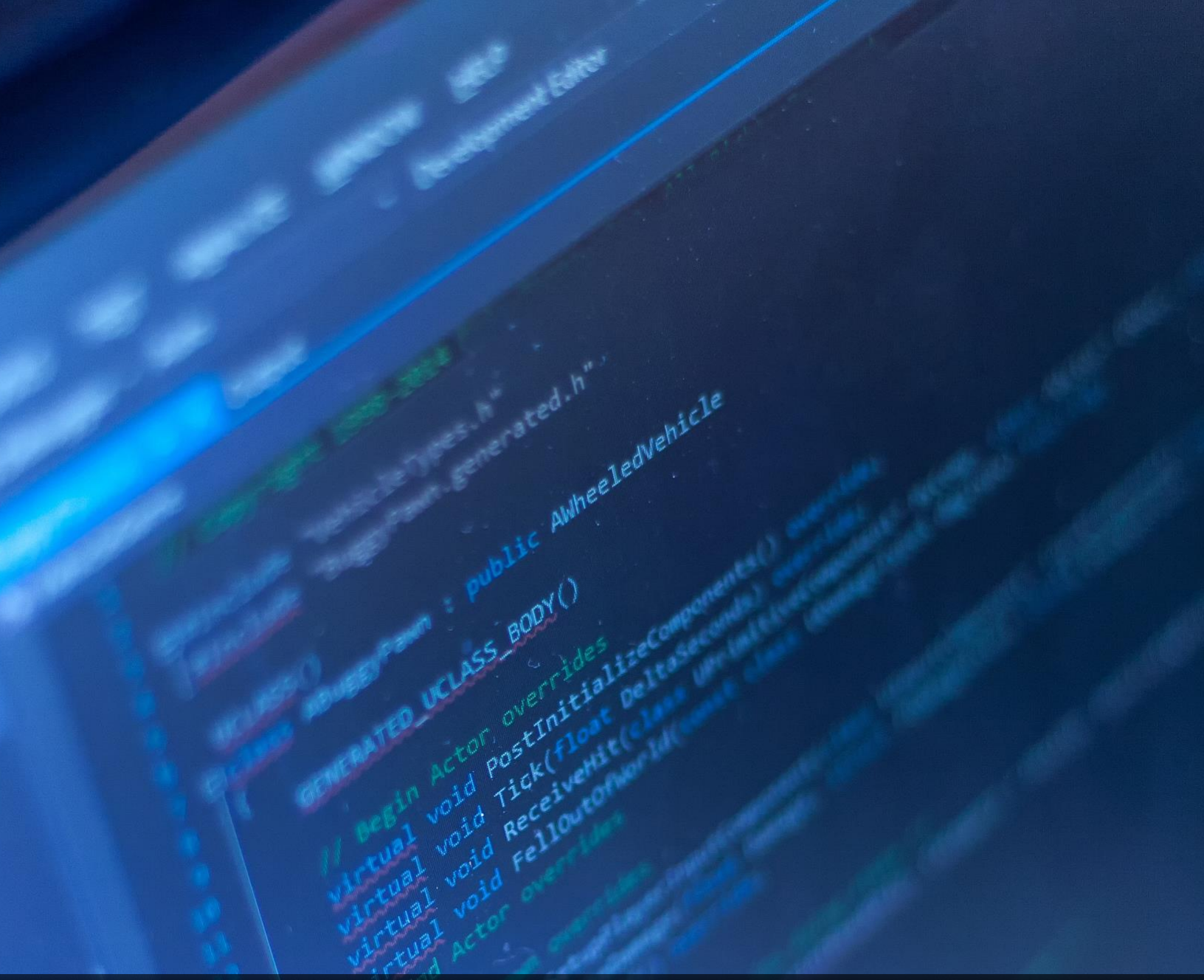


SECURITY COMPASS RESEARCH WHITEPAPER

# Gap Analysis of Code Scanners

## Exploring Solutions for the Problem of False Negatives



Security Compass

# INTRODUCTION

Static analysis is a key part of conducting secure software development today. Catching errors before deploying into a production environment can help reduce cost and improve quality. There are many different types of static analysis tools. For the purpose of this paper, we are focused on the subset of tools called Static Application Security Testing (SAST) tools. These tools are useful in the development stage where code is analyzed to identify security vulnerabilities early in the life cycle.

