nccgroup
freedom from doubt

An NCC Group Publication

Best Practices for the use of Static Code Analysis within a Real-World Secure Development Lifecycle

Prepared by:
Jeremy Boone

Contents

1 Executive Summary

Static application security testing (SAST) is the analysis of computer software that is performed without the need to actually execute the program. The term is usually applied to analysis performed by an automated tool, whereas human analysis is typically called security-focused code review. The primary objective of SAST is to gain an understanding of the software's behaviour, usually with the aim of uncovering security, privacy, and quality defects.

In recent years, commercial SAST solutions have matured considerably, and they now offer numerous methods of integrating with various development processes and support systems: continuous integration, bug trackers, revision control, peer code review tools, and so on. However, NCC Group routinely encounters ineffective or suboptimal static analysis deployments that either fail to accommodate the requirements of a secure development lifecycle (SDLC) or tend to impose a significant burden on development staff, leading to disengagement and patterns of misuse. These shortcomings frequently result in the SAST solution failing to serve its primary purpose: to improve software security.

In this paper we describe a methodology for evaluating and selecting the most appropriate static code analysis solution for your software organisation, as well as best practice guidance for effectively integrating that solution with your development procedures as part of a mature secure development lifecycle. Unfortunately, we must intentionally avoid recommending a specific SAST solution, because in our experience there is no "one size fits all" solution; besides which, vendors typically place DeWitt clauses[1] in their EULAs, to prevent the publishing of benchmark data.

2 Purpose and Motivation

The primary motivator that drives the use of static code analysis within a software development organisation is the ability to deliver higher-quality and more secure products at a cheaper price. Studies indicate that automated static code analysis can detect up to 60% of post-release failures[2]. This metric becomes significant when you consider that a single failure can lead to widespread and costly product recalls. For example:

- In 2002, the National Institute of Standards and Technology (NIST) showed that the annual cost of software defects to the American economy is approximately $59.5 billion[3].
- In 2015, Charlie Miller and Chris Valasek discovered security defects that led to the recall of 1.4 million Fiat Chrysler vehicles[4].
- In 2011, the FDA stated that 24% of medical device recalls can be attributed to software defects[5].

So it becomes easy to imagine how a modest investment in SAST can help to avoid expensive recalls or patching of in-field products. Fearmongering aside, it should be recognised that static code analysis is neither a silver bullet nor a panacea. Forrester's TechRadar Q2 2015 Application Security report [6] summarised it well:

> *SAST provides a significant value to its customers — it has a proven track record and provides extensive benefits. To gain maximum advantage from SAST […] teams must be fairly mature and capable of systematic follow-through on remediation. SAST will continue to see significant success […] for another five to 10 years as application environments become ever more complex and their attack surfaces and risk profiles change.*

[1] http://sqlmag.com/sql-server/devils-dewitt-clause
[2] Q. Systems, "Overview large Java project code quality analysis," QA Systems, Tech. Rep., 2002.
[3] http://www.nist.gov/director/planning/upload/report02-3.pdf
[4] http://www.reuters.com/article/2015/07/24/us-fiat-chrysler-recall-idUSKCN0PY1U920150724
[5] https://threatpost.com/fda-software-failures-responsible-24-all-medical-device-recalls-062012/76720/
[6] https://www.forrester.com/TechRadar+Application+Security+Q2+2015/fulltext/-/E-RES117395

In other words, while commercial static code analysis solutions are beginning to mature, your overall success when with SAST technologies will require diligence and careful preparation. An early-and-often approach is needed, placing emphasis on the correct and calibrated application of the tools from the very start. Great care must be taken when deploying SAST within a software organisation.

3 Why SAST Often Fails

The ultimate success of any SAST deployment is highly dependent on adoption and engagement rates. In other words, if the developers don't trust the static analysis solutions, for example because of high false positive rates, then they won't use them. The best way to build trust is by showing that the solution provides value. The best method of demonstrating value is by showing that the tool is accurate, fast, and does not impose a burden on the developer.

First, you should continually aim to improve the SAST service by seeking out and removing friction between developers and the tools. Where possible, the SAST service should be integrated seamlessly into existing development support systems such as bug trackers, peer collaborative code review tools, source repositories, and the build environment. This helps reduce the overhead every time a user needs to interact with the static analysis service.

Secondly, great effort should be made to reduce the triage burden that is placed on developers. This can be accomplished through concerted efforts to suppress spurious or irrelevant warnings such as false positive reports, historical legacy defects, and defects in third-party libraries. This is not a one-time action, but instead should be continually measured and improved over time.

Finally, the effectiveness of SAST will begin to erode when the use of SAST is not mandatory, or when SAST metrics are not considered essential product launch criteria. To succeed, you need to achieve top-down agreement for the necessary process changes that make static analysis a required part of the release criteria for all software products, along with KPIs and expected minimum quality bars. Otherwise, it becomes too easy for development teams to ignore the defects reported by SAST by adding them to the infinitely-long "for future fix" defect backlog.

4 Methodology

In the opinion of NCC Group, there are five core principles of a successful static code analysis deployment. We believe that you will see the greatest success if you follow these principles when evaluating a SAST solution for purchase, and when deploying it across your organisation. These objectives are in congruence with the goals of a secure development lifecycle.

Principle	Objectives
Automation	Through the use of automation, you can ensure that all software written within or for your organisation will be scanned by SAST. The SAST tool should be ubiquitous and integrate transparently with existing development support systems such as bug trackers, revision control systems, continuous integration, and peer code review systems. Defects that are discovered by SAST should be immediately and automatically assigned to the developer that introduced the defect. Automated SAST will improve velocity by reducing the cost of expensive manual procedures such as peer code reviews, software testing, security vulnerability assessments, and privacy auditing.

