IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

ISO 3297:2007 Certified ⁜ Impact Factor 7.39 ⁜ Vol. 11, Issue 7, July 2022

DOI: 10.17148/IJARCCE.2022.11764

A Survey on Coverity Scan Analysis

Abhiroop Saha[1], Prof. Raghavendra Prasad S G[2]

Information Science Engineering, RV College of Engineering, Bangalore, India[1,2]

Abstract: The compiler often misses significant programme flaws. Finding vulnerabilities and minimizing faults in a software programme is possible through static code analysis. This paper presents an overview of static code analysis using a tool called Coverity.The Coverity Analysis package offers checkers that do runtime analysis of the code with dynamic as well as static analysis. Checkers look for problems in two areas in general: Quality problems Identify any code that, if executed, will fail in some way. Code that is vulnerable to attack is identified by security problems. For developers that want flexible, in-depth, and accurate source code analysis, coverity static analysis is the go-to solution since it yields a thorough insight of the build environment and source code.This paper will give an insight of some types of coverity defects along with examples.

Keywords: Static analysis, tools, alerts, warnings, vulnerabilities

I. INTRODUCTION

The amount of flaws in a software may be found and reduced in a variety of methods. JUnit, for example, is a very helpful tool for developing tests in Java. The easiest technique to get rid of flaws, according to research, is by studying the code. This is not always feasible since it is highly challenging to teach individuals and gather them for research and programme problem identification. Additionally, using code inspections on a project's whole code base is almost unfeasible. As people frequently fall into the same traps, the majority of mistakes may be classified into well-known categories. Consequently, a software created to check for bugs in other programmes is referred to as a static analyzer or checker. Static analyzers could discover uncommon events or covert back doors.

A source code analyzer should identify vulnerabilities and describe their position and severity. The Common Weakness Enumeration (CWE) entries relate to the weakness class. There are numerous tools that can also report conditions that could expose the weakness, data or control flow related to it, more details about that class of weakness, including examples of how to fix it, the certainty that the weakness is a vulnerability (not a false alarm), or some rating of the severity or ease of exploit.A tool may, at its discretion, create a report that may be utilized by other tools.

The source code checker must contain a way to suppress notifications of flaws deemed to be false alarms or otherwise to be disregarded in order to be useful in repeated runs.An important consideration in static code analysis is false positives. Static analysis tools theoretically create a model of a programme that they examine for specific attributes. The calculated model or the analysis is approximate since static analysis issues are typically intractable. These approximations can lead tools to miss flaws (false negatives) or report strong code as weak (false positives).

A tool must "have an acceptably low false positive rate" in order to be recognised and accepted.Developers may obtain a precise analysis in their integrated development environment (IDE) while they code with the Code SightTM plugin. As well as providing problem triage and management tools within their IDE, Coverity provides developers with all the information they need to address issues, including descriptions, classifications, severity, CWE statistics, defect location, in-depth remedial instructions, and dataflow traces.By pointing at the source code, users of Coverity's Point and Scan desktop programme may onboard apps (including an IaC build capture function). The Coverity CLI functionality is for development teams who prefer a command-line interface.

In this paper,we analyzed various kinds of defects or vulnerabilities which coverity would detect and possible solutions to fix the bugs.

II. LITERATURE SURVEY

[1] proposed describes an architecture that examines alert occurrences, alert retention times, and developer triage of the Coverity database alerts. Additionally, it lists the warnings that were addressed by modifying the code (i.e., were actionable).This paper intends to support academics and tool developers in improving the usability of static analysis tools through an empirical assessment of developer response to warnings reported by Coverity, a cutting-edge static analysis tool.

[2] suggests readers the chance to conduct real-time Coverity studies on various codebases to identify vulnerabilities across a wide range of categories (such as buffer overruns, cross-site scripting XSS, SQL Injections, etc.), and it offers helpful guidance on how to address and resolve these issues.The findings of this technique accurately cluster similar

complaints by using the syntactic and structural information found in static bug reports, speeding up the maintenance process.

[3] listed the static analysis tools that are accessible for the Technical Debt identification. They compared the popularity, characteristics, and empirical support for the validity of nine instruments. Results can assist practitioners and developers in choosing the best solution out of the competition based on the measured data that best meets their goals. They did not, however, assess the agreement and accuracy of the detection.

[4] examined five bug-finding tools for Java that leverage syntactic bug pattern identification (Bandera33, ESC/Java234, FindBugs35, JLint36, and PMD37). They concentrated on the various warnings (also known as rules) that each tool offered, and their findings show some commonalities between the different sorts of mistakes found. This may be because each tool employs a distinct set of trade-offs to produce false positives and false negatives. They concluded that the cautions offered by the various technologies are not connected to one another..

[5] examined test failures and customer reports for three sizable network service software systems to assess the efficacy of static analysis technologies. They come to the conclusion that code-level flaws may be found using static analysis techniques.

[6] looked at OpenBSD's source base to see if security was getting better. For fundamental vulnerabilities (established before the research period), they examined the rate of vulnerability reports and discovered that it declines gradually. Although the aim of our study is to determine the impact of tools, it also looks at vulnerability reports.

III. METHODOLOGY

A. How Coverity Prevent Works

Coverity Prevent finds errors in the code by combining statistical analysis methods with interprocedural data flow analysis.

• Analysis of the data flow between procedures. Each function is examined by Prevent, which then produces a context-specific report for each one. Each summary details every aspect of the function that has a significant impact on behavior that can be seen from the outside. These summaries are known as "function models" by Coverity. Prevent wherever possible uses these summaries to identify the consequences of function calls during data flow analysis.In order to exclude impractical pathways from consideration, Prevent additionally does false path pruning. This lowers the cost of computing and the likelihood of producing false positives.

• Statistical investigation. In order to find significant trends in the code, Prevent use statistical inference techniques. These findings are then used to search for statistically significant code abnormalities that may deviate from the programme developers' intended behavior.
Because its algorithms are private, the precise facts are typically unknown to those outside the organization.

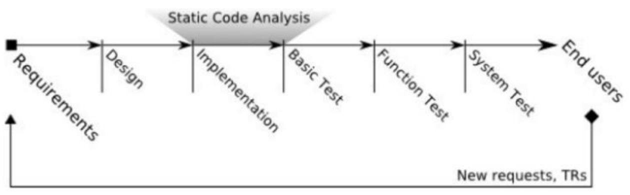


Fig 1:Static code analysis development cycle

B. Process

Initially, it was solely intended to evaluate C and C++ source code.But Prevent currently offers compatibility for the majority of widely used development OSs.Windows, Mac OS, Linux, FreeBSD, NetBSD, Solaris, and HP-UX and compilers for embedded and conventional systems (GCC/G++, Microsoft ARM CC, Sun CC, Green Hills Compiler, Visual Studio, Intel Compiler, etc.).Systems for parallel compilation are fully supported.It's doubtful that project size will cause any issues. Prevent has demonstrated to work well despite having code bases with millions of lines of output code. Additionally, Prevent's user interface has adaptable multi-user functionality.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

ISO 3297:2007 Certified ⁕ Impact Factor 7.39 ⁕ Vol. 11, Issue 7, July 2022

DOI: 10.17148/IJARCCE.2022.11764

The three steps below are typically followed when using Coverity Prevent to evaluate the code:

1. First, emit. Prevent will integrate with your current build process after being configured for your compiler or compilers. It will concurrently process your code using its own built-in C/C++ parser and listen to all compiler calls when you create your project (for example, using the "make" command).In "emit repositories," as Coverity refers to them, it may produce internal binary representations of the project code you provide. When it's feasible, Prevent processes only altered code throughout future build runs.
2. The "inference engine" of Prevent examines the emit repositories utilizing the inter-procedural data flow and statistical analysis methods that were previously discussed. It keeps its derived judgments and function summaries in a database.After that, the "analysis engine" applies a search algorithm to look for flaws.number of checks added by the inference engine to the database.Each checker makes an effort to match for a particular class of probable flaws.
3. Lastly, Prevent compiles the findings from the checkers and translates them to XML format. The final product can then be viewed through or used as inputs to other applications using the graphical web (HTTP) interface that is supplied. Prevent lets you establish creating user accounts to enable shared access to the analysis findings and have their alterations and remarks recorded.

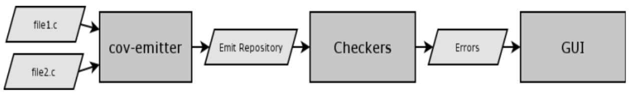


Fig 2: Coverity Prevent process steps

C. Defect Checkers

Various checkers performed by coverity are explained in this section.

• NULL RETURNS: Prior to usage, a function that has the potential to return NULL must be verified. Such dereferences of NULL return values are detected by this checker.

• FORWARD NULL: Normally, if a NULL pointer is dereferenced, a programme would crash. When the pointer has been tested against NULL and is later dereferenced, this is one circumstance in which it may occur. By looking at all potential routes where such NULL dereferences may occur, this check detects such a scenario.

• REVERSE NULL: When a NULL pointer is dereferenced, a programme will often crash. When the pointer is dereferenced before it has been tested against NULL, this can also occur. The check coder should be forewarned to check for NULL before dereference if the dereference is NULL. This check locates such an instance by investigating all potential routes where such NULL dereferences could take place.

• UNUSED VALUE: When a variable is given a pointer value received from a function call but is never utilized elsewhere in the source code, it might lead to unpredictable behavior in addition to inefficient resource utilization. After a value is assigned to a variable, this checker finds all variables that are never used again in the programme.

• REVERSE NEGATIVE: Utilizing a negative value isn't always a good idea. Checking for negative value following a potentially harmful application is one technique to prevent such use. When the value should not be negative before usage, there may be an issue in this case. This checker finds these criteria by investigating every avenue where the use of a negative number could be appropriate.

• RETURN LOCAL: Memory corruption and unpredictable behavior may occur if a function returns a reference to a local stack variable. Such returns are recognised by this checker, who labels them as faults.

• REVERSE NULL: Normally, dereferencing a NULL pointer causes a programme to crash. When the pointer is dereferenced without first being compared to NULL, this can also occur. The check coder should be informed to do a check against NULL prior to the dereference if the dereference is NULL. By looking at all potential routes where such NULL dereferences may occur, this check detects such a scenario.

• UNUSED VALUE: When a variable is given a pointer value received from a function call but is never utilized elsewhere in the source code, it can lead to wasteful resource utilization as well as unpredictable behavior. This checker discovers those variables that, when a value is set to them, are never utilized again in the programme.

• STACK USE: Some programmes, such as device drivers, impose restrictions on the overall stack allocation. To find out if there is a violation of overall stack consumption, use this checker. It is necessary to enable this particular checker. This checker is not automatically used in the analysis.

• RESOURCE LEAK: Resource leaks can negatively impact the programme in a variety of ways. Program crashes may result from memory leaks. File descriptor and socket leaks can result in software crashes as well as other negative outcomes. All of these types of source code leaks are found by this checker, which flags them as errors. Along with routine memory leak checks, it also looks for intriguing circumstances like aliasing.

• CHECKED RETURN: In various situations, it may be required for the developer to verify the value that a function call has returned. To verify whether all error situations are addressed during the function call, this checker use static analysis.

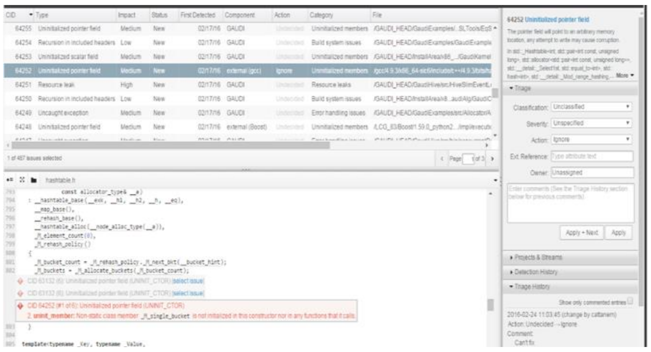
• DEADCODE: A section of source code is considered dead code if it can never be reached while the programme is being run. Normally innocuous, it might pose issues if the dead code includes some crucial code for the software to run as intended. This tool recognises similar static analysis, programme code fragments.

• UNINIT: When variables are not initialized, nondeterministic behavior frequently ensues. In rare circumstances, it can potentially compromise security vulnerabilities. Such variables are recognised by this checker, who highlights their use.

For instance, RESOURCE_LEAK checks for leaks of system resources such as allocated memory and file descriptors by tracking aliases to these resources and searching for control flow paths that result in a resource having no remaining in-scope aliases despite still being allocated. In contrast, CHECKED_RETURN uses statistical analysis to infer when a given function's return value is usually explicitly checked, and then flags anomalous cases where it is not checked.

IV. RESULTS AND ANALYSIS

The figure below shows a typical window of the coverity scan dashboard.



CID	Type	Impact	Status	First Detected	Component	Action	Category	File
64255	Uninitialized pointer field	Medium	New	02/17/16	GAUDI	Undecided	Uninitialized members	/GAUDI_HEAD/GaudiExamples/...SLTools/EqS
64254	Recursion in included headers	Low	New	02/17/16	GAUDI	Undecided	Build system issues	/GAUDI_HEAD/GaudiExamples/GaudiExample
64253	Uninitialized scalar field	Medium	New	02/17/16	GAUDI	Undecided	Uninitialized members	/GAUDI_HEAD/InstallArea/x86_.../GaudiKernel
64252	Uninitialized pointer field	Medium	New	02/17/16	external (gcc)	Ignore	Uninitialized members	/gcc/4.9.3/x86_64-slc6/include/c++/4.9.3/bits/h
64251	Resource leak	High	New	02/17/16	GAUDI	Undecided	Resource leaks	/GAUDI_HEAD/GaudiHive/src/HiveSlimEventL
64250	Recursion in included headers	Low	New	02/17/16	GAUDI	Undecided	Build system issues	/GAUDI_HEAD/InstallArea/x8...audiAlg/GaudiC
64249	Uncaught exception	Medium	New	02/17/16	GAUDI	Undecided	Error handling issues	/GAUDI_HEAD/GaudiExamples/src/Allocator/A
64248	Uninitialized pointer field	Medium	New	02/17/16	external (Boost)	Undecided	Uninitialized members	/LCG_83/Boost/1.59.0_python2.../impl/execut

1 of 457 issues selected   Page 1 of 3

hashtable.h

```
            const allocator_type& __a)
    : __hashtable_base(__exk, __h1, __h2, __h, __eq),
      __map_base(),
      __rehash_base(),
      __hashtable_alloc(__node_alloc_type(__a)),
      _M_element_count(0),
      _M_rehash_policy()
    {
      _M_bucket_count = _M_rehash_policy._M_next_bkt(__bucket_hint);
      _M_buckets = _M_allocate_buckets(_M_bucket_count);
 CID 63132 (6): Uninitialized pointer field (UNINIT_CTOR) [select issue]
 CID 63132 (6): Uninitialized pointer field (UNINIT_CTOR) [select issue]
 CID 64252 (#1 of 6): Uninitialized pointer field (UNINIT_CTOR)
 2. uninit_member: Non-static class member _M_single_bucket is not initialized in this constructor nor in any functions that it calls.
    }

  template<typename _Key, typename _Value,
```

64252 Uninitialized pointer field

The pointer field will point to an arbitrary memory location, any attempt to write may cause corruption.

In std::_Hashtable<int, std::pair<int const, unsigned long>, std::allocator<std::pair<int const, unsigned long>>, std::__detail::_Select1st, std::equal_to<int>, std::hash<int>, std::__detail::_Mod_range_hashing,... More

Triage

Classification: Unclassified
Severity: Unspecified
Action: Ignore
Ext. Reference: Type attribute text
Owner: Unassigned

Enter comments (See the Triage History section below for previous comments)

Apply + Next   Apply

Projects & Streams

Detection History

Triage History

Show only commented entries

2016-02-24 11:03:45 (change by cattanem)
Action: Undecided → Ignore
Comment:
Can't fix

Fig 3: Coverity Dashboard

The following characteristics are used for each alert:

• CID: A distinct alert's unique identification;

• Kind: Depending on whatever rule checker found the alert, Coverity reports the type of each alert;

• Effect: Level of alert severity (High, Medium, Low);

• Date First Detected: The moment an alert was first discovered;

• Status: New for warnings that were still in the code at the time the most recent analysis report was gathered, Fixed for alerts that were removed, and Dismissed for alerts that were designated as false positives or deliberate by the developers;

• Classification: 'Unclassified' is the default setting for the alerts when they are first set up. Some warnings may be manually categorized by developers as a bug, false positive, intentional action, or anything that is still pending.

Some defects are analyzed with examples below.

1. CHECKED_RETURN

CHECKED_RETURN finds many cases where the return value of a function is ignored when it should have been checked. For example, it detects the case where the code neglects to handle an error code returned from a system call. This is a statistical type checker: it determines which functions should be checked based on use patterns. For this checker, you establish the pattern as a ratio between the number of times the return value of a function is checked and the total

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

ISO 3297:2007 Certified ⁂ Impact Factor 7.39 ⁂ Vol. 11, Issue 7, July 2022

DOI: 10.17148/IJARCCE.2022.11764

number of times that function is called.

➢ For C/C++

The CHECKED_RETURN checker finds instances of inconsistencies in how function return values are handled. For example, it detects the case where the code neglects to handle an error code returned from a system call that the code does check in other places.Ignoring returned function error codes and assuming that an operation was successful can cause incorrect program behavior, and in some cases system crashes. The only way to suppress returned function error codes is to cast the called function result to a void.

For the following functions (which read data from a byte-oriented input, store it into a buffer, and return the number of bytes that have been copied to the buffer or a negative value), CHECKED_RETURN also reports issues if the returned value is not used or is not compared to 0. In such cases, the buffer can potentially be accessed outside of the range where the data has been copied by the read operation.

size_t read(int fd, void *buf, size_t count);

size_t fread(void *buf, size_t size, size_t count, FILE *fp);

Example:

```
int f(int fd)
{
   char buf[10];
   if (read(fd, buf, 10) < 0)
      return -1;
   return buf[9];
}
```

The CHECKED_RETURN checker does not check overloaded comparison operators that are used as function arguments (for example, the != operator).

➢ For Java

Java methods typically throw exceptions to indicate errors, but occasionally programmers use return values to indicate special cases. CHECKED_RETURN is a statistical checker that determines whether the return value of a method should be tested after each call.

This CHECKED_RETURN performs the following actions:

- Examines the number of call sites for each method that return a primitive value (see the following for default methods).
- Counts the number of times the return value is checked.
- If the ratio of checked call sites to total call sites is greater than 80 percent (which can be changed with the stat_threshold option), defects are reported at call sites of methods that need to be checked where the value is not checked or used at all.