Many organizations feel that the exclusive use of code scanners is sufficient to ensure security and compliance of their applications. As this paper will show, this is not the case. In fact, there are many categories of problems that static analyzers are not capable of addressing. As (Chess 2007) explains, "All static analysis tools are guaranteed to produce some false positives or some false negatives." (Zhioua, 2014) agrees that "...these tools fall short in verifying the adherence of the developed software to application and organization security requirements." We intend to show the various categories of issues scanners cannot catch.

Our goal with this research is not to downplay the importance of code scanners; rather, our aim is to continually educate software developers, managers, directors, and CISOs about how scanners alone cannot catch all vulnerabilities. We show that scanners need complementary tools and/or processes at a higher level of abstraction in order to overcome the problem of false alarms by generated by scanners. Some of these complementary tools and processes include threat modeling, automated requirements management through policy-to-procedure platforms, and Software Development Life Cycle (SDLC) integrations.

# HOW SCANNERS WORK

Before attempting to analyze the gaps with scanners, it is important to understand how they work. In a nutshell, they all work in a similar way. As Chess aptly summarizes, "They all accept code, build a model that represents the program, analyze that model in combination with a body of security knowledge, and finish by presenting their results back to the user" (Chess 2007). Based on the work of (Zhioua, 2014) here are some common techniques used in scanners:

- Data flow analysis
- Control flow analysis
- Symbolic analysis
- Taint analysis

Based on a series of logical security tests, a scanner will produce a result to indicate whether or not the test fails (i.e., whether or not a vulnerability exists in the code). This may or may not correspond to the truth. If the scanner predicts the actual state of the code, it leads to a correct prediction. In those cases where the scanner did not catch an actual error, this is referred to as a *false negative*. If the scanner identifies an error where none actually exists, it is a *false positive*. The goal of a scanner is to minimize both false positives and false negatives.

In essence, we have four possibilities:

| | | **Actual State of Vulnerability in Code** | |
|---|---|---|---|
| | | **TRUE** | **FALSE** |
| **Scanner Predicted Result** | **Scanner identified vulnerability** | Correct prediction | False positive (wrong prediction) |
| | **Scanner did not identify vulnerability** | False negative (wrong prediction) | Correct prediction |

## Data-flow analysis

The technique used by scanners traces the flow of data through the code. "It aims at representing data dependencies in the source code and allows [you] to track the effect of input data" (Zhioua 2014). Using a symbol table, it is possible to determine whether the values for a given data variable are consistent within the bounds of that type. Any type mismatch or out of bounds exceptions can be caught.

When tracking the flow of data through code, there is a potential gap with programmer intent. For example, (Aspinall, 2016) suggests the following code snippet will generate a false positive around "possible loss of precision":

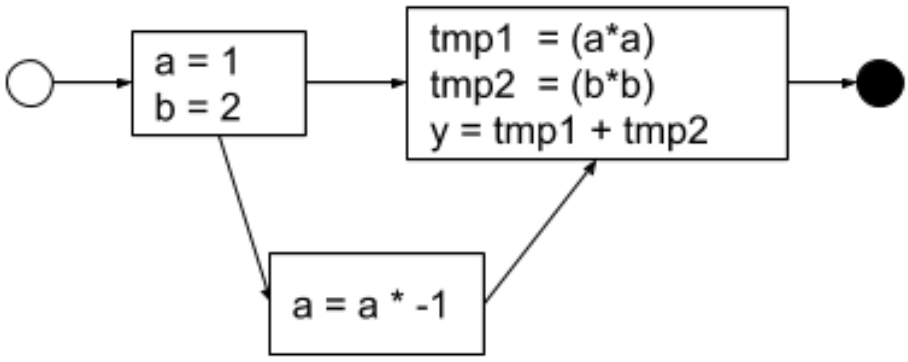
```
short s = 0;
int i = s;
short r = i;
```

Here, reassigning the value of *s* into an integer variable trips the scanner into thinking the value cannot be assigned to a short data type subsequently.

## Control-flow analysis

Keeping an application in a predictable state is an important attribute of secure software. Using a directed graph, it is possible to determine the program flow across various modules within the application. The goal is to ensure each transition is accounted for and leaves the application in a predictable state. "A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate" (Parasoft 2011).

A sample CFG looks something like the following:




The CFG can be constructed using the combination of an Abstract Syntax Tree and adding control flow information.

In today's event driven environments, the flow of control across an application grows significantly with each new module or component. Best practices in software development encourage modular programming and some categories of applications such as microservices encourage the use of hundreds of components. Working through each of these flows can be very time consuming and prevent teams from achieving their target of fast delivery. In discussing Control Flow Analysis, (Barbosa, 2012) explain "In general, the problem of discovering all the possible execution paths of a code is *undecidable*. (cf. *Halting problem*)". Elsewhere, (Ernst 2003) explain that "Because there are many possible executions, ...[static analysis]...must keep track of multiple different possible states. It is usually not reasonable to consider every possible run-time state of the program; for example, there may be arbitrarily many different user inputs or states of the runtime heap."

In cases where code cedes control to a third party module in which code is not available, scanners will not be able to address the security risk. "Sometimes there is clear evidence of a vulnerability in the HTTP response, such as a cross-site script in the HTML. But usually the evidence is less clear, such as when you send a SQL injection attack and the response is a 500 error. Was there a SQL injection?  Or did the application just break?" (Williams, 2015)

This problem is further complicated when the third party uses custom objects. "Even if you know that the request is JSON or XML and you have some kind of schema for the API, it's still exceptionally difficult to provide the right data to automatically invoke an API correctly...Applications using Google Web Toolkit (GWT), for example, has its own custom syntax and can't be scanned without a custom scanner" (Williams 2015).

## Symbolic analysis

This technique translates the code into a model that can be analyzed mathematically for correctness. "Symbolic analysis is deemed to be useful in transforming unpredictable loops to predictable sequences, and is mainly used for code optimization..." (Zhioua 2014).

While symbolic analysis can help address code complexity, it does not account for compiler optimization (which we will discuss later). Even if a particular implementation of symbolic analysis can help reduce security vulnerabilities, the compiler used to construct a production build may have a completely different set of optimizations thereby leading to false positives and negatives.

Some organizations try to mitigate this risk by running bytecode analysis instead. (Masood 2015) explains this position, "For the modern IL (intermediate bytecode) based languages, most of the new static analysis tools perform the byte code analysis instead of looking at the source files directly. This approach leverages compiler optimizations, and provides best of the both worlds." However, as (Curran 2016) points out, this has some consequences:

| | Static Code Analysis | Bytecode Analysis |
|---|---|---|
| ***Detection*** | Fast feedback for developers as they code | Late stage detection only once code is compiled |
| | Developers mitigate risks on the go | Compiler optimization may hurt results |
| ***Remediation*** | Easy access to the source code | Vulnerabilities must be prioritized and then sent back to developers for mitigation |
| ***Time consumption*** | Minimal | High |
| | Incremental scanning can save 80% of time | All the code must be scanned |
| ***ROI*** | High | Low |
| | Saves time, money and educates developers | Time consuming, may not detect important findings (hardcoded passwords etc.) |

Bytecode analysis is a late stage detection strategy which can make regression scaling difficult. It also introduces the possibility of additional false negatives when it comes to hardcoded passwords.

### Taint analysis

Taint Analysis attempts to take advantage of the data flow from user input into vulnerable functions. From a security perspective, the user input is ideally sanitized before reaching these vulnerable functions. By tainting the user input, a trace is done to determine the impact on these vulnerable functions. If the vulnerable function is impacted from a security perspective, the variable is flagged.

## WHY A SCANNER WILL ALWAYS GENERATE FALSE POSITIVES AND NEGATIVES

1. **It is highly unlikely for someone to create a static analyzer that catches all known security vulnerabilities.**

   In computability theory, there is something called the Halting problem. In a perfect world, we are able to execute a program from the start state through the termination (or Halt) stage. As software becomes more complex, this becomes non-deterministic. According to Rice's Theorem, any non-trivial semantic properties of programs are undecidable. The assumption made by scanners, however, is that we can execute code from start to finish predictably.

   Some examples of non-halting problems are:

   - Array out of bounds: when testing whether an index extends beyond an allocated buffer size, the test itself will have to extend beyond the buffer to prove true. Therefore the program will actually not Halt but still continue to run.
   - SQL string comes from tainted source: taint analysis tries to track an input value to a vulnerable function. “Because they use a binary classification for data (tainted or untainted), they classify functions as either being sanitizers...or being security irrelevant. Because the policy that these techniques check is context-agnostic, it cannot guarantee the absence...” (Wassermann 2007).
   - Is pointer used after it is freed?
   - Is accessing a variable the source of race condition?

As a result static analyzers will have some of the following problems:

1. Nontermination (the inability to accurately predict that code execution will end as expected)
2. False alarms (false positives)
3. Missed errors (false negatives)

All static analyzers sit somewhere between sound and complete. (Blackshear 2012) clarifies that "A *sound* static analysis overapproximates the behaviors of the program. A sound static analyzer is guaranteed to identify all violations..but may also report some 'false alarms'...A *complete* static analysis underapproximates the behaviors of the program...there is no guarantee that all actual violations..will be reported". In an effort to be more sound, the internal rules engines try to catch every possibility, leading to a lot of noise for developers to sift through. On the other hand, emphasizing completeness will not yield a complete picture of the security of the application.

**2. Scanners are typically optimized for a certain class of vulnerabilities.**

Not all scanners are created to catch the same category of vulnerabilities. Some are focused at the syntax level while others perform a detailed model analysis to try and derive data and flow information. Because of this, some organizations have opted to use multiple scanners to try and fill the gaps. This creates a problem, however, because a higher level scanner might falsely report an issue that a lower level scanner understands to be valid. "Each of these tools is focused on specific security properties and requires human intervention to different degrees" (Zhioua, 2014).

As a case study, (Ye, 2016) demonstrates the results of detecting and fixing a buffer overflow error (in particular, notice the column indicating the False Negative rate):

| Scanner | # Identified bugs | False Negative rate | # Identified fixes | False Positive rate |
|---|---|---|---|---|
| Scanner A | 19 | 68.3% (41/60) | 13 | 31.6% (6/19) |
| Scanner B | 32 | 68.0% (68/60) | 8 | 75.0% (24/32) |

| | | | | |
|---|---|---|---|---|
| Scanner C | 10 | 56.5% (13/23) | 0 | 100.0% (10/10) |
| Scanner A + Scanner B | 42 | 58.0% (58/100) | 14 | 66.7% (28/42) |
| Scanner B + Scanner C | 39 | 61.0% (61/100) | 7 | 82.1% (32/39) |
| Scanner A + Scanner C | 26 | 59.4% (38/64) | 13 | 50.0% (13/26) |
| All | 47 | 53.0% (53/100) | 13 | 72.3% (34/47) |

In this study, the  results suggest that scanners missed over half of the buffer overflow errors even when used in combination with each other.  This leads us to believe that, while they have a useful role in identifying application security gaps, scanners cannot be the only activity used to identify security vulnerabilities.  A recent survey by (Security Compass, 2017) revealed that static analysis was, on average, one of the top 3 security activities performed by large organizations.  This represents a significant gap between reality and execution.  As an example of the ubiquity of this security vulnerability, at the time of this writing, a simple search for "buffer overflow" in the MITRE Common Vulnerabilities and Exposures (CVE) List returned over 8,300 results.  A search for "sql injection" (the top category in OWASP's Top 10 list) returned less than 6,800 results.

Scanners face a tradeoff between speed and accuracy.  This has an impact on the result set of identified security vulnerabilities.  For example, a scanner that performs only syntax analysis and does not rely on compiled code will perform faster.  Since its accuracy will be based on pattern detection, it might result in a lot of false positives.  From a probability perspective, having a lot of false positives can imply greater accuracy but the manual cost of sifting through the results can be cost prohibitive.  Therefore, in an effort to reduce the number of false positives, scanners need to trade off greater accuracy.

3. **Compiler optimization can inject security vulnerabilities.**

Static analyzers examine code during the development phase. Compiling code can introduce security vulnerabilities even though a scanner did not find an issue. As (Deng, 2016) states, "A compiler can be correct and yet be insecure."

D'Silva and his team identified three classes of security weaknesses introduced by compiler optimizations:

i. information leaks through persistent state

ii. elimination of security-relevant code due to undefined behavior

iii. introduction of side channels

They produced the following example where a compiler can introduce security vulnerabilities.

```
crypt(){
key = 0xC0DE; // read key
... // work with the secure key
key = 0x0; // scrub memory
}
```

Here the value of *key* is reset by the developer as a way to clear the value of the variable. However, a compiler can view the last instruction as unnecessary because it is never used again. To improve software performance, the compiler may completely ignore the last statement. This creates a security vulnerability by way of any reference to the memory location addressing the variable *key*.

4. **Scanners do not understand intent**

Because scanners rely on a predefined set of rules, they cannot interpret the intent. (Zhioua, 2014) states that "If we had a security policy that expresses the 'confidentiality of user's payment information' requirement, current static code analysis tools will not be able to concretize it or to relate it the concrete user information variables in the source code."

(Zhioua, 2014) demonstrates this concept with an example:

```
// input data: credit card number + expiry date + crypto
int creditCardNumber = input_creditCardNumber;
Date expiryDate = input_date;
int cryptogram = input_cryptogram;

// Encrypt credit card number
Cipher rsa = Cipher.getInstance("RSA/ECB/PKCS1Padding",
                                SunPKCS11-eTokenPKCS11);

rsa.init (Cipher.ENCRYPT_MODE, sharedKey);

byte[] encrypted_creditCardNumber = rsa.doFinal(Integer
          .toString(creditCardNumber).getBytes());

// Encrypt cryptogram
byte[] encrypted_crypto = rsa.doFinal(Integer
          .toString(cryptogram).getBytes());

// Encrypt expiry date
byte[] encrypted_date = rsa.doFinal(Integer
          .toString(expiry_date).getBytes());

// Store user information
storeUserInfo(encrypted_creditCardNumber, encrypted_date,
     encrypted_crypto);

// logging encrypted data
logMessage = "User Information encrypted: " +
     encrypted_creditCardNumber + " " + encrypted_date + "
" +
     encrypted_crypto + " " + sharedKey.toString();

logger.log(Level.INFO, logMessage);

// Send user information to invoice_edit_service
sendUserData(creditCardNumber, expiry_date, cryptogram);

// Logging User Information
```

```
        logMessage = "User information sent: " + creditCardNumber +
        " " +
              expiry_date + " " + cryptogram;
        logger.log(Level.INFO, logMessage);
…
```

And the method sendUserData() is as follows:

```
        public void sendUserData(int n, Date d, int c) throws
        NetworkException {
                    try {
                          HttpClient httpclient =
        HttpClients.createDefault();
                          HttpPost httppost = new
                                HttpPost("http://www.domain.com/invoice/"
                          );

                          // HTTP Request parameters
                          List<BasicNameValuePair> params = new
                                ArrayList<BasicNameValuePair>();
                          params.add(new BasicNameValuePair("cardNumber",
                                String.valueOf(n)));
                          params.add(new BasicNameValuePair("expDate",
                          d.toString()));
                          params.add(new BasicNameValuePair("crypto",
                                String.valueOf(c)));

                          httppost.setEntity(new
        UrlEncodedFormEntity(params,
                                "UTF-8));

                          // Execute and get the response
                          HttpResponse response =
        httpclient.execute(httppost);
                          ...
```

The final line of code in sendUserData() passes information in plaintext. (Zhioua, 2014) further explains, "This is a security breach that automated source code vulnerability detection tools cannot recognize automatically using string-matching, and independently from the application expected security objectives."