Principle	Objectives
Analysis precision	The SAST tools will provide value through the accurate identification of quality, security, and privacy defects. False positives will be suppressed to reduce noise and eliminate “needle in the haystack” problems when attempting to triage the defect backlog. Custom rules will be developed that detect emergent classes of security defects, or in response to security incidents that affect your software products.
Metrics-driven decision making	The SAST defect trends should be measured, and quality gates should be established to ensure no software is shipped with unresolved critical defects. A team’s SAST compliance will be based on concrete and measurable data such as defect density, severity, and mean time to fix.
Continual improvement	The accuracy of the SAST tool will regularly be measured, and actions taken to improve the precision of the results. Triage burden will be eased by suppressing historical legacy defects. This backlog will be dealt with in a structured fashion (hint: critical defects first). The SAST solution will commit to a high service level agreement for the sake of its users.
User education	Users will be trained on the use of SAST, and how to remedy common classes of defects.

5 Integration with the Secure Development Lifecycle

Given the best-practice methodology described above, it becomes clear that a successful SAST deployment must be integrated throughout the entire software development process. Furthermore, SAST activities must take place during each phase of the secure development lifecycle. As part of a typical SDLC, static analysis tools are most commonly used by developers during the implementation phase, but they can also be used effectively in other SDLC phases, such as by quality assurance teams during verification, or by the incident response team during the sustainment phase.

Following Microsoft’s [7] model and structure for the secure development lifecycle, we propose where and how SAST should be used as an essential element of each phase of the SDLC.

[7] https://www.microsoft.com/en-us/sdl/default.aspx

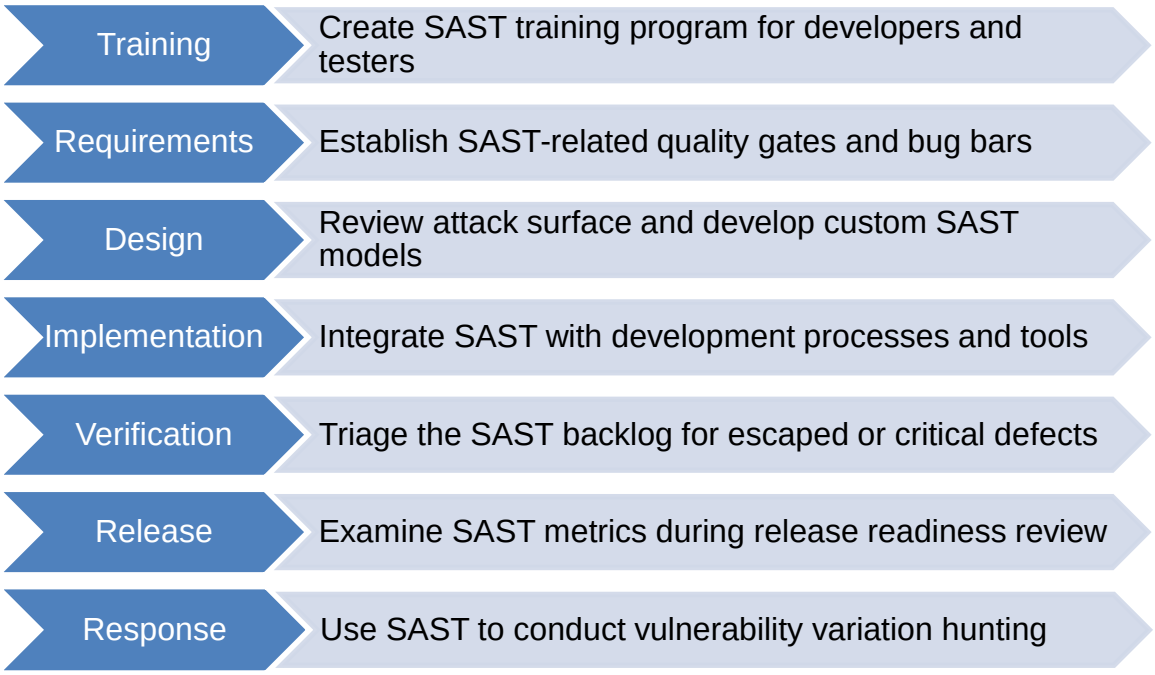

Figure 1. Typical SAST activities during each phase of the SDLC

5.1 Training

Often overlooked, training is an essential aspect of integrating SAST into the software development process. Engineers should be familiar with the SAST solution's capabilities as well as its limitations. It is not sufficient for a developer to massage their code until the reported defect disappears. The developer must understand the root cause of the class of vulnerability detected and be able to develop an appropriate patch. Improper fixes are seen again and again, such as the recent failed attempt to patch the Android Stagefright vulnerability that was quickly demonstrated to be woefully insufficient[8].

When deploying SAST across your organisation, it is important to educate both developers and testers. Training topics should cover:

- What is static analysis?
- What are common classes of defects, and what are the optimal remediation tactics?
- What are the best practices for addressing false positives?
- How was SAST integrated with development support systems at your organisation?
- What process changes have been made to accommodate SAST?

Most SAST vendors offer some form of training, either on site, on the web, or at a remote location. Furthermore, most static-analysis tools come with extensive online documentation, including tutorials. You may also wish to develop a training programme yourself or seek independent guidance from a third party.

5.2 Requirements

During the requirements phase of the classic SDLC, you will typically establish quality gates and bug bars. These agreed internal standards help to define strict security, privacy, and quality criteria that must be met before a software product can ship. Telemetry from your SAST service should be included in these criteria. For example, you may wish to decide that no products will ship until all critical defects identified by SAST have been resolved. For teams that follow Agile methodologies,

[8] http://googleprojectzero.blogspot.co.uk/2015/09/stagefrightened.html

SAST metrics should be part of the acceptance criteria or definition of done[9] for user stories and sprints.

5.3 Design

As part of a SDLC's design phase, best practice states that you should establish design requirements, analyse the software's attack surface, and conduct threat modelling exercises. The outputs of these exercises can help to identify risk areas within your product that may require additional attention from your SAST service. For example, you may discover the use of new technologies, software stacks, or third-party libraries. These components must be studied and adequately modelled within the static analysis engine to ensure the most accurate and precise results are produced.

5.4 Implementation

The implementation phase is where the proverbial rubber hits the road. Here, SAST should be integrated throughout the development process: desktop integrated development environment, continuous integration systems, nightly or continuous builds, collaborative peer code review systems, bug tracker, revision control system, and so on.

Through this broad level of integration, your organisation can discover and resolve security, privacy, and quality defects early in the product development cycle, when the defects are cheapest to fix. In fact, you should strive to identify software defects the moment that they are introduced by a developer, or even before code is checked into the source repository. By tightening the window between when a bug is introduced and when it is resolved, you can reduce the downstream costs associated with typical quality assurance, code review, vulnerability assessment, and privacy audit job functions.

The ultimate goal should be to make SAST ubiquitous by transparently integrating with these systems without introducing friction between the developer and the tools. Every single commit should be analysed by SAST, and a "no new warnings" policy should be introduced.

5.5 Verification

It is not only the development team that has use cases for the static analysis tools. The quality assurance team will have responsibilities as well. The verification phase is QA's time to shine.

During verification, the QA team should triage the SAST defect backlog to identify critical issues that were perhaps overlooked or deferred during implementation. No critical defects should be allowed to escape before the software product is released.

The QA team should also analyse the overall defect trends presented by the SAST service. This includes tracking down hot spots within the code that have higher than average defect densities, or isolating rulesets that trigger abnormally high false positive rates. The QA team should then take corrective action by customising the analysis engine to suppress false positives or by writing custom rules to detect new classes of defects.

5.6 Release

During the release phase of a typical SDLC, you will conduct the final security review that examines all security activities that were performed and decides whether the software is ready to ship. As with any other quality metric, SAST should become part of this release readiness decision making process. You will want to ensure that your software product meets the static analysis bug bars and quality gates that you previously established during the requirements phase.

[9] http://guide.agilealliance.org/guide/definition-of-done.html

5.7 Response and Sustainment

As part of the sustainment phase of an SDLC, you must be prepared to react to security emergencies by executing your product incident response plan. A healthy incident response process will make use of SAST tooling and automation.

First, you should consider auditing your SAST reports to understand whether the externally reported defect was previously known to the engineering team. If so, that's great, but now you have another problem: why didn't they fix the bug before shipping? When an escaped defect is found externally, it can be quite embarrassing. These insights can be extremely valuable from a process improvement perspective.

On the other hand, if the vulnerability was not found by the SAST tool, then you must take corrective action. Part of the incident response due diligence process should include hunting for variations of that defect. Here, static analysis is a great and valuable asset. A custom ruleset can be developed to model the vulnerability's pattern and root cause, and that ruleset should be used to scan the entire codebase. This "find it once, fix it forever" attitude is an important mind set to maintain throughout a secure development lifecycle.

6 Evaluating and Deploying a SAST Solution

A SAST deployment is most effective when it is transparently integrated with your development process and with your various development support systems. This helps to improve adoption rates by reducing the friction between the developers and the tools. A high level of user engagement is a key factor for success with any SAST deployment, and will result in more security, privacy, and quality defects being found and fixed.

The section on static code analysis in NASA's Software Engineering Handbook[10] does an effective job of stating the problem with deploying SAST:

> *Introducing the routine use of static analyzers in an existing development process can be a tricky proposition. Even if the software team is convinced of the benefits such tools can bring, projects should be careful and make the introduction of static analysis as unobtrusive as possible.*

This section describes the criteria for evaluating and selecting a commercial off the shelf (COTS) or open source (OSS) static code analysis solution that is the best fit for your software development process and the support systems you use. You will want to consider whether the SAST tool supports multiple compilers and the analysis of multiple programming languages, whether and how the solution integrates with your development support systems, and the overall accuracy and configurability of the SAST analysis engine.

Before any purchase is made, the first step must be to evaluate multiple SAST solutions. Commercial SAST vendors will happily provide a temporary license for this purpose. You should base your decision to purchase a commercial static analysis tool on your own evaluation, not on a vendor demonstration. Once you have trial licenses, you can begin evaluating each tool against the criteria outlined below.

6.1 Programming Language Support

Perhaps the most important question when selecting a SAST solution for your organisation is whether it supports all of the programming languages and platforms that your development team uses. For example, if your organisation develops an Android app, a static analyser that has knowledge of Android's API is better than one that merely has knowledge of C/C++ and Java.

You should also question how long the tool has supported the programming languages. This can be an indicator of the level of maturity and whether the vendor has comprehensively modelled the quirks of the language syntax and all ways to use and abuse the language's standard library and APIs.

[10] https://swehb.nasa.gov/display/TOOL/SWE-135

6.2 Selecting Natural and Artificial Code Bases

The initial setup of a SAST solution for a code base can be a cumbersome process. The tools can be finicky, requiring careful pre-configuration of a variety of environmental and compiler options. It may take a few attempts at running the tool before it finishes an entire analysis job without hitting an unrecoverable parser error. Furthermore, it is common for a static analysis tool to introduce a runtime overhead of 2x to 10x on your build process.

Understandably, it is extremely frustrating when the SAST tool fails at 3:00 AM after compiling and analysing your code overnight. For this reason, during the SAST evaluation process, you should select a handful of candidate codebases that are sufficiently small and normally compile in thirty minutes or less. This way, you can tweak the tool's settings and incrementally work towards an ideal configuration without needing to wait hours (or days) between runs.

The candidate codebases should also be fairly representative of your overall codebase. This will give you the best visibility into the capabilities of the SAST solution, as different software products will have unique and interesting defect profiles.

In addition to analysing natural code, you should also consider analysing an artificial code base. A great candidate would be the Juliet test suite[11], which was created by the Software Assurance Metrics and Tool Evaluation (SAMATE) project, and was sponsored, in part, by NIST. Researchers at the Toyota InfoTechnology Center (ITC) published a similar a similar set of artificial tests for benchmarking SAST tools[12]. Combined, these two test suites contain hundreds of tiny C/C++ and Java programs that were carefully crafted to demonstrate a wide variety of defects.

By scanning a mix of natural and artificial code, you will get an accurate indication of the overall detection capabilities of a SAST solution.

6.3 Development Support Systems

Often, an indicator of the maturity of a SAST solution is how well it integrates with the software development process. The most polished solutions will provide varied integration options and a high level of flexibility in how the tool can be deployed internally. Some solutions will integrate with the desktop IDE (integrated development environment) and output their warnings alongside the compiler errors, while others are best deployed into the build environment or continuous integration system, or supplied as a managed service.

Of significant importance when selecting a SAST solution for your organisation is whether the tool supports the wide variety of development support systems on which you rely to create and ship your software products. This includes your compilers, build system, IDE, continuous integration system, peer collaborative code review system, revision control system, and bug tracker. Broad support for these systems and tools will allow you to integrate SAST transparently across the entire development process in support of a mature secure development lifecycle.

When planning a SAST deployment, you must also be realistic about the capability and capacity of your DevOps team. Many SAST and DevOps tools have rich APIs, but does your team have the skill or time to glue them together by creating all of the required custom integration scripts? Throughout the SAST deployment planning process you must carefully decide whether you will build the integrations yourself or seek independent advice and support.

6.3.1 Compiler and Build System

Some static analysis tools will conveniently monitor the build process and watch for invocations of the compiler and linker. In many cases this can be set up by simply prefixing your build command with the name of the SAST wrapper application, such as "`SastWrapper make <target>`". This

[11] http://samate.nist.gov/SARD/testsuite.php
[12] https://github.com/regehr/itc-benchmarks

handily eliminates much of the initial configuration required to get the service off the ground, so you can start seeing analysis results much sooner.

Of course, this is only possible if the SAST tool recognises your build environment and compiler. Quite often this type of setup is fragile to simple renaming of the build system (e.g., `make` to `nccmake`) or compiler executable (e.g., `gcc` to `arm-linux-androideabi-gcc-4.8`) It is vital that you ensure your compilers and build systems are explicitly supported, and when they aren't supported, you must determine whether the SAST tool can be configured to recognise additional compilers.

By monitoring the build process, the SAST tool is able to see precisely which source files are compiled into your target. Many large projects will often have unused source files littered throughout the tree, and it is convenient to exclude those files from your analysis results. No engineer wants to spend time triaging bugs in dead code. In addition to seeing which source files are compiled, the SAST tool can see *how* those files are compiled (i.e., the compiler and linker flags). This is of great importance, because the SAST tool will produce better results if it can make more accurate assumptions about the target architecture (for example x86 vs. ARM, 32-bit vs. 64-bit).

However, it should also be stated that not all SAST solutions are capable of wrapping your build process. Some tools require you to hack your build scripts, which can quickly become a maintenance nightmare. Other SAST tools are completely agnostic when it comes to compiler and build environment, and they simply analyse all source files that reside in a provided directory. Proceed with caution here, because these static analysis tools are more likely to emit spurious false positive defect reports, as they have no visibility into preprocessor directives and include paths that are passed at compile time.

Finally, it is sometimes useful for a SAST tool to support mixed-language builds. This feature is valuable if, in your build environment, a single invocation of "`make`" will compile both C and Java code. Many tools do not support this, and so you are forced to separate your Java source files from the C/C++ files. It can be quite difficult to modify legacy build scripts to accomplish this and, once again, doing so could introduce maintenance problems.

6.3.2 Integrated Development Environment

A mature SAST solution will often ship with premade plugins for a variety of IDEs, such as Eclipse, Visual Studio, IntelliJ, and so on. These plugins will analyse a software engineer's local changes and emit warnings right alongside compiler errors. This is extremely handy, because it encourages the software engineer to use SAST as part of their daily routine. It is always cheaper to find and fix defects mere moments after they are introduced by the developer, while the code is still fresh in mind.

This feature is sometimes referred to by commercial vendors as "incremental" analysis or "fast" analysis. It relies on an interesting technique in which a central SAST server will host a set of intermediate analysis by-products, often referred to as a baseline. When you perform an analysis locally, the IDE plugin will determine which of those intermediates need to be recomputed as a consequence of your local changes. Those source files are then re-analysed, and the rest of the baseline is taken as-is. This can greatly speed up analysis when local changes are confined to just a few source files, as is typical.

Software developers tend to have strong opinions on their IDE of choice (which is perhaps the understatement of the century). You will never succeed in convincing a developer to switch to a different IDE because it does not support the SAST tool you are deploying. The SAST tool you select should support the most commonly used IDEs by your development organisation, and if not, the tool should provide an API enabling you to create your own IDE plugin.

6.3.3 Continuous Integration

A core tenet of Scrum/Agile is enabling your development team to react quickly to change. Continuous integration (CI) systems are often part of an effective Agile development model, because they provide developers with instant feedback when a commit breaks a build or causes a test failure. So if your organisation uses a CI system, then it is important to imagine ways to integrate SAST with your development workflow.

The current best practice is to use your CI system as a trigger that initiates a run of the static analysis tool. This way, when a developer commits a change that violates a SAST rule, thereby introducing a new defect, they can be notified immediately or soon after commit.

A mature SAST solution will provide a powerful API or command line interface that makes it possible to create these types of automation scripts. You should make sure that the development team is given appropriate time to develop these scripts and become accustomed to using SAST as part of the regular development process. A nominal investment (a slightly reduced velocity for one or two iterations) will pay off once the SAST tool is running smoothly.

6.3.4 Collaborative Peer Code Review Systems

SAST should also be integrated with the collaborative peer code review systems that are in use across your organisation. Common examples of these systems include Gerrit, Code Collaborator, and Review Board. You want to take advantage of the engineer's pride in their craftsmanship, and make it easy for them to find and fix any defects found in their code in view of their peers.

The typical way to integrate SAST with a code review system is to create a new user that represents the SAST account, and set that account as default reviewer on every code review that is published to the server. When a new code review is created, the CI trigger should initiate an analysis job automatically. Then, glue should be created to allow the SAST code review account to read defect reports from the backend SAST server and insert those reports directly into the code review system. A nice addition to this setup is to allow the SAST account to +1 or -1 the code review, depending on whether or not the proposed change scanned clean.

This type of system is most effective when code reviews are a mandatory part of the development process. The main advantage is that developers are provided with near-instant automated feedback, which reduces the burden that SAST places on the process. Developers don't need to consult external SAST systems to learn whether their code change scanned clean. The results are put right in front of their face in the code review tool.

