3 Static Analysis as Part of the Code Review Process

In preparing for battle, plans are useless
but planning is indispensable.
—Dwight Eisenhower

There's a lot to know about how static analysis tools work. There's probably just as much to know about making static analysis tools work as part of a secure development process. In this respect, tools that assist with security review are fundamentally different than most other kinds of software development tools. A debugger, for example, doesn't require any organization-wide planning to be effective. An individual programmer can run it when it's needed, obtain results, and move on to another programming task. But the need for software security rarely creates the kind of urgency that leads a programmer to run a debugger. For this reason, an organization needs a plan for who will conduct security reviews, when the reviews will take place, and how to act on the results. Static analysis tools should be part of the plan because they can make the review process significantly more efficient.

Code review is a skill. In the first part of this chapter, we look at what that skill entails and outline the steps involved in performing a code review. We pay special attention to the most common snag that review teams get hung up on: debates about exploitability. In the second part of the chapter, we look at who needs to develop the code review skill and when they need to apply it. Finally, we look at metrics that can be derived from static analysis results.

3.1 Performing a Code Review

A security-focused code review happens for a number of different reasons:

- Some reviewers start out with the need to find a few exploitable vulnerabilities to prove that additional security investment is justified.
- For every large project that didn't begin with security in mind, the team eventually has to make an initial pass through the code to do a security retrofit.
- At least once in every release period, every project should receive a security review to account for new features and ongoing maintenance work.

Of the three, the second requires by far the largest amount of time and energy. Retrofitting a program that wasn't written to be secure can be a considerable amount of work. Subsequent reviews of the same piece of code will be easier. The initial review likely will turn up many problems that need to be addressed. Subsequent reviews should find fewer problems because programmers will be building on a stronger foundation.

Steve Lipner estimates that at Microsoft security activities consume roughly 20% of the release schedule the first time a product goes through Microsoft's Security Development Lifecycle. In subsequent iterