PARASOFT®

WHITEPAPER

Getting Started With Static Analysis

WHAT IS STATIC ANALYSIS?

Static analysis is the process of examining source and binary code without execution, usually for the purposes of finding bugs or evaluating quality. Unlike dynamic analysis (unit testing, system testing, and so on), which requires a running program to work, static analysis can be run on source without the need for an executable.

This means static analysis can be used on partially-complete code, libraries, and third-party source code. Static analysis is accessible to the developer, to be used as code is being written or modified, or to be applied on any arbitrary code base. In the application security domain, static analysis goes by the term Static Application Security Testing (SAST). Many commercial tools support both security vulnerability detection alongside bug detection, quality metrics and coding standard conformance.

Static analysis tools are mandated or highly recommended by safety standards such as ISO 26262, DO-178B/C, IEC 62304, IEC 61508, and EN 50128, for their ability to detect hard-to-find defects and improve security of software. Static analysis tools also help software teams conform to coding standards such as MISRA, AUTOSAR, CWE or CERT.

HOW DOES STATIC ANALYSIS WORK?

Static code analyzers use a compiler-like front-end to build a syntactic and semantic model of the software. The syntactic model is then analyzed against a set of rules or "checkers" to see if the code is in violation. These checkers use pattern-matching algorithms to detect errors such as poor use of language constructs, use of insecure functions, and violations of coding guidelines. The specific set of checkers used is configurable by the user. Pre-set configurations are provided for convenience, for instance for coding standards such as MISRA C.

More sophisticated checkers employ semantic analysis that uses data and control flow to detect complex bugs and security vulnerabilities. To do this, the static analyzer builds an execution model of the software, considers possible paths through the code, and evaluates use of data as it flows from source (like user input) to its destination (such as an API call or system call). Analyzing every single possible condition and path would be too time consuming, so the analyzer uses heuristics to detect the most likely paths for evaluation. Types of errors detected by these checkers include null pointer deference, buffer overflows, and security vulnerabilities like command and SQL injections.

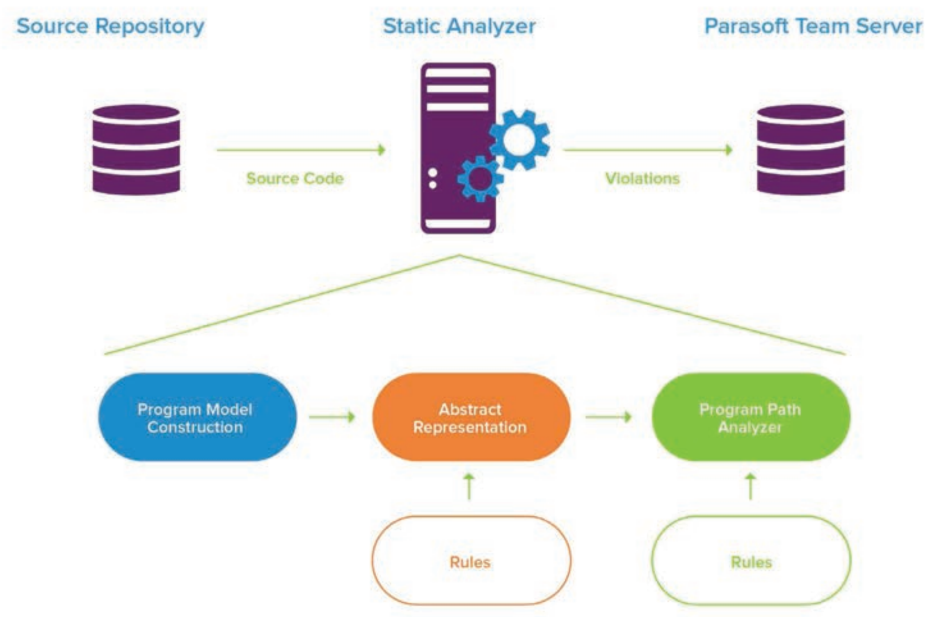


*Figure 1:
A high level view of the static analysis process.*

Static analysis is most often used in two ways:

1. At the developer desktop integrated into their development environment (IDE).
2. At the command line as part of a build or continuous integration process.

Commercial tools like Parasoft C/C++test and Jtest for Java, development, integrate with leading IDEs, and provide solutions for users to effectively manage analysis results. In fact AI & ML can also be applied in the prioritization and management of all identified violations, reducing effort and risk.

So where do you start? This paper will guide you through the introduction of an advanced static analysis tool for your project. The primary assumption is that software already exists and the team is actively working on developing or maintaining a product. Greenfield (a completely new project) is also considered, however, many of the processes and techniques do overlap.

INTRODUCING STATIC ANALYSIS INTO YOUR PROJECT

A great thing about static analysis tools is that they can be introduced and used at any software development phase of a project, effective even if a project is incomplete and partially coded. The biggest challenge with introducing static analysis is that a large amount of code can produce a large number of warnings. Therefore, the focus when integrating static analysis into a project, should be on getting the team productive as soon as possible, and minimizing the opportunity for the team to get overwhelmed by all the static analysis warnings. This is not to diminish the importance of these warnings, but most developers don't have the luxury of fixing existing or legacy code, at least not immediately.

The focus of the approach outlined below is to first integrate the tools into everyday processes so that access and usability is maximized, and then deal with the most critical bugs and security vulnerabilities. Finally, as the team becomes more proficient, optimize the tools and processes to increase the return on investment.

START WITH THE END IN MIND

To get the most out of static analysis tools, it's important to understand the end goal. If the goal is better security, for example, that will shape the focus of analysis and remediation, or if the goal is complying a coding standard such as MISRA C, the focus will become satisfying the coding standard and proving it to certification entities as required.

When first adopting static analysis, it is especially important to keep the end goal in mind. It is easy to fall into the trap that more is better, or in other words, that more analysis and more warnings means getting the most value out of the tool. This is a common trap. Instead, stay focused on the goal. If security is the focus, keep the focus on improving security and reduce the distraction of other types of warnings. Of course critical bugs are always important to track down, but they shouldn't distract from the main goal.

Over time, as the team becomes more proficient, you will be able to incorporate other secondary goals such as improving overall quality and enforcing coding standards. As static analysis becomes part of each developer's daily routine, they will be able to analyze results more quickly and fix bugs more efficiently. At this time, the secondary goals will more effectively be achieved, instead of simply being overwhelming.

STATIC ANALYSIS AT EACH STAGE OF PRODUCT MATURITY

It's incredibly important to keep these goals in mind and ease introduction of static analysis, in order to reduce the likelihood of overwhelming the team. Also important to consider is the maturity of the product under development, as it impacts the way static analysis can be adopted. Consider the major stages of development below.

EXISTING PROJECT IN CURRENT DEVELOPMENT

The most common scenario is a software organization that decides to use static analysis and rolls it out to their current projects. Each project may choose to adopt the tools at the start of a sprint or at the beginning of a major new feature update. Realistically, software teams are always working—even as one product is "finished," another version or variant is underway.

The key aspect of this adoption scenario is that there is a significant body of existing code and new code being developed daily. The recommended approach to integration is called **"a line in the sand"** approach, which we will discuss in more detail in just a moment. At a high level, this approach means improving new code as it's developed, while deferring less critical warnings as technical debt.

EXISTING PROJECT FOR PRODUCT ON THE MARKET IN MAINTENANCE

Adopting static analysis for a mature product may have different goals than a project still under development. This is a product that is in the elder years of the software development lifecycle, in which little new code is being written, only to fix lingering bugs and security vulnerabilities. The primary approach to adopting static analysis for these projects is called **"acknowledge and defer."** Since little new code is being developed, all of the discovered bugs and security vulnerabilities are added to the existing technical debt.

GREENFIELD PROJECT

Although it's not often that software teams get to have a fresh start, a new product and project is the ideal point to integrate new tools and techniques into the development process. In these projects, little existing code specific to the project exists, but it still may rely on third-party and open-source software. Developers can integrate static analysis in their development environments from the start, ensuring a high standard of quality as code is being written. This allows for the adoption of coding standards an ensuring critical static analysis warnings are dealt with as they arise, thus adding less bugs and vulnerabilities to the technical debt pile. The approach to adoption in this case is aptly named **"greenfield."**

Learning what to address immediately and what to defer is the key to success.

MANAGING EARLY STATIC ANALYSIS RESULTS

Once a static analysis tool has been installed and configured in a project and any dependency issues have been sorted out, there is usually a fairly lengthy report of violations and warnings reported by the tool. This can be overwhelming, especially in a large code base, so how these initial results are managed will directly influence the success of integrating the tool into the project.

Not all warnings are critical, so everything doesn't need to be dealt with immediately. Learning what to address immediately and what to defer is the key to success. As mentioned above, the maturity and size of the product has a direct influence on approach, outlined below in more detail.

LINE IN THE SAND APPROACH

As the name implies, in this approach, developers decide that after the initial analysis, they won't let any more critical warnings and violations enter the code base. In other words, they make a commitment to analyze each critical warning to decide its veracity, and implement a timely fix, if it's indeed a bug.

The team may also decide to add critical warnings already discovered in existing code to be added to the list of bugs in their reporting tool. Examples of these types of warnings might be critical security vulnerabilities such as SQL injections or serious memory errors like buffer overflows. In most cases, the less serious warnings can be deferred for later analysis.

You might be thinking, "doesn't this just add to our technical debt?" And you're right, but at this stage, we're ok with that. Any potential bugs within these warnings were already in the technical debt pile. At least now, they are identified and much easier to fix at a later time.

ACKNOWLEDGE & DEFER APPROACH

In the case where a product is already on the market and under maintenance, it is still beneficial to identify any lingering bugs and security vulnerabilities in the code, but it's not feasible for developers to analyze (let alone fix) all these warnings.

In such a case, it makes sense to look at the top most critical reports and decide a course of action. The rest of the warnings are acknowledged, as in the software team recognizes they exist, but they are mostly deferred for a later time. (This again adds to technical debt of the organization, but as mentioned above, these bugs technically exist already as technical debt.)

This approach differs from the line-in-the-sand approach in that after identifying the key warnings, you simply defer the rest, without necessarily any analysis.

GREENFIELD APPROACH

A project with little existing code is an ideal starting point for static analysis. In this case, the software teams can investigate all warnings that arise and fix found bugs. Unlike the other approaches, there are only a few warnings to manage, so developers can tackle the additional workload. This is also an ideal time to implement and enforce a coding standard through the tools, since violations can be identified and fixed right within the IDE and before any code is submitted to version control (which you could also do in the other scenarios described here).

The adoption of static analysis in the three major stages of maturity are differentiated by how they deal with the backlog of warnings, as illustrated below.

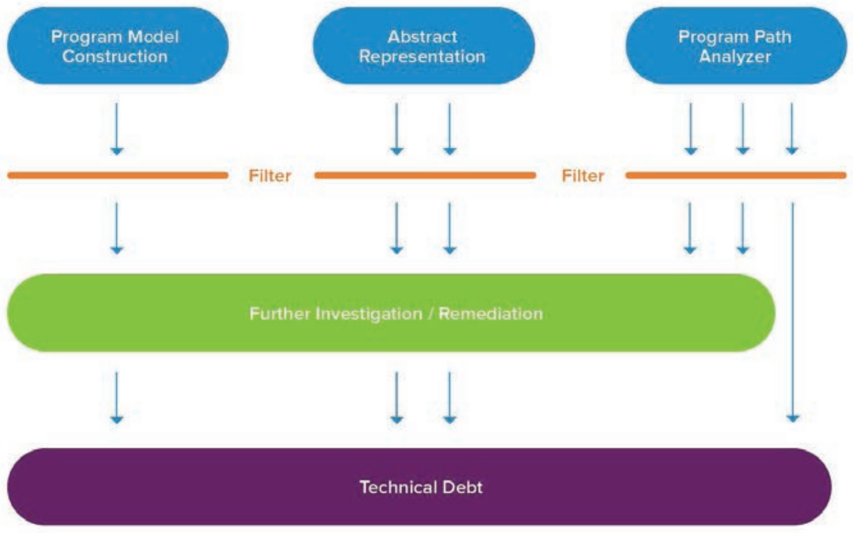


Figure 2:
In a greenfield project, most reported warnings are investigated and fixed with little going into the technical debt pile. Projects under development tend to have a backlog of warnings that are mostly deferred with only critical warnings being dealt with, and products under maintenance tend to have most warnings deferred.

CONFIGURATION VS. FILTERING

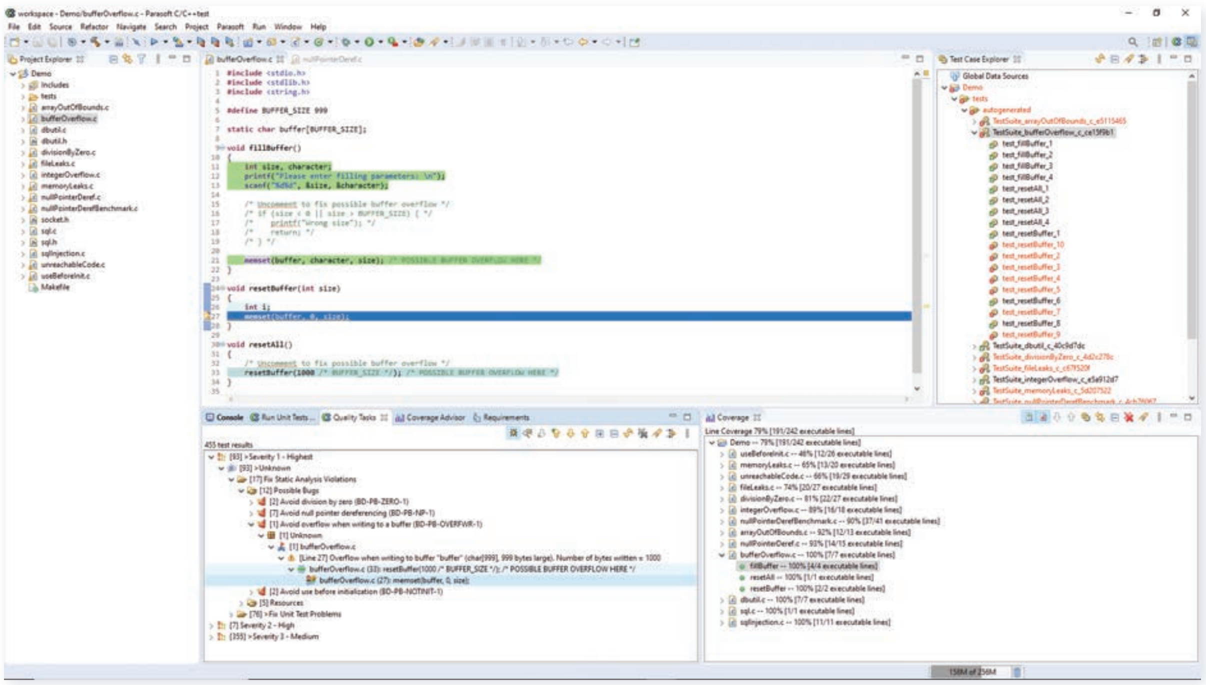
One of the main differences between open source or lightweight static analysis tools and commercial advanced static analysis tools is the ability to configure which set of checkers are enabled for the analysis, and filter out reported results based on warning category, file name, severity and other attributes. This helps with the goal of not getting overwhelmed—developers can focus on just the types of warnings that they are interested in, and reduce the amount of information provided at any given time.

There is also a difference worth noting between configuring checkers and filtering results. Although initially it might seem better to limit the number of rules in the global configuration, filtering should often be used instead, to limit the scope of reporting rather than eliminate the checker entirely. If a rule that later turns out to be important is turned off in the configuration, there will be no history in the warning repository, so you won't be able to find out if the error was introduced by recent changes or already in the code before static analysis was adopted.

A recommended approach is to use configuration to simply limit the set of rules to those that are foreseeable as useful for the software team. Again, start with the end goal in mind: if improving security is the key goal, it makes sense to enable all security-related rules, disable less important rules, and enable one of the built-in secure coding standards such as CERT C. Then, if you're using an advanced static analysis solution like Parasoft C/C++test, you can leverage its built-in management tools to deal with the data produced from the static analysis reports and drive future development focus.

INTEGRATING STATIC ANALYSIS INTO EVERYDAY WORKFLOWS

The key to making static analysis a success in a project is to make sure the tools are easy to use and accessible by developers, so the tool must provide useful, actionable information upfront without overwhelming users with information. This is best achieved within the environment the developer is working in, such as Eclipse or Visual Studio. Static analysis warnings are delivered in the same manner as compiler errors in the IDE, and these warnings are highlighted in the code to make analysis and fixing much easier. See the following example of Parasoft C/C++test integrated in Eclipse.



Demo
- Includes
- tests
- arrayOutOfBounds.c
- bufferOverflow.c
- dbutil.c
- dbutil.h
- divisionByZero.c
- fileLeaks.c
- integerOverflow.c
- memoryLeaks.c
- nullPointerDeref.c
- nullPointerDerefBenchmark.c
- socket.h
- sql.c
- sql.h
- sqlInjection.c
- unreachableCode.c
- useBeforeInit.c
- Makefile

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define BUFFER_SIZE 999
6
7  static char buffer[BUFFER_SIZE];
8
9  void fillBuffer()
10 {
11     int size, character;
12     printf("Please enter filling parameters: \n");
13     scanf("%d%d", &size, &character);
14
15     /* Uncomment to fix possible buffer overflow */
16     /* if (size < 0 || size > BUFFER_SIZE) { */
17     /*     printf("Wrong size"); */
18     /*     return; */
19     /* } */
20
21     memset(buffer, character, size); /* POSSIBLE BUFFER OVERFLOW HERE */
22 }
23
24 void resetBuffer(int size)
25 {
26     int i;
27     memset(buffer, 0, size);
28 }
29
30 void resetAll()
31 {
32     /* Uncomment to fix possible buffer overflow */
33     resetBuffer(1000 /* BUFFER_SIZE */); /* POSSIBLE BUFFER OVERFLOW HERE */
34 }
35
```

Global Data Sources
- Demo
  - tests
    - autogenerated
      - TestSuite_arrayOutOfBounds_c_e5115465
      - TestSuite_bufferOverflow_c_ce15f9b1
        - test_fillBuffer_1
        - test_fillBuffer_2
        - test_fillBuffer_3
        - test_fillBuffer_4
        - test_resetAll_1
        - test_resetAll_2
        - test_resetAll_3
        - test_resetAll_4
        - test_resetBuffer_1
        - test_resetBuffer_10
        - test_resetBuffer_2
        - test_resetBuffer_3
        - test_resetBuffer_4
        - test_resetBuffer_5
        - test_resetBuffer_6
        - test_resetBuffer_7
        - test_resetBuffer_8
        - test_resetBuffer_9
      - TestSuite_dbutil_c_40c9d7dc
      - TestSuite_divisionByZero_c_4d2c278c
      - TestSuite_fileLeaks_c_c67f520f
      - TestSuite_integerOverflow_c_e5a912d7
      - TestSuite_memoryLeaks_c_5d207522
      - TestSuite_nullPointerDerefBenchmark_c_4cb26067

```
455 test results
[93] >Severity 1 - Highest
  [93] >Unknown
    [17] Fix Static Analysis Violations
      [12] Possible Bugs
        [2] Avoid division by zero (BD-PB-ZERO-1)
        [7] Avoid null pointer dereferencing (BD-PB-NP-1)
        [1] Avoid overflow when writing to a buffer (BD-PB-OVERFWR-1)
          [1] Unknown
            [1] bufferOverflow.c
              [Line 27] Overflow when writing to buffer "buffer" (char[999], 999 bytes large). Number of bytes written ≤ 1000
                bufferOverflow.c (33): resetBuffer(1000 /* BUFFER_SIZE */); /* POSSIBLE BUFFER OVERFLOW HERE */
                  bufferOverflow.c (27): memset(buffer, 0, size);
        [2] Avoid use before initialization (BD-PB-NOTINIT-1)
      [5] Resources
    [76] >Fix Unit Test Problems
[7] Severity 2 - High
[355] >Severity 3 - Medium
```

```
Line Coverage 79% [191/242 executable lines]
Demo -- 79% [191/242 executable lines]
  useBeforeInit.c -- 46% [12/26 executable lines]
  memoryLeaks.c -- 65% [13/20 executable lines]
  unreachableCode.c -- 66% [19/29 executable lines]
  fileLeaks.c -- 74% [20/27 executable lines]
  divisionByZero.c -- 81% [22/27 executable lines]
  integerOverflow.c -- 89% [16/18 executable lines]
  nullPointerDerefBenchmark.c -- 90% [37/41 executable lines]
  arrayOutOfBounds.c -- 92% [12/13 executable lines]
  nullPointerDeref.c -- 93% [14/15 executable lines]
  bufferOverflow.c -- 100% [7/7 executable lines]
    fillBuffer -- 100% [4/4 executable lines]
    resetAll -- 100% [1/1 executable lines]
    resetBuffer -- 100% [2/2 executable lines]
  dbutil.c -- 100% [7/7 executable lines]
  sql.c -- 100% [1/1 executable lines]
  sqlInjection.c -- 100% [11/11 executable lines]
```

158M of 256M

Figure 3:
Example of static analysis tool integration into an IDE.

So what does a developer do when analyzing each warning? This is best described in the process diagram shown below.

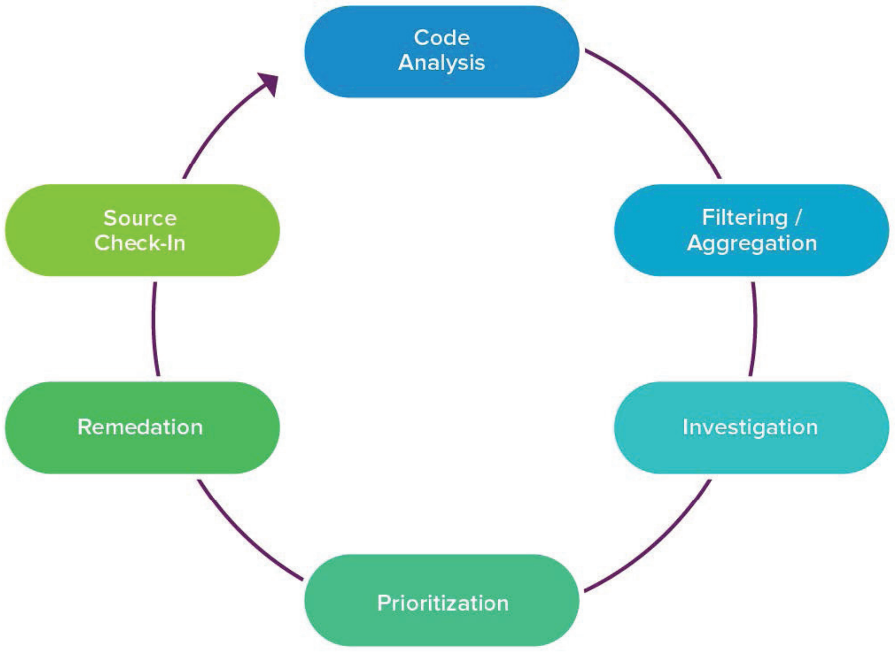


Figure 4:
The static analysis warning lifecycle.

It starts with aggregating and filtering the static analysis results, a critical first step to prioritize and focus on key warnings. Usually, QA and team leaders decide on the quality goal in mind and structure the configuration of the analysis around this goal. To improve security, for example, checkers related to security would be enabled, like a secure coding standard such as CERT C/C++.

Developers then investigate and fix the warnings they find, based on the policies the team has in place and the maturity of the product. As explained above, in a greenfield project, most warnings would be investigated and prioritized since the amount of code would be relatively small, while at later stages of maturity the filtering and prioritization would be stricter so developers would only address truly critical warnings. In either case, the process is the same.

After changes are made in response to a static analysis warning, code is checked into version control and analyzed again during the next build. This short and tight feedback loop greatly improves the quality and security of the code right at the time it's being written and modified.

INTEGRATION INTO BUILD SYSTEMS AND CONTINUOUS INTEGRATION PIPELINES

The main integration point for static analysis tools and build systems is through a command line interface. Static analysis used in this fashion acts somewhat like a compiler would in the build structure. Files are processed in the same manner, although the output isn't an executable but rather results that are stored in a repository, indexed by file and build number.

With Parasoft C/C++test, this is handled by Parasoft's reporting and analytics system (Parasoft DTP), which is both a repository and an intelligence engine that analyzes the results. This analysis and accompanying information is fed back to each developer to their IDE and made available through Parasoft's web portal for managers and team leads.

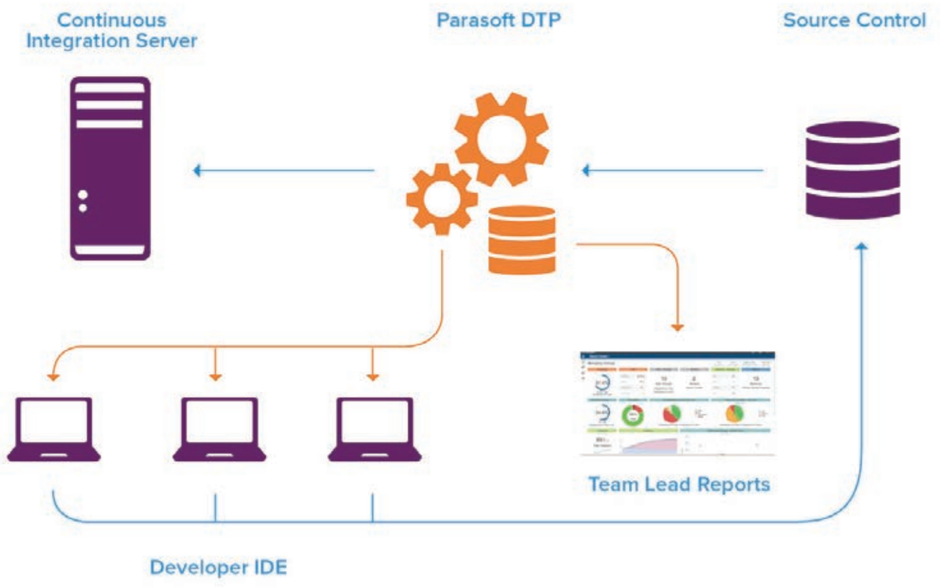
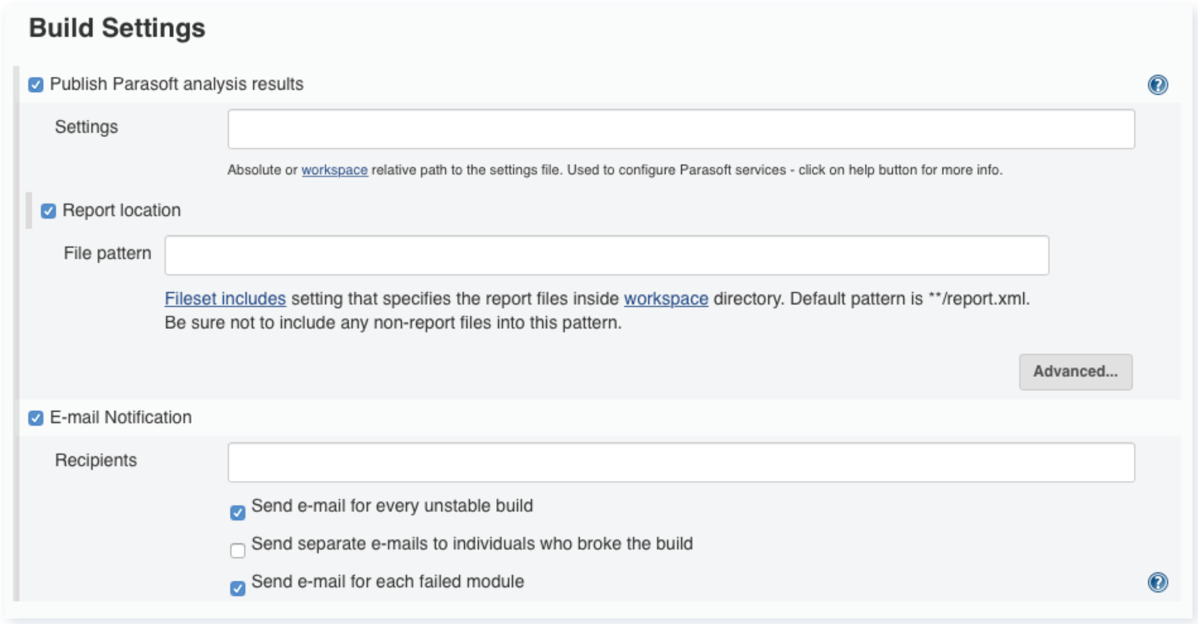


Figure5:
Integrating Parasoft static analysis into a continuous integration pipeline. Parasoft DTP is the central repository and analysis engine for warnings.

Parasofts automation capabilities are designed to be integrated into any kind of build system used at a customer premises, including continuous integration processes. Parasoft tools integrate with popular CI tools such as Jenkins, GitLab, GitHub, Bitbucket and Azure DevOps is making it easy to include automated unit testing and static analysis as part of an integration, build and deploy cycle. For example, the Parasoft plug in for Jenkins provides a build settings menu to enable the analysis and location for reports.

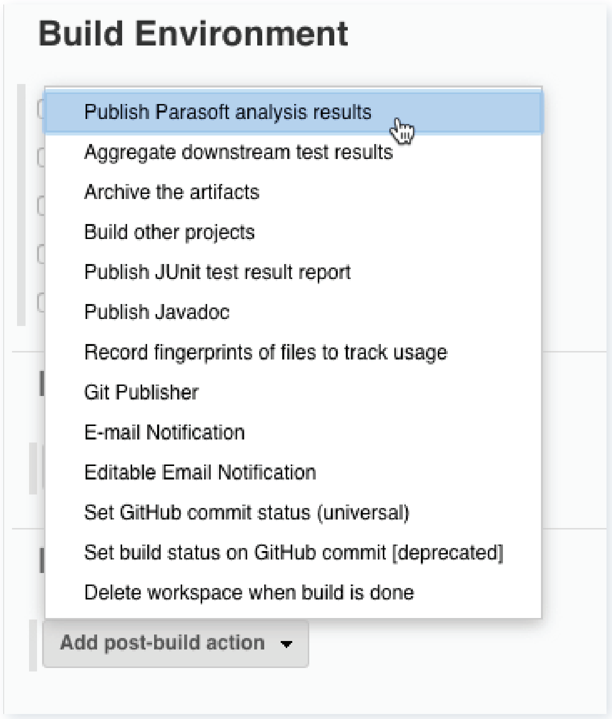


Build Settings

- [✓] Publish Parasoft analysis results
  - Settings:
  - Absolute or workspace relative path to the settings file. Used to configure Parasoft services - click on help button for more info.
  - [✓] Report location
    - File pattern:
    - Fileset includes setting that specifies the report files inside workspace directory. Default pattern is **/report.xml. Be sure not to include any non-report files into this pattern.
    - Advanced...
- [✓] E-mail Notification
  - Recipients:
  - [✓] Send e-mail for every unstable build
  - [ ] Send separate e-mails to individuals who broke the build
  - [✓] Send e-mail for each failed module

Figure 6:
Build settings specifically for Parasoft static analysis within Jenkins.

Also, the plugin includes a custom build environment for Parasoft analysis, integrating the analysis process into an existing pipeline setup. See the following example from Jenkins build menu. Integrating directly with the common tools used in the industry makes including full-project testing and static analysis straightforward.



Build Environment
Publish Parasoft analysis results
Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Publish Javadoc
Record fingerprints of files to track usage
Git Publisher
E-mail Notification
Editable Email Notification
Set GitHub commit status (universal)
Set build status on GitHub commit [deprecated]
Delete workspace when build is done
Add post-build action

Figure 7:
Custom build environment forParasoft analysis.

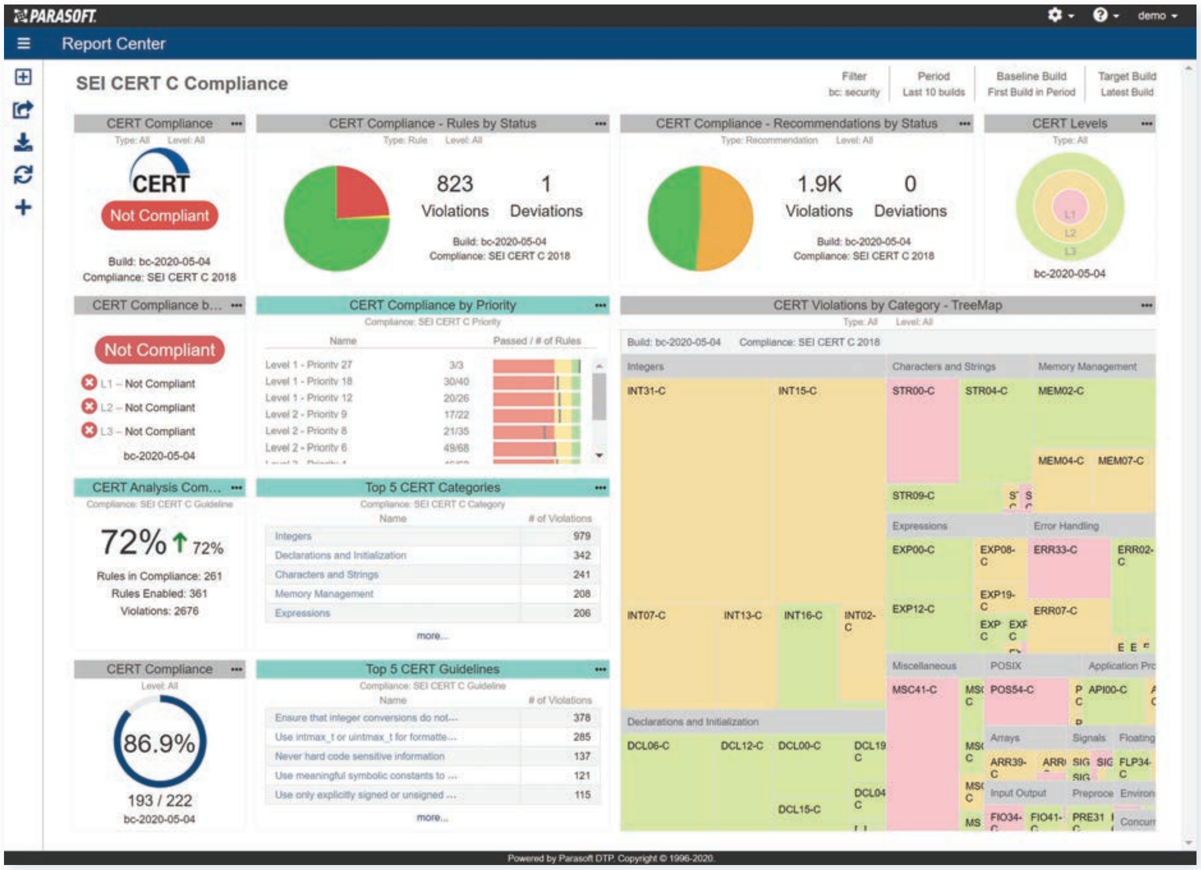
TACKLING THE BACKLOG OF WARNINGS & TECHNICAL DEBT

The next phase of adopting static analysis tools is working to reduce the backlog of warnings and technical debt in a project. In the case of a product under maintenance or in development, there is likely a sizeable backlog to deal with. In the case of a greenfield project, there is less backlog, although the recommendations remain the same for each stage of maturity.

The best starting point for dealing with a backlog of warnings is to prioritize and filter the results based on the desired outcome. Using security as an example, it makes sense to prioritize the backlog in terms of security warnings, ranked by criticality. This is a continuation of the approach used when first introducing static analysis into a project, but now the focus is on the next digestible set of warnings for the team to analyze. This is handled is several ways by Parasoft C/C++test, as we'll get into below.

DASHBOARDS FOR HIGH LEVEL ANALYSIS

Parasoft's centralized reporting and analytics dashboards provide developers and managers with the ability to see the current state of a project from various viewpoints, and further navigate into more detail where needed to establish a set of warnings for further investigation. Consider the following dashboard showing a project's current compliance with the CERT C coding standard.



Build: bc-2020-05-04
Compliance: SEI CERT C 2018

CERT Compliance by Priority
Compliance: SEI CERT C Priority

Name	Passed / # of Rules
Level 1 - Priority 27	3/3
Level 1 - Priority 18	30/40
Level 1 - Priority 12	20/26
Level 2 - Priority 9	17/22
Level 2 - Priority 8	21/35
Level 2 - Priority 6	49/68

CERT Analysis Com...
Compliance: SEI CERT C Guideline
Rules in Compliance: 261
Rules Enabled: 361
Violations: 2676

Top 5 CERT Categories
Compliance: SEI CERT C Category

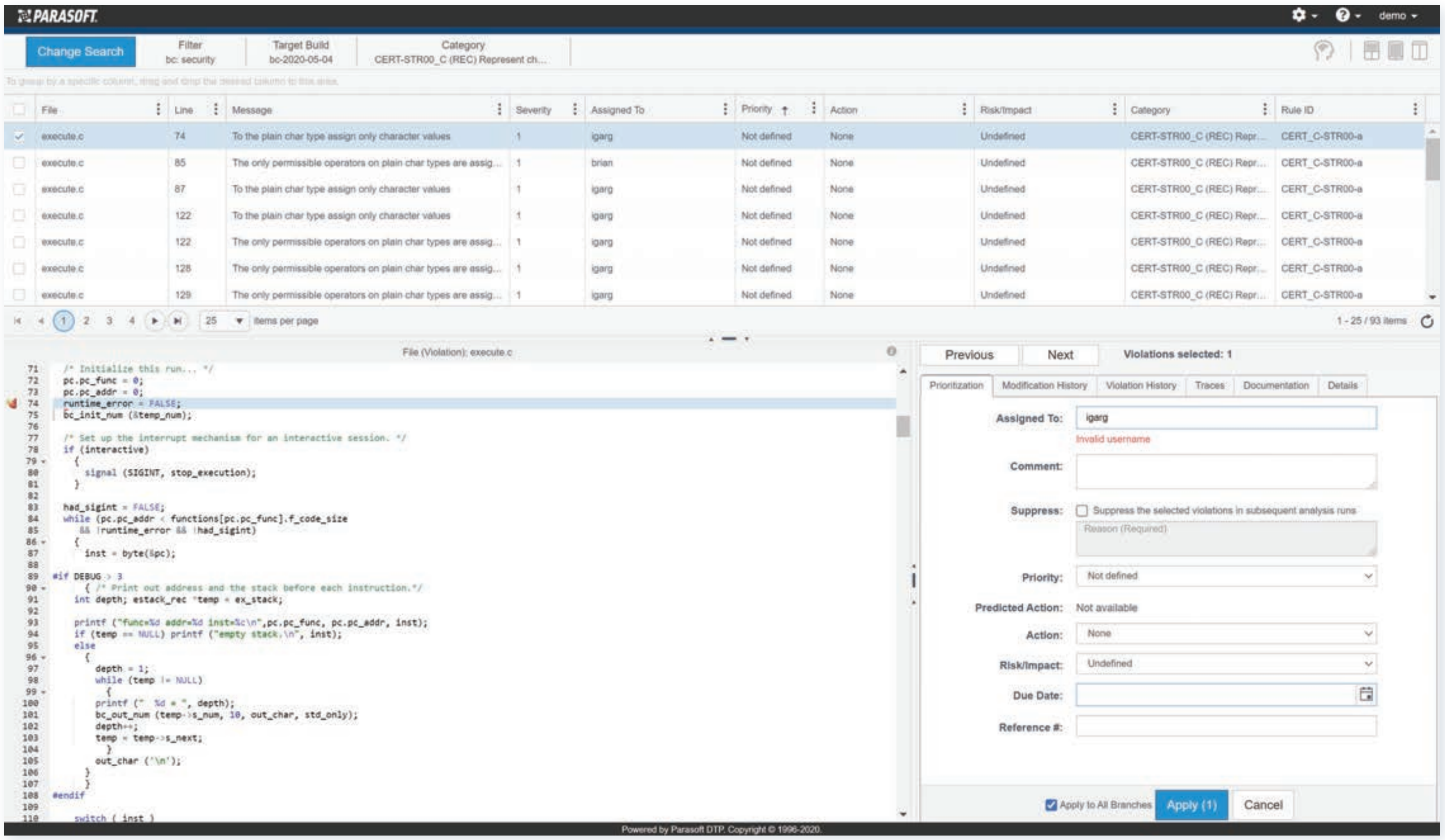
Name	# of Violations
Integers	979
Declarations and Initialization	342
Characters and Strings	241
Memory Management	208
Expressions	206

Top 5 CERT Guidelines
Compliance: SEI CERT C Guideline

Name	# of Violations
Ensure that integer conversions do not...	378
Use intmax_t or uintmax_t for formatte...	285
Never hard code sensitive information	137
Use meaningful symbolic constants to ...	121
Use only explicitly signed or unsigned ...	115

Figure 8:
An example of a web portal dashboard. In this case, a summary of CERT C compliance.

Using this web portal, users can dig deeper into the analysis, down to the file and code level if needed, and investigate warnings entirely within the web portal or in an IDE. Warnings can be prioritized, assigned to developers, suppressed, or marked as a false positive at this stage. See the following example from the Parasoft explorer.



Filter: bc: security | Target Build: bc-2020-05-04 | Category: CERT-STR00_C (REC) Represent ch...

To group by a specific column, drag and drop the desired column to this area.

	File	Line	Message	Severity	Assigned To	Priority	Action	Risk/Impact	Category	Rule ID
✓	execute.c	74	To the plain char type assign only character values	1	igarg	Not defined	None	Undefined	CERT-STR00_C (REC) Repr...	CERT_C-STR00-a
	execute.c	85	The only permissible operators on plain char types are assig...	1	brian	Not defined	None	Undefined	CERT-STR00_C (REC) Repr...	CERT_C-STR00-a
	execute.c	87	To the plain char type assign only character values	1	igarg	Not defined	None	Undefined	CERT-STR00_C (REC) Repr...	CERT_C-STR00-a
	execute.c	122	To the plain char type assign only character values	1	igarg	Not defined	None	Undefined	CERT-STR00_C (REC) Repr...	CERT_C-STR00-a
	execute.c	122	The only permissible operators on plain char types are assig...	1	igarg	Not defined	None	Undefined	CERT-STR00_C (REC) Repr...	CERT_C-STR00-a
	execute.c	128	The only permissible operators on plain char types are assig...	1	igarg	Not defined	None	Undefined	CERT-STR00_C (REC) Repr...	CERT_C-STR00-a
	execute.c	129	The only permissible operators on plain char types are assig...	1	igarg	Not defined	None	Undefined	CERT-STR00_C (REC) Repr...	CERT_C-STR00-a

1 2 3 4 — 25 items per page — 1 - 25 / 93 items

File (Violation): execute.c

```
 71   /* Initialize this run... */
 72   pc.pc_func = 0;
 73   pc.pc_addr = 0;
 74   runtime_error = FALSE;
 75   bc_init_num (&temp_num);
 76
 77   /* Set up the interrupt mechanism for an interactive session. */
 78   if (interactive)
 79     {
 80       signal (SIGINT, stop_execution);
 81     }
 82
 83   had_sigint = FALSE;
 84   while (pc.pc_addr < functions[pc.pc_func].f_code_size
 85      && !runtime_error && !had_sigint)
 86     {
 87       inst = byte(&pc);
 88
 89 #if DEBUG > 3
 90       { /* Print out address and the stack before each instruction.*/
 91     int depth; estack_rec *temp = ex_stack;
 92
 93     printf ("func=%d addr=%d inst=%c\n",pc.pc_func, pc.pc_addr, inst);
 94     if (temp == NULL) printf ("empty stack.\n", inst);
 95     else
 96       {
 97         depth = 1;
 98         while (temp != NULL)
 99           {
100         printf ("  %d = ", depth);
101         bc_out_num (temp->s_num, 10, out_char, std_only);
102         depth++;
103         temp = temp->s_next;
104           }
105         out_char ('\n');
106       }
107       }
108 #endif
109
110     switch ( inst )
```

Violations selected: 1

Assigned To:	igarg Invalid username
Comment:	
Suppress:	☐ Suppress the selected violations in subsequent analysis runs Reason (Required)
Priority:	Not defined
Predicted Action:	Not available
Action:	None
Risk/Impact:	Undefined
Due Date:	
Reference #:	

☑ Apply to All Branches

Powered by Parasoft DTP. Copyright © 1996-2020.

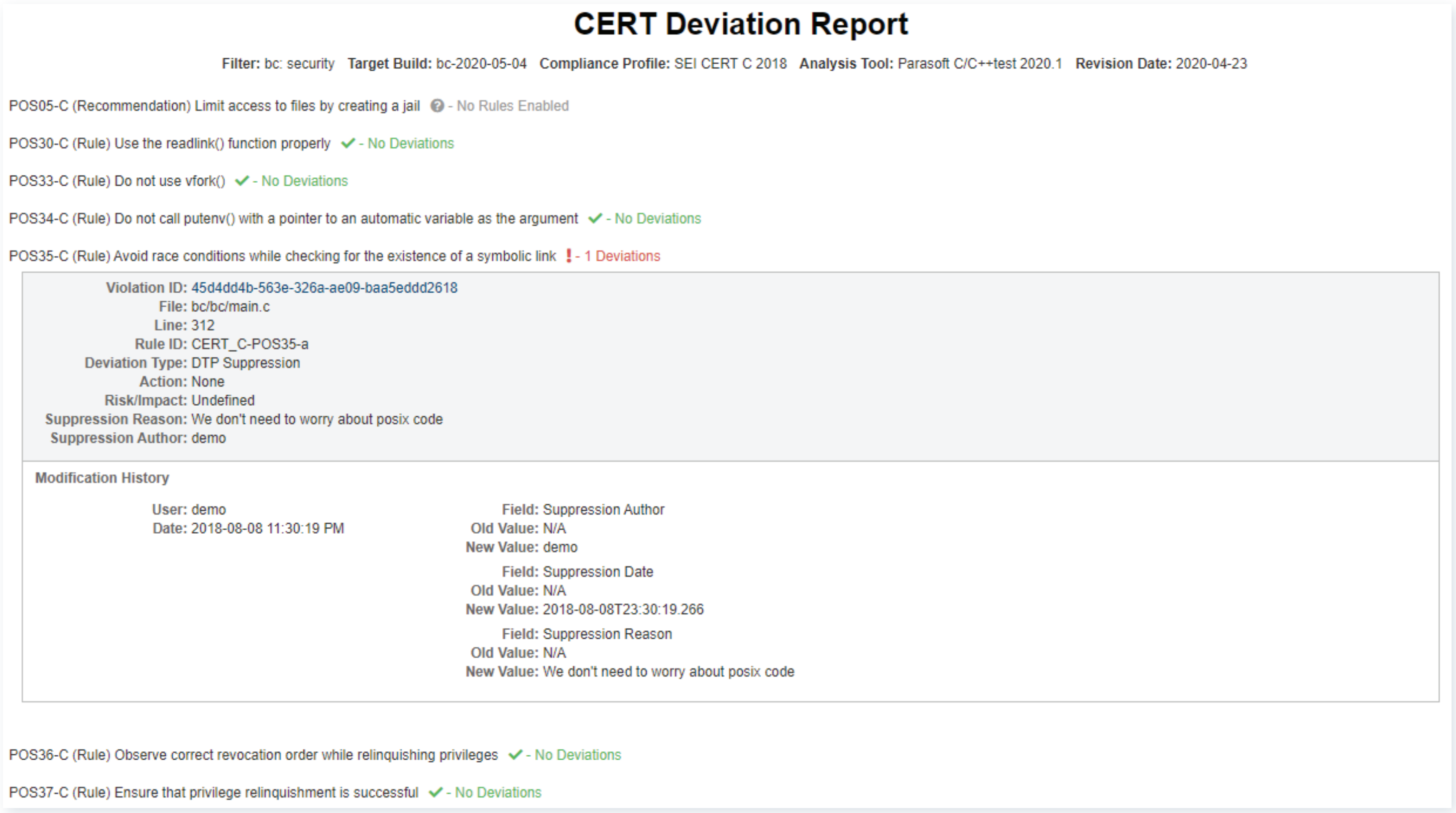
Figure 9:
The Parasoft violation explorer.

MANAGING VIOLATIONS OF CODING STANDARDS

In most cases when analyzing source for coding standard compliance, violations are reported as static analysis warnings. In a large project, there are initially going to be lots of warnings, and managing them quickly and efficiently is critical. Parasoft's violation explorer is the key tool to navigate, evaluate, prioritize, and assign reported errors for remediation. If a static analysis rule violation turns out to be valid but justifiable, considered harmless, or not applicable, a developer can suppress the error and a deviation can be documented. These deviations are reported up through each level of the project to the dashboard and compliance documentation.

To make coding standard compliance feasible for existing projects, it's critical that teams focus on the rules that are considered mandatory first. Compliance is often based on meeting the mandatory requirements with violations of recommended rules if they are documented appropriately. Standards allow rules to be re-categorized, if non-mandatory, allowing for violations if justifiable and documented. Without this, trying to correct every violation becomes infeasible.

Parasoft saves its users many extra hours of work by providing management with a navigable interface to explore violations and automatically generate reports for certification evidence, if needed. An example of a MISRA C deviation report is shown below.


CERT Deviation Report

Filter: bc: security **Target Build:** bc-2020-05-04 **Compliance Profile:** SEI CERT C 2018 **Analysis Tool:** Parasoft C/C++test 2020.1 **Revision Date:** 2020-04-23

POS05-C (Recommendation) Limit access to files by creating a jail - No Rules Enabled

POS30-C (Rule) Use the readlink() function properly ✓ - No Deviations

POS33-C (Rule) Do not use vfork() ✓ - No Deviations

POS34-C (Rule) Do not call putenv() with a pointer to an automatic variable as the argument ✓ - No Deviations

POS35-C (Rule) Avoid race conditions while checking for the existence of a symbolic link ! - 1 Deviations

Violation ID: 45d4dd4b-563e-326a-ae09-baa5eddd2618
File: bc/bc/main.c
Line: 312
Rule ID: CERT_C-POS35-a
Deviation Type: DTP Suppression
Action: None
Risk/Impact: Undefined
Suppression Reason: We don't need to worry about posix code
Suppression Author: demo

Modification History

User: demo
Date: 2018-08-08 11:30:19 PM

Field: Suppression Author
Old Value: N/A
New Value: demo

Field: Suppression Date
Old Value: N/A
New Value: 2018-08-08T23:30:19.266

Field: Suppression Reason
Old Value: N/A
New Value: We don't need to worry about posix code

POS36-C (Rule) Observe correct revocation order while relinquishing privileges ✓ - No Deviations

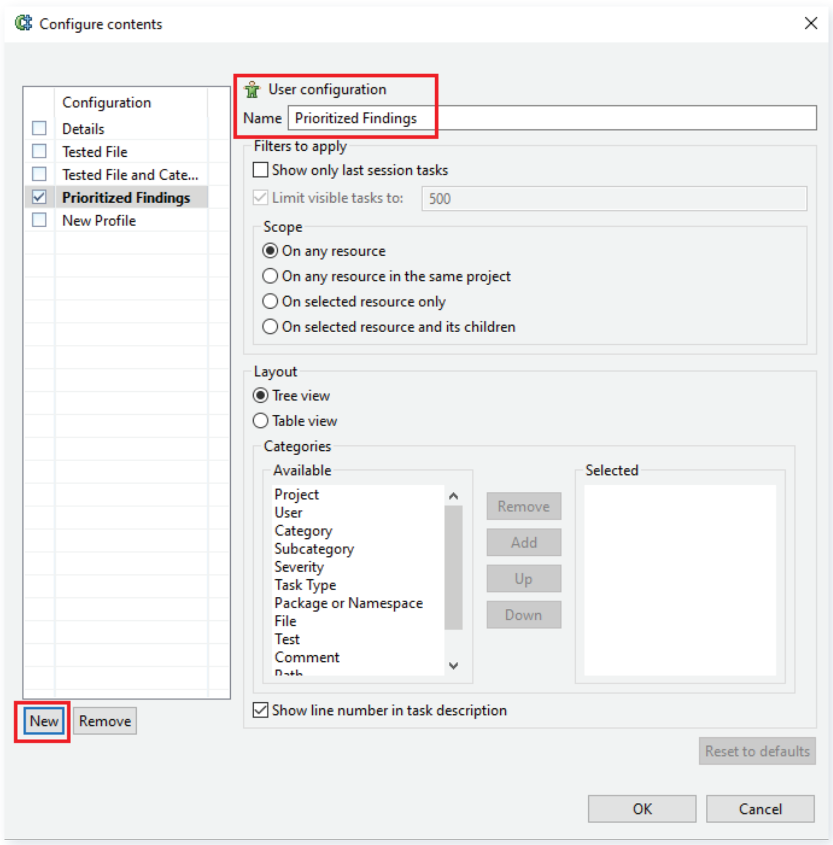
POS37-C (Rule) Ensure that privilege relinquishment is successful ✓ - No Deviations

*Figure 10:
An example Parasoft MISRA C Deviation Report.*

MANAGING BACKLOG OF BUG & SECURITY WARNINGS

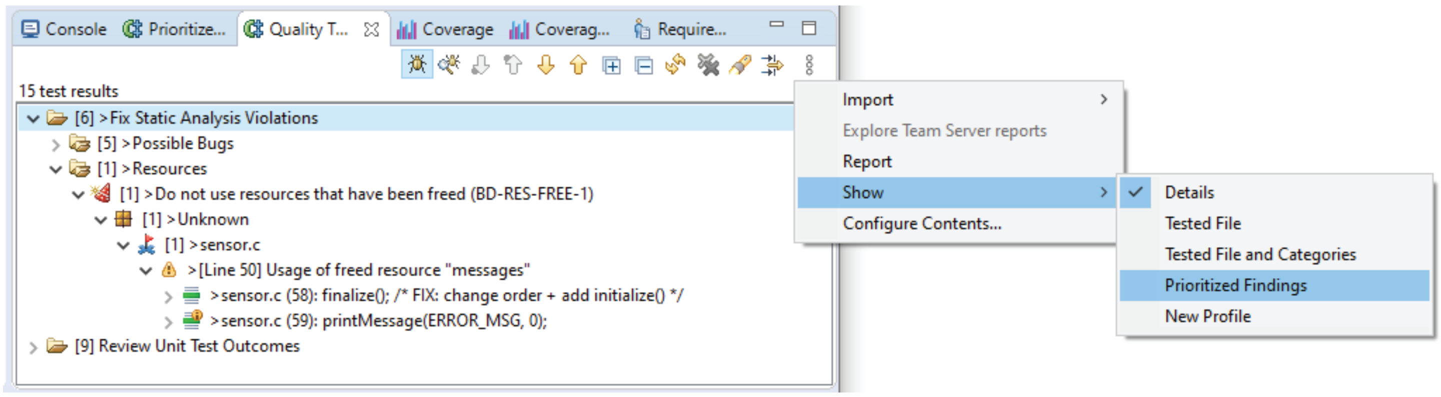
It's important for teams adopting static analysis to understand that it isn't necessary to fix or analyze all warnings. All warnings are not created equal, so the severity level is the best indicator of how much effort should be placed on investigating and fixing a warning. Continuing the "line in the sand" approach discussed above, when digging into the backlog of warnings, we are effectively moving the line in the sand a bit farther each time.

Parasoft Jtest and Parasoft C/C++test enable users to prioritize and filter warnings within the IDE using configurations. For example, severity and category (type of warning, like security-related) can be used to create a set of warnings suitable for analysis. An example new user configuration is shown in the following image.



Name: Prioritized Findings
[ ] Show only last session tasks
[✓] Limit visible tasks to: 500
(●) On any resource
( ) On any resource in the same project
( ) On selected resource only
( ) On selected resource and its children
(●) Tree view
( ) Table view
[✓] Show line number in task description

Figure 11:
Custom test configuration settings inside the IDE.



```
15 test results
[6] >Fix Static Analysis Violations
  [5] >Possible Bugs
  [1] >Resources
    [1] >Do not use resources that have been freed (BD-RES-FREE-1)
      [1] >Unknown
        [1] >sensor.c
          >[Line 50] Usage of freed resource "messages"
            >sensor.c (58): finalize(); /* FIX: change order + add initialize() */
            >sensor.c (59): printMessage(ERROR_MSG, 0);
[9] Review Unit Test Outcomes
```

Figure 12:
Configurations can be selected in the DTP Findings view in the IDE.

Incrementally moving the “line in the sand” to tackle the next highest priority and category is the best approach for dealing with a large backlog of warnings. Eventually, a cut off point is reached due to time and budget, but software teams should feel comfortable that they have made significant improvement in quality and security despite any remaining backlog of warnings.

OPTIMIZING STATIC ANALYSIS

At some point, when a software team has integrated static analysis into their day-to-day activities and everyone is comfortably working with the results, the team will want to customize the tools to better fit their project and organization.

This is an important stage of static analysis adoption: optimization. Optimizing static analysis means adapting the rules and the way violations are reported to improve the efficiency and ROI for the organization. Parasoft's static analysis tools support a variety of configuration and customization options to give complete control, if needed, over the static analysis rules and reports.

Define the set of rules to be used. Developers can decide which set of rules are to be enabled, globally, for an entire static analysis run. As mentioned above, it's important to only disable rules that the team doesn't see as useful in the foreseeable future.

Modify settings for data flow analysis. In Parasoft's static analysis tools, flow analysis rules are handled as a special category since they involve more complex analysis. Developers can enable and disable which rules that want to use in this category. There is also the ability to report on un-validated violations (cases where bugs are detected but path analysis has not validated the possibility of this error being possible). This means more potential bugs found (lower false negatives) at the cost of more incorrect findings (false positives).

There are several options for data flow analysis to control the depth of the analysis which impacts performance and accuracy. There are also options for detecting multithreading issues, verbosity of the reports, and custom terminations (functions that terminate a flow of execution). There are also options to control the list of resource functions such as memory creation and deletion (to detect memory leaks, for example). This is also where data sources and sinks are defined for tainted data flow analysis.

Customize rule groups, identifiers, titles and severities. In organizations that have already established coding guidelines, it's possible to customize the warnings reported by Parasoft's static analysis tools to conform to a completely custom set of warnings, descriptions, and categories. These custom configurations can be shared amongst the entire team through the team server as well as directly into each IDE.

Customize existing rules. An important feature of Parasoft's static analysis is the ability to customize both the set of rules to check and the rules themselves—including creating new custom rules. The rules provided with Parasoft tools are a comprehensive set to select from, but specific coding guidelines may differ slightly, or require well-defined exceptions to the rules. Since Parasoft tools use both pattern-based analysis and the more complex flow analysis, customizing each type of rule is different.

Pattern-based rules have checks for pattern matching built into each rule. These are usually modified by parameterizing the rule to suit a user-specific situation. In Parasoft tools, this is easily done with the RuleWizard widget, which provides an easy-to-use interface for customizing existing and new rules (explained in more detail below).

Flow-based rules are more complex and aren't fully customizable, but are parametrized instead, allowing for new applications of the rule logic. An example of this would be modifying the memory leak detection rule to detect leaks from project-specific memory allocation and deallocation functions, or even resource leaks from other resources such as files.

Create new static analysis rules. Software teams can create whole new pattern-based rules, if needed, or base their custom rule on an existing rule. This customization is achieved in the user-friendly RuleWizard, as mentioned above. Rules are created graphically by creating a flow-chart-like representation of the rule or automatically by providing code that demonstrates a sample rule violation.

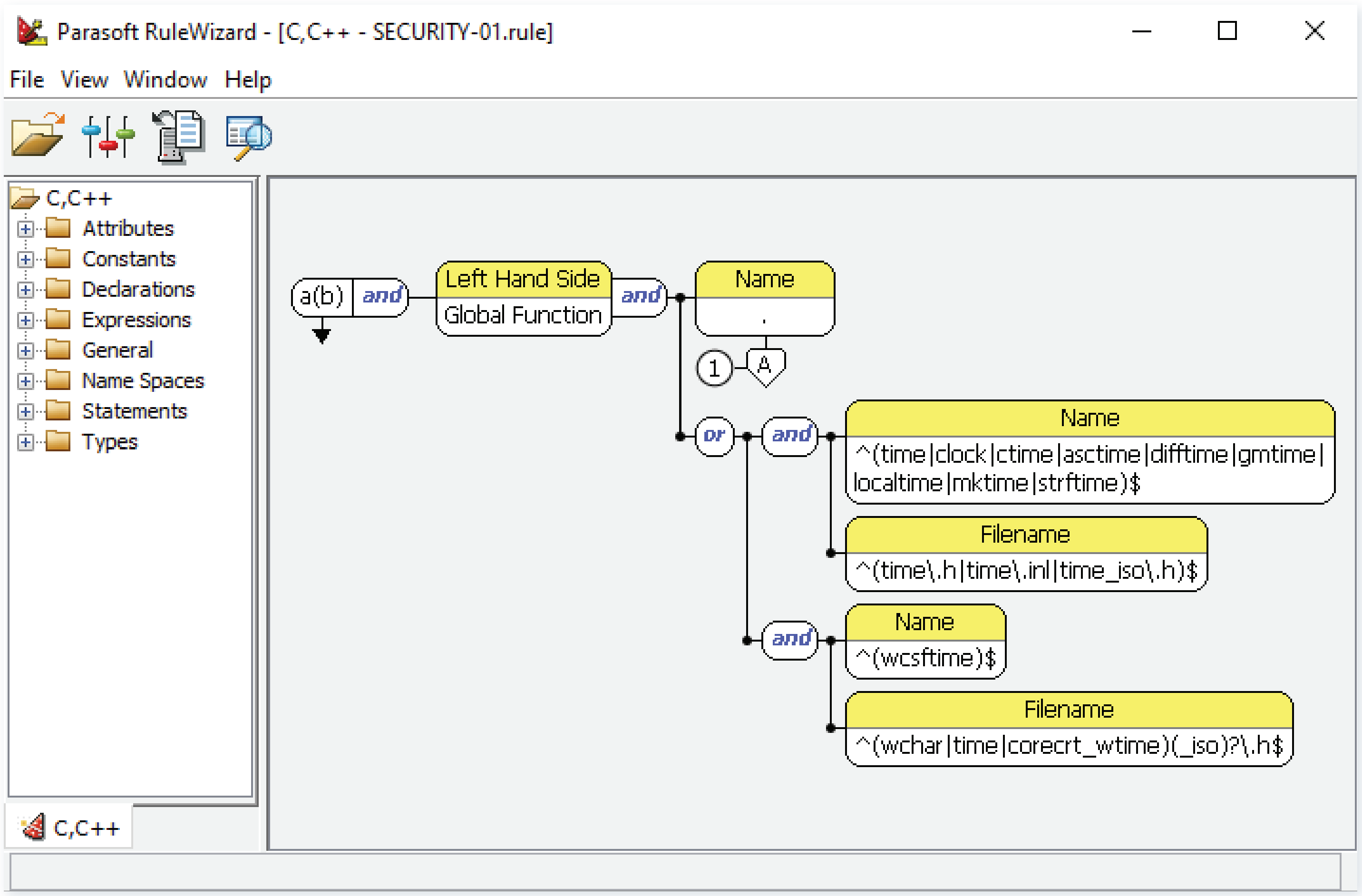


Figure 13:
The RuleWizard provides a visual way to customize static analysis rules.

There are many other customization opportunities with Parasoft static analysis tools, to meet the specific demands of every unique software organization. As adoption of a tool matures, teams can make the most of their investment with customized settings and rules.

SUMMARY

Static analysis tools provide software organizations with the ability to detect and track bugs and security vulnerabilities without needing to execute code. These tools can be applied to existing, legacy, and third-party code, and is also useful for incomplete code, to enable bug detection as soon as code is written, right in the developer's IDE. Parasoft's static analysis tools support industry-leading IDEs and integrate into existing build systems and continuous integration pipelines, while static analysis warnings are stored on a file and build basis alongside test results for prioritization and management.

The adoption of static analysis varies to some extent based on the maturity of the project. A large body of code does result in numerous warnings, but this is completely manageable and the success of adoption depends on how teams decide to tackle the results. Various techniques are introduced for each major maturity level of a project and how these tools can be integrated into the day-to-day workflow for developers, team leads, and managers.

As adoption of Parasoft static analysis tools matures, software teams can optimize their tool chain by customizing the tools themselves to meet their specific needs, increasing their ROI and achieving their project's end goals.

TAKE THE NEXT STEP

Request a demo to see how your embedded software development team can reach complete coverage for security standards, functional safety standards, and more.

ABOUT PARASOFT