However, static analysis can sometimes run slowly. SAST should never be the bottleneck for the code review approval process, as any interference encountered by the developers will erode the tool's credibility. Therefore, it is best to run only the fast-and-accurate rulesets during this phase, ensuring that the SAST tool finishes analysis within the time window of a typical code review.

Few SAST solutions have out-of-the-box support for code review systems. It is likely that you will have to build this integration yourself, using the provided SAST API or command line interface.

6.3.5 Nightly Builds

In addition to integrating SAST with your continuous integration and code review system, you may also wish to integrate SAST with the nightly build system. This will be necessary because a small number of defects will not be caught by the SAST that runs within your CI system, due to the way independent commits tend to interact. Two separate commits can be analysed and determined to be free of defects, and only when those commits are combined will the defect be observed. A CI system that analyses changes independently will, of course, miss such defects. These are often referred to as "escaped" defects.

Unlike analysis performed during the code review phase, where speed is of the utmost importance, the overnight builds have the luxury of time. If you optimised the code review phase for speed, then the overnight phase should be configured to execute the slower rulesets that were previously disabled.

Be warned that extremely large codebases may not complete their analysis within an overnight timeframe. In such cases you might consider falling back to a once-weekly analysis that occurs over the weekend. Of course, this solution has one major drawback: it produces infrequent results, making it much more difficult for developers to integrate SAST with their daily routine.

Regardless of whether you perform nightly or weekly analysis, you should consider other speed-up options, such as the parallelisation of the analysis jobs. This can be accomplished by splitting your build process into small independent components that can analyse quickly. But if modifying the build

is not possible, then your last resort is to throw hardware at the problem, and investigate the use of analysis servers with an obscene number of cores and memory.

6.3.6 Distributed Analysis

You must ensure that SAST never becomes a bottleneck in the development process. As previously stated, one way to speed up the tool is to disable the slow rulesets when integrated with the continuous integration or peer code review systems. Another method of accomplishing this is to investigate the use of a dynamic elastic computing service.

The ebb and flow of software development is notoriously difficult to predict, and yet it has a massive impact on what kind of computing infrastructure you will need to ensure that your SAST service meets its SLA (e.g., all changes must be analysed within thirty minutes of commit, or the nightly analysis must complete in under twelve hours).

If you build out your infrastructure to match the average commit load, then your analysis will fall behind at peak times. If you build out your infrastructure to match peak loads, then your analysis servers will be idle much of the time. This is further exacerbated in large software organisations that operate in multiple time zones – there is never any idle time on the CI servers!

One example solution relies on dynamic elastic computing infrastructure. You will need a central server that speaks to your CI system, nightly build system, or code review system. This central server will, in turn, farm out jobs to dumb VMs that do the actual analysis. This requires each VM to have a copy of the master branch, as well as any partial or pending commits by developers.

This type of solution is not easily supported by all commercial SAST systems. It is likely you will need to proceed carefully here. NCC Group recommends that you run your own experiments, taking measurements of the average commit frequency and size to determine how best to scale the system.

6.3.7 Bug Trackers

Many commercial SAST tools ship with out-of-the-box support for common bug trackers such as JIRA or Bugzilla. These plugins allow defects to be migrated automatically from the SAST database to your company's bug tracker. This is often done for the sake of convenience, so that developers do not have to learn yet another bug triage system. In many cases, this can reduce friction between the developer and the SAST tool.

Take great care when deciding whether and how to integrate SAST with your bug tracker. It is strongly advised that you do not migrate all defects in bulk, as the first run of static analysis against a new code base can produce tens of thousands of warnings, many of which are likely inconsequential stylistic issues or false positives. You can easily overwhelm a development team with such a load, and most likely you will undermine your long-term success if you do this.

Defects should only be migrated to a bug tracker when they are too complicated to be resolved before check-in or within the scope of a short code review window. If set up properly, the static analysis that occurs as part of the nightly build should catch these escaped defects, and will automatically log them in the bug tracker.

6.3.8 Revision Control System

Integration with revision control systems, while not vital, is still a useful feature. Many SAST tools can interact directly with your source repository and extract data that can supplement a defect report. For example, this is most useful when automatically assigning SAST defects to specific developers, based on commit history for the affected source file.

However, be warned that this type of attribution system relies on a "best guess" heuristic, which can mistakenly assign a defect to the wrong developer when the issue arises from a combination of code changes rather than a single change. If defect assignment accuracy matters to you, then it is best to avoid integrating with revision control systems in this way.

6.4 The Analysis Engine

Under the hood, a SAST solution can rely on different sorts of code or binary analysis techniques, ranging from grep-like naïve pattern matching, to syntactically-aware control-flow analysis, to data-flow analysis that is capable of estimating and tracking value propagation through the program by building an abstract syntax tree. Each technique has its own advantages and disadvantages. Pattern-matching can be useful as a fast and efficient way of identifying the use of banned API functions. Data flow, while tending to be a slower form of static analysis, can identify more complex inter-procedural defects that require specific knowledge of a variable's value (or value range).

The analysis engine within a single SAST solution may make use of all of the above techniques. Therefore, the engine itself should be carefully studied to ensure a match for the needs of your organisation. You should check whether the tool is capable of detecting a wide range of vulnerabilities, whether it is accurate and precise, whether you can fine-tune the sensitivity of the built-in rulesets, whether you can suppress false positives, and whether you can customise or extend the engine to detect entirely new classes of vulnerabilities.

6.4.1 Defect Classes

First and foremost, when evaluating a SAST solution, you must carefully study its built-in rulesets. You will want to understand whether the tool detects a wide variety of security, quality, and privacy defects across all of the languages that matter most to you.

In addition to reviewing the defect classes for each supported language, you must also ensure that the analysis engine supports all frameworks that your software product relies upon. This is especially necessary in web application development, where the use of frameworks is quite common. A framework will frequently introduce architectural abstractions that can confuse a vanilla parser, or encourage programming patterns that can have unintentional security consequences. The SAST tool must support these frameworks, because you want to get the broadest coverage of all defect classes that affect your software product.

You should also consider comparing and contrasting the rulesets between multiple SAST solutions for a more accurate indicator of typical vulnerability detection support.

6.4.2 Accuracy and Precision

Measuring the true positive, false positive, and false negative rates can be challenging. And yet, knowing these numbers is essential when comparing multiple SAST solutions for purchase. As previously stated, you must be careful not to overwhelm your software engineers with spurious warnings (false positives). Likewise, you want to ensure the tool is not overlooking valuable security vulnerabilities (false negatives).

Measuring the false positive rate is quite possible, although it requires some effort and a little elbow grease. You should point the SAST tool at your natural codebase and spend a day or two triaging the reported defects. You don't need to triage all defects, just a representative sample from each category or ruleset that is reported by the tool. When you're done you should have a list of true positives and false positives, and will be able to extrapolate the false positive rate for the entire code base. Through this exercise you can also learn which rulesets are more or less accurate than others.

Measuring the false negative is a little harder, because you can't measure what's not there. One method of testing the false negative rate is by manually cross-referencing a series of previously-known defects from your bug tracker against the defects found by static code analysis. However, this is an *ad hoc* process that can be quite time consuming. An automated method would be more desirable.

For this reason, it is time to take a look at an artificial code base. The previously-mentioned Juliet and Toyota ITC test suites were specifically crafted to allow a scientific and methodical approach to measuring the accuracy and precision of SAST tools. The suite contains a set of "known answer" synthetic tests cases, comprising both "good code" and "flawed code". By inspecting whether or not a SAST tool detects defects in an artificial test suite, you can get an accurate representation of the

false negative rate. A framework for conducting this type of evaluation is thoroughly described by NIST[13] [14] [15].

6.4.3 Sensitivity

Whether the static analysis engine can be fine-tuned makes a big difference to its usefulness. Mature solutions will allow for adjustments to be made to the detection sensitivity for a variety of rules. Often this is accomplished by enabling additional inter-procedural value-tracking features, enabling false path pruning, or by modifying statistical thresholds. In most cases a higher accuracy comes at a cost of slower analysis. When evaluating a SAST system for wide deployment across your organisation, you will want to have a firm grasp of whether and how to adjust the engine's sensitivity.

You will want to adjust downwards, making the tool more accurate. In other words you want to strive for fewer false positives at the cost of more false negatives. Perhaps this is counterintuitive, but you must recognise that the average developer will not want to be bothered by false positives. When you strive for a high engagement rate among your engineering team, a low false positive rate matters most of all.

By following this advice you will introduce more false negatives. But those false negatives shouldn't be allowed to escape. During typical quality assurance activities, the testing and security teams will want to run the SAST tool in the most aggressive mode, ensuring that no critical defects were missed during development. These teams should be able to afford to deal with higher false positive rates.

6.4.4 False Positive Suppression

In addition to fine-tuning the ruleset sensitivity, another way to reduce noise in the SAST results is to outright suppress false positives or defect classes that are inconsequential due to the attack surface or threat model for your product or your team's coding style. In particular, it's crucial to look at how the tool allows false positives to be suppressed. Typically you will see four options for suppressing FPs, each with their own advantages and disadvantages:

(1) Passing suppression flags on the command line to the analysis engine,
(2) Supressing warnings via configuration files that live on the analysis server,
(3) Setting an individual defect as a false positive via the triage GUI,
(4) Placing specially-formatted comments directly in your code to instruct the analysis engine to ignore a certain warning type for a specific file, function, or line of code.

Wide-range suppression of entire classes of defects is best accomplished on the command line or using configuration files. Furthermore, the list of disabled rules should be owned by the central static analysis team, because the list is easier to audit when kept in a single place.

Perhaps the most convenient method is to suppress false positives in the SAST tool's built-in GUI. This allows anyone to audit the suppression settings, and leaves a convenient paper trail for each defect that is marked as an FP. Besides, developers typically won't have access to the server-side configuration files or the ability to change the tool's command line flags.

Suppressing rules or warnings by adding code-level comments can quickly get out of control, and before you know it, you'll have a maintenance problem on your hands: are those suppressions still valid? Has the code been refactored since the comment was added? These questions can be near impossible to answer when the code is many years old, or the primary developers have left the team. To avoid abuse of code-level suppression comments, a process should be instated that requires a peer review of all such comments before commit.

[13] http://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf
[14] http://samate.nist.gov/docs/SATE4/SATE%20IV%206%20Stick%20to%20Facts%20II%20Erno.pdf
[15] http://samate.nist.gov/docs/NIST_Special_Publication_500-297.pdf

When deploying a SAST solution, you must think about your preferences for suppressing false positives. Best practice typically states that a suppression hierarchy is needed, where it is preferred to suppress certain types of warnings in config files, others on command line, using code-level comments, or with the triage GUI. It is important that you establish these criteria before deploying SAST widely across your organisation.

6.4.5 Dealing with Legacy Defects

The problem of dealing with legacy defects is summarised well by Walter W. Schilling in his paper titled "Integrate static analysis into software development"[16]:

> *Applying static analysis to existing code that's not intended for obsolescence can be challenging. Depending on the age of the code, the engineer's programming style, and the paradigms used, applying static analysis to existing code can range from difficult to nearly impossible if a disciplined approach isn't followed. Many legacy projects have approached static analysis only to abandon it when the first run of the tool generates 100,000 or more warnings. With legacy code, it's often not practical to remove all statically detectable faults.*

In other words, you must develop a strategy to deal with legacy defects. You must keep in mind that while many of these defects are legitimate, you introduce risk by attempting to develop patches. For example, there is probably some amount of legacy code whose maintainers are no long employed at your company, while other software components may be fragile due to an inadequate level of unit test coverage.

So what do you do? Best practice sometimes states that all legacy defects should be hidden or suppressed from view of the development team. Once again, while counterintuitive, this is about reducing the burden placed on the average developer, and avoiding "needle in the haystack" type of problems.

A "zero new defect" policy should be introduced that at least requires all *new* software to scan clean in your SAST tool. Finally, the quality assurance or security teams should be tasked with crawling through the legacy defect backlog to isolate the most critical security vulnerabilities, leaving the rest alone. These policies should be established before SAST is deployed across your organisation.