By default, CHECKED_RETURN checks the following methods:

- java.io.InputStream
  - read()
  - read(byte[])
  - read(byte[], int, int)
  - skip(long)
- java.io.Reader
  - read()
  - read(char[])
  - read(char[], int, int)
  - read(java.nio.CharBuffer)
  - skip(long)

For the following methods (which read data from a byte-oriented input, store it into a buffer, and return the number of bytes that have been copied to the buffer or a negative value), CHECKED_RETURN also reports issues if the returned

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

ISO 3297:2007 Certified ⁜ Impact Factor 7.39 ⁜ Vol. 11, Issue 7, July 2022

DOI: 10.17148/IJARCCE.2022.11764

value is not used or is compared to 0. In such cases, the buffer can potentially be accessed outside of the range where the data has been copied by the read operation.

- int InputStream.read(byte[] buf);
- int InputStream.read(byte[], int offset, int count);
- Any override of the preceding methods.

Example:

```
int f(InputStream is) throws IOException
{
  byte buffer[] = new byte[10];
  // Number of copied bytes is ignored
  if (is.read(buffer, 0 , 10) < 0) {
    return -1;
  }
  // 'buffer' may be accessed out of range.
  return buffer[9];
}
```

2. DEADCODE

DEADCODE finds code that can never be reached due to branches whose condition will always evaluate exactly the same each time. In other words, DEADCODE reports false paths.

Also, DEADCODE does not warn about function-level dead code such as static functions that are never called.

Faulty code assumptions or other logic errors are often responsible for dead code. These defects can have a broad range of effects. For example, if you make a logic error, such as <= instead of a <, or && instead of ||, the resulting behavior might be incorrect. At best, dead code increases the size of source code (and associated binaries). More seriously, logic errors can cause important code to never execute, which can adversely affect program results. At worst, logic errors can cause a program to crash.

Some dead code might be intentional. Defensive error checks, for example, might cause some currently unreachable error paths, but are included to guard against future changes. Also, code that uses #if preprocessor statements to conditionally compile different blocks for different configurations might have dead code in certain configurations.

Fixing these defects depends on what the code was intended to do. Removing truly dead code will eliminate the defect.

Other checkers report events along viable execution paths. DEADCODE does the opposite: It reports paths that are not viable. If you don't keep this in mind, the messages it generates might seem confusing.

When DEADCODE reports a defect, the main event is one of the following:

- dead_error_line: When the dead code consists of a single line
- dead_error_begin: When the dead code includes more than one line of code.

Dead code must lie on the path of an infeasible condition. The dead_error_condition event points to that condition.

The DEADCODE event can also show additional path events that are related to the infeasible condition. These events should explain why the condition cannot be satisfied.

In the following C/C++ example, dead code appears in the second if statement because p cannot be null. Check if handle_error() should be called earlier.

```
int deadcode_example1(int *p) {
   if( p == NULL ) {
      return -1;
   }
   use_p( *p );
   if ( p == NULL ) {      // p cannot be null.
      handle_error();     // Defect: dead code
      return -1;
   }
   return 0;
}
void deadcode_example2(void *p) {
   int c = ( p == NULL );
   if ( p != NULL && c ) {       // Always false
      do_some_other_work();     // Defect: dead code
   }
}
```

3. UNINIT_CTOR

The UNINIT_CTOR checker finds instances of a non-static data member of a class or struct that is declared with the class or struct, not in a parent class, and not initialized in a path in the constructor.

The constructor of a class is generally required to adhere to the contract that it initializes all of the members of the class. This is a very common coding standard. Uninitialized data members are unsafe because calling member functions can access them either directly, if it is public, or through a member function. These defects can cause the usual problems with accessing uninitialized variables, such as corrupting arbitrary data within the address space of the program.