### 5. Some vulnerabilities can not be identified through automation

Achieving 100% automation for static analysis is not possible. According to (White 2016), "Out of the total 160 rules available there are 85 that either state explicitly that they may be automated or appear to be automatable. Also, the CERT website divides the secure

coding rules into 20 categories. Three out of the 20 categories do not contain any rules that can be automated leaving 17 categories with eligible rules to automate."

Here are some example rules from the SEI CERT Oracle Coding Standard for Java where sound automation is not feasible in the general case:

| Rule | Description |
|---|---|
| NUM03-J | Use integer types that can fully represent the possible range of unsigned data |
| NUM08-J | Check floating-point inputs for exceptional values |
| OBJ02-J | Preserve dependencies in subclasses when changing superclasses |
| OBJ05-J | Do not return references to private mutable class members |
| OBJ11-J | Be wary of letting constructors throw exceptions |
| FIO05-J | Do not expose buffers created using the wrap() or duplicate() methods to untrusted code |
| FIO06-J | Do not create multiple buffered wrappers on a single byte or character stream |
| FIO12-J | Provide methods to read and write little-endian data |
| SER02-J | Sign then seal objects before sending them outside a trust boundary |
| SEC00-J | Do not allow privileged blocks to leak sensitive information across a trust boundary |
| SEC04-J | Protect sensitive operations with security manager checks |
| SEC06-J | Do not rely on the default automatic signature verification provided by URLClassLoader and java.util.jar |
| ENV00-J | Do not sign code that performs only unprivileged operations |
| ENV01-J | Place all security-sensitive code in a single JAR and sign and seal it |

| | |
|---|---|
| ENV03-J | Do not grant dangerous combinations of permissions |
| ENV04-J | Do not disable bytecode verification |
| ENV05-J | Do not deploy an application that can be remotely monitored |
| ENV06-J | Production code must not contain debugging entry points |
| JNI00-J | Define wrappers around native methods |
| JNI02-J | Do not assume object references are constant or unique |
| MSC00-J | Use SSLSocket rather than Socket for secure data exchange |

# CONCLUSION

Static code analysis using scanners is an important part of the software development process. We should not, however, rely solely on the results of scanners. There are many situations where scanners will miss known vulnerabilities and target non-existing ones.

Some teams rely on Dynamic Analysis to complement Static Analysis. The challenge with Dynamic Analysis suggested by (Ernst 2003) is that "There is no guarantee that the test suite over which the program was run (that is, the set of inputs for which execution of the program was observed) is characteristic of all possible program executions."

So how do we address the gap left behind by code scanners? (Kaur 2014) states that "The Software Development process itself appears to look at security as an add-on to be checked and deployed towards the end of the software development lifecycle which leads to vulnerabilities in web applications." This is a costly approach. We need to shift the focus earlier in the Software Development Lifecycle (SDLC), integrating software security into the development process. When you automate security by building it into the SDLC (e.g., by using a policy-to-procedure platform), you're capable of managing virtually all known vulnerabilities, rather than only those that scanners detect. This ensures that your applications are more secure and that your overall risk is lower.

To help overcome the code scanner security gap, there are two key activities that can be performed earlier in the SDLC:

- Start by managing software security requirements. This way, you can build security controls into the design of your software. For one, this serves to abstract the code to a higher level, so that you can clarify its intent and help reduce some of noise with static analysis. For example, (Calderón 2007) have created a taxonomy of security requirements as shown below:

| First level categories | Second level categories |
|---|---|
| Non-repudiation requirements | Non-repudiation |
| Integrity requirements | Modification |
| | Deletion |
| | Datum validation |
| | Exception handling |
| | Prerequisite |
| | Separation of duties |
| | Temporal |
| Availability requirements | Response time |
| | Expiration |
| | Resource allocation |
| Confidentiality requirements | Encryption |
| | Authentication |
| | Aggregation |
| | Attribution |
| | Consent and notification |
| | Cardinality |
| | Traces |

- Perform threat modeling, an activity which "...provides complete information on how the software can be attacked, what can be attacked, which areas are attack prone, what kind of threats are applicable etc. This would give developers an opportunity to solve problems, remove irregularities and non-conformance to standards, and remove unwanted complexity early in the development cycle." (Kaur, 2014). Threat modeling should also be used to drive the implementation of security requirements. This way, these security requirements can be managed throughout the SDLC.