6.4.6 Creating Custom Rulesets

It is essential that you consider creating customised rulesets that are capable of detecting emergent classes of security vulnerabilities, or simply to catch defects that are unique to your development team and its coding style guide. This is often important when reacting to previously-unseen security defects, such as those disclosed during security incidents.

In these situations you will need to check whether the newly discovered flawed coding pattern is repeated anywhere else in your large codebase. For example, in recent memory, there was a rush by commercial SAST vendors to detect the general patterns exhibited by the 'Heartbleed' [17] [18] [19] and 'Goto Fail' [20] [21] vulnerabilities. Any time your organisation is hit by a security incident, you will want to create a custom ruleset and hunt for variations of the underlying defect. Static code analysis can be extremely effective at automating a find-it-once-fix-it-forever approach to security incident response.

You will want to understand how custom rulesets are defined, and even try writing a few rules yourself during the evaluation period. Often, rules are defined in one of two ways: (1) By using the

---

[16] http://m.eetindia.co.in/ARTICLES/2006NOV/PDF/EEIOL_2006NOV05_EMS_INTD_TA.pdf
[17] http://blog.regehr.org/archives/1125
[18] http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html
[19] http://blog.klocwork.com/software-security/saving-you-from-heartbleed/
[20] http://security.coverity.com/blog/2014/Feb/a-quick-post-on-apple-security-55471-aka-goto-fail.html
[21] http://blog.klocwork.com/static-analysis/rather-than-fail-goto-success/

SAST tool's analysis engine hooks and writing rules that are actually little C or Java programs, or (2) defining the new rule in an XML or other type of text file.

The flexibility of the extension language is of key importance. You will want it to support a variety of rule types, from simple grep-like pattern matching to full intra- and inter-procedural data-flow analysis. For each custom rule you create, you should also consider writing unit tests that serve a similar purpose to the Juliet or Toyota ITC test suites. Each test should be a variation on the pattern the rule is trying to match. This will help keep the false positive and false negative rates under control. These test cases will also assist you when evaluating updates to the SAST engine when you wish to understand whether backwards compatibility was maintained.

It is also desirable for the analysis engine to provide the ability to override any default or built-in rules and models. For example, maybe your company has developed its own custom heap implementation, in which case you must train the engine to recognise your customised `mymalloc` and `myfree` routines, or else it will never catch any heap corruption vulnerabilities in your code base.

Being able to customise the analysis engine is an essential aspect of reducing your false negative rate and supporting your security incident response activities.

6.4.7 Analysis Speed

A comparison of multiple SAST solutions should also consider the speed of analysis. Some tools that wrap and monitor the build process will add considerable overhead, sometimes increasing the build time by a factor of two or more. Likewise, the speed of analysis is also important. The range here is massive, with different tools varying between hundreds and thousands of lines of code analysed per second.

SAST tools use complex algorithms and analysis methods that can run extremely slow, or use large amounts of memory. These factors will result in slow analysis or memory exhaustion problems. You *will* be forced to make SAST configuration decisions that trade precision for speed.

Many commercial SAST tools now support parallel analysis, which goes a long way towards speeding up the process. A relatively new feature that is coming to commercial SAST tools is known as "incremental analysis", which also helps to speed up analysis times. This works by caching the intermediate outputs from previous analysis jobs, avoiding the need to execute a monolithic clean build each time you want to analyse your code. This is most useful when developers run SAST locally on their desktops or when integrating with the peer code review system, where speed is crucial.

While running experiments against your natural and artificial codebases, you should measure the speed of the SAST tool. It is useful to measure the effect of configuration changes as well. Turning some rulesets on or off can have significant impact on the runtime. This is useful when you wish to enforce SLAs that convey performance goals to your users, or when you want to make predictions about the server infrastructure investment that will be required upon wide deployment of the SAST tool.

6.5 Defect Triage Maturity

When comparing SAST solutions, the maturity of the built-in defect triage process should also be carefully studied to understand whether it is compatible with your existing procedures. A cumbersome or inflexible triage workflow can make or break the adoption by users. The various aspects of the triage workflow that need to be studied are summarised in the following sections.

6.5.1 Severity

Some SAST tools allow for a rule's default severity to be changed. This is helpful because you will often disagree with the pre-defined defaults. If you wish to create policies such as "Our product will not ship until all critical defects are resolved", then it is important that you agree with your SAST tool about what a critical defect actually means! Auditing each rule in this way can be a tedious process, but the payoff is worth it.

6.5.2 Detailed Trace and Remediation Guidance

Another relevant factor that affects the triage maturity is whether the SAST tool emits a detailed and comprehensive trace of the defect, and whether the remediation guidance is clear and actionable. You will want to review the SAST solution's built-in defect taxonomy to ensure that the guidance is comprehensive, compact and precise.

Regarding the defect trace, some systems will inscrutably state "`Defect XXX in file YYY on line # ZZZ`", which leaves a lot to be desired. The most effective tools will include a detailed trace that describes each conditional branch that was taken to arrive at the vulnerability, including the values (or estimated value ranges) of the tracked variables. The more detail the better.

You should view the remediation guidance as an opportunity to teach developers about software security. For this reason, clear and actionable documentation is imperative. Using SAST everyday can be immensely beneficial for their security training. Some tools also provide links to external supplementary guidance such as OWASP policies, which can also benefit developer training.

Finally, few solutions will actually allow you to modify the remediation guidance. This is important when you disagree with the predefined guidance, or when your organisation has extra steps that you want your engineers to follow when patching a certain type of defect.

6.5.3 Collapse Similar Defects

The defect backlog and triage workload can be reduced considerably when the SAST system intelligently collapses similar or identical defects into a single issue. Often, especially after the tool is run for the first time, you will find that dozens or hundreds of defects can be squashed with a single one-line fix. Isolating and fixing these defects first can help reduce the needle-in-a-haystack problem that is common with first-time SAST users.

6.5.4 Support for Multiple Branches

With the rise of Git, branching is easy and has become an everyday practice. You will want to understand how the SAST tool supports multiple branches and whether it is compatible with your organisation's branching strategy. Many SAST tools have recently attempted to solve the problem of multiple branches, but these features have not yet fully matured.

Questions worth asking are: How do I ensure that all branches are analysed by my SAST tool? Does the tool merge duplicate defects in shared code that are identical across branches? If the defect is fixed in one branch, is it still reported in the other branches?

6.5.5 Triage Workflow and Multi-User Collaboration

Mature SAST solutions will frequently provide a web-based GUI that serves as the presentation layer on top of the defect database. This GUI will impose a triage workflow that may or may not be compatible with your team's own defect triage process. If you find that this workflow is incompatible, then it is worth attempting to understand the tool's database schema or report generation APIs, as you may need to export the defect database into an alternate work-tracking system.

If you decide to use the built-in triage workflow, it is useful to know whether multiple users can interact and leave comments on a single defect. In addition to allowing engineers to collaborate (as they would in a peer code review system like Gerrit) it also adds a meaningful level of traceability when the results are audited later. This ensures that important conversations are not brought offline.

6.5.6 Isolate Users or Teams

Not all users will want to see every defect in the repository each time they log into the SAST web interface. In fact, seeing every defect can be overwhelming to some developers. The SAST tool should provide the option to isolate users to just the warnings that reside in the trees for which they are responsible.

An added benefit of mapping directories to teams is that you can begin generating metrics based on which teams have the highest engagement levels or lowest defect densities with the SAST service. Be careful of sharing these metrics with the users as they may act as perverse incentives, triggering behaviours that are good for gaming the statistics but bad for the real goal of improving software security.

6.5.7 Report Generation

It is not good enough simply to deploy the SAST solution; you will wish to measure carefully whether the tool is functioning effectively and as intended. Most likely it is not. There will be some sore spots – isolated teams with low engagement, or code areas with especially high false positive rates.

You may wish to measure the user engagement and adoption rates, as well as false positive rates, and defect trends either by severity, or by team. Through careful measurement of these data points, your organisation can recognise problems in your SAST deployment and quickly respond with corrective actions. This sort of continual improvement behaviour is a core tenet of the Agile methodology.

The SAST solution must support the ability to generate reports, although it is very likely the tool will not support generation for all types of reports you are interested in. So another key feature is the ability to export raw data so you can conduct your own analysis. Below are some example reports that, in NCC Group's experience, have proven to be effective:

User Engagement, Adoption, and Perception

Report	Corrective action
Measure the percentage of the overall code base that is scanned by SAST.	For each unscanned code area, identify the owners, figure out why the code was missed, and work to migrate them under the purview of SAST.
List of developers on each team that do not have accounts on the SAST server, or users that do not regularly log into the server.	Identify disengaged individuals or teams that can be targeted for additional security training. Or perhaps some other corrective action is needed, such as reducing the FP rate.

Also consider routinely sending a survey to employees in the software organisation. The survey should solicit details on perception, engagement, and so on. This method can be extremely effective at allowing developers to vent their frustration at how SAST interferes with their everyday job. All responses should be taken seriously; remember, the only way to succeed with SAST is by winning over the developers, and this is best accomplished by removing friction between them and the tool.

Security and Code Quality Metrics

Report	Corrective action
Measure the total count of unresolved defects.	For each product or codebase, the count of unresolved defects should always trend downwards.
Measure the user-reported false positive rate.	The false positive rate should remain below X%. If the rate is too high, then corrective action should be taken to disable noisy rules.

Report	Corrective action
Measure the types and quantity of observed defects.	If a certain type of defect is pervasive across your organisation, this can indicate a poor coding style guide, or that specific targeted training is necessary for your engineering staff.
Measure the defect density (# defects / KLOC).	The defect density should always trend downwards, or remain below a set threshold. If not, this means developers are introducing new defects and ignoring the SAST tool.
All software products will ship with zero critical defects.	Setting this type of release quality metric must be done from the top down. Executives must agree that static analysis matters, and product quality must be measured by the results of the tool.

Service Level Agreement

You must make a commitment to your users that the tool will be available when needed, and that corrective action to address false positives will happen with expedience.

Report	Corrective action
The SAST service will be available X% of the time.	You probably do not need a Six Sigma uptime SLA for your internal SAST service, but you should at a minimum ensure that the service is never down during the workday across all time zones where your developers are based.
90% of changes will be analysed within thirty minutes of the creation of a peer code review.	You never want the SAST tool to be the bottleneck for the code review process. Therefore, you should set an aggressive SLA here.
False positives, once identified, will be suppressed in under X days.	False positives should not sit in the backlog, as they only remind users how the SAST service is inaccurate at times. If a pattern of false positives is identified, then you must configure the tool to suppress that category of warning.
Every SAST configuration change will be thoroughly tested to measure the impact on the FP and FN rates.	If you create a custom rule, or modify the SAST server configuration, you must test your changes on both natural and artificial code. A change made blindly raises the risk of introducing FPs or FNs.

6.6 Vendor Roadmap

A good commercial SAST vendor should be able to show you a roadmap for their product. A great vendor will have a roadmap that extends multiple years into the future. A poorly-formed roadmap (or unwillingness to share the roadmap) might indicate a poor vision for the tool. You are, after all, about to make a considerable monetary and time investment in the evaluation and deployment of a SAST

solution, so these things matter. You want to make sure that you can continue using the tool for years to come, without being forced to switch.

You should also ask to see the changelog for the last few releases. Red flags should be raised if the vendor frequently breaks backwards compatibility, provides customer updates infrequently, or rarely makes improvements to the defect detection capabilities or analysis engine.

7 Final Words

This paper distils the author's decade of experience in using numerous static code analysis tools. Deploying SAST widely across a large software development organisation is no small feat, and great care must be taken from the very beginning and throughout the entire deployment process. The solution must be fast, accurate, and most importantly it should not impose a burden on the developers who are required use the tool as part of their daily routine. These problems are not insurmountable, although they do require vigilant planning and creative problem solving, as each organisation will have a unique approach to its software development procedures. In other words, no commercial SAST solution is one-size-fits-all. NCC Group believes that by following the guidance and best practices that are outlined here, you are more likely to achieve success in the use of static code analysis within your organisation.