The checker tracks each uninitialized member interprocedurally, starting from the initialization list. The checker follows the member variable down all call chains from within the constructor, checking for initializations. This is repeated for all paths within the constructor. Because the callee does not pass interprocedural context to the caller, the **cov-make-library** command is ineffective in suppressing false positives from the UNINIT_CTOR analysis. As with UNINIT, the best way to suppress an UNINIT_CTOR false positive is to use a code-line annotation to suppress an event.

The following example shows a constructor that does not initialize a data member.

```
class Uninit_Ctor_Example1 {
    Uninit_Ctor_Example1(int a) : m_a(a) {
    // Defect:  m_p not initialized in constructor
    }
    int m_a;
    int *m_p;
};
```

DOI: 10.17148/IJARCCE.2022.11764

Coverity constructs and does an analysis on the code after receiving it from the private repository. Following that, a Coverity server receives those findings.Based on the defect category, the flaws or vulnerabilities in VIA code are examined..The defect occurrences count obtained from local dev branch would need to compared with master branch coverity analysis report.The commit command after analysis needs to be commented so that issues are not committed to coverity server.A separate Jenkins build job is created for running coverity analysis on the local dev branches. The Jenkins build receives the updated code for verification.Then we have to merge changes to this branch and run coverity for this.We analyze the build logs to find the number of issues reported in this run. Then we compare this with latest coverity logs from master branch.The following logs provide information about the total number of coverity issues reported and also a breakup of different kinds of issues.

V. CONCLUSION

When it comes to the effectiveness of its defect identification, Prevent excels in highlighting critical flaws while maintaining a low false positive rate. It will likely improve the bug hunting skills, increase productivity, and provide more time to focus on other facets of the software product.

Additionally, prevent is a really useful tool. It includes several features that are obviously intended for end consumers. Its web-based user interface is quite simple to use, and it gives teams a great way to keep track of defects discovered. Its multi-user features assist managers in assigning problem solutions to specific developers and preventing duplication and pointless overlap. It also complies with the preferences of the majority of developers by minimizing the impact on their code by not needing code annotations or modifications to the build process, while nevertheless producing extremely valuable results despite the limited input it demands.

In this paper we discussed various defects which coverity tool could detect and explained a few of them with examples.We also discussed how these bugs after correction goes into Jenkins build for verification.In fact, the exorbitant expense of utilizing coverity is the only thing keeping us from totally recommending it to everyone. We are convinced that the developers will not only uncover more bugs more quickly, but actually enjoy the bug-hunting process a lot more if the enterprise can justify the expense of acquiring the software.

REFERENCES

[1] N. Imtiaz, B. Murphy and L. Williams, "How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage," 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 2019, pp. 323-333, doi: 10.1109/ISSRE.2019.00040.

[2] B. Baloglu, "How to find and fix software vulnerabilities with coverity static analysis," 2016 IEEE Cybersecurity Development (SecDev), 2016, pp. 153-153, doi: 10.1109/SecDev.2016.041.

[3] A. Ozment and S. E. Schechter, Milk or Wine: Does Software Security Improve with Age? 2006 Usenix Security Symposium, Vancouver, B.C., Canada, 31 July-4 Aug. 2006.

[4] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P.Hudepohl, and M. Vouk, On the Value of Static Analysis for Fault Detection in Software, IEEE Trans. on Software Engineering, v. 32, n. 4, Apr. 2006.

[5] Al Mamun MA, Khanam A, Grahn H, Feldt R (2010) Comparing four static analysis tools for java concurrency bugs. In: Third Swedish Workshop on Multi-Core Computing (MCC-10)

[6] Aniche M, Treude C, Zaidman A, Van Deursen A, Gerosa MA (2016) Satt: Tailoring code metric thresholds for different software architectures. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 41–50

[7] Antoine JY, Villaneau J, Lefeuvre A (2014) Weighted krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation. In: 14th Conference of the European Chapter of the Association for Computational Linguistics", pp 550–559

[8] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman.Analyzing the state of static analysis: A large-scale evaluation in open source software. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1,pages 470–481. IEEE, 2016.