Ultimately, there is no single solution to software security; ensuring that applications meet stringent security and compliance standards is a multifaceted process that requires human intervention and tools, as well as advanced automation platforms.

An important consideration when managing software security requirements earlier on in the software development life cycle is automation: it provides a fail-safe way to reduce vulnerabilities in your software, while expediting the security implementation process. Security Compass' policy-to-procedure platform, SD Elements, is an automation platform with an extensive knowledge-base, covering security, compliance, languages, frameworks, and deployment. It's built and maintained by our PhD research team, advisory consultants, and subject matters experts, and it integrates with virtually all ALMs to fit in seamlessly with any organization's workflow. SD Elements optimizes management of security requirements in the early stages of the SDLC, and it fills in many of the gaps left behind by code scanners. It also stands as a light-weight threat modeling tool, which can supplement human-run threat modeling processes.

Another solution for the security gap left by code scanners is investing in a comprehensive DevSecOps program that is tailored to your organization's unique needs. Security Compass offers such a program, where we start by assessing the current state of your AppSec program, building from ground zero or building on what has been started. Our experts then develop a program strategy tailored to your organization's needs. This means that, wherever we find gaps in your application security practices, we leverage what you have–whether that's a toolset or professional team– to make up for it. While SD Elements is a major component of most of our DSO programs, we also provide other essential resources and tools, like a Security Champions program and Just-in-Time Training, to create a security-centric organization.

To learn more about Security Compass' SD Elements platform, get a free demo here.

# BIBLIOGRAPHY


1. Ye, Tao et al. "An Empirical Study on Detecting and Fixing Buffer Overflow Bugs", 2016.
2. Taft, S. "Systematic Scanning for Malicious Source Code", 2008.
3. Svoboda, David et al. "Static Analysis Alert Audits: Lexicon & Rules", 2016.
4. Zhioua, Zeinet et al. "Static Code Analysis for Software Security Verification: Problems and Approaches", 2014.
5. White, Benjamin. "Secure Coding Assistant: Enforcing Secure Coding Practices Using the Eclipse Development Environment", 2016.
6. Chess, Brian et al. "Secure Programming with Static Analysis", 2007.
7. D'Silva, Vijay et al. "The Correctness-Security Gap in Compiler Optimization".
8. Aspinall, David. "Secure Programming Lecture 13: Code Review and Static Analysis", 2016.
9. Barbosa, Edgar. "Control Flow Analysis", 2012.
10. Parasoft, "Write better C# Code Using Data Flow Analysis", 2011.
11. Li, Ming. "Algorithms", 2011.
12. Wikipedia. "Static Code Analysis".
13. Wassermann, Gary et al. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities", 2007.
14. Deng, Chaoqiang et al. "Securing a Compiler Transformation", 2016.
15. Security Compass. "Managing Application Security", 2017.
16. Ernst, Michael. "Static and dynamic analysis: synergy and duality", 2003.
17. Kaur, Navdeep. "Mitigation of SQL Injection Attacks using Threat Modeling", 2014.
18. Calderón, Marta E. et al. "A Taxonomy of Software Security Requirements", 2007.
19. Zhioua, Zeinet et al. "Towards the Verification and Validation of Software Security Properties Using Static Code Analysis", 2014.
20. Masood, Adnan et al. "Static Analysis for Web Service Security – Tools & Techniques for a Secure Development Life Cycle", 2015.
21. Williams, Jeff. "What Do You Mean My Security Tools Don't Work on APIs?!!", 2015.
22. Curran, Paul. "Source Code versus Bytecode Analysis", 2016.
23. McManus, Joe. "SEI CERT Oracle Coding Standard for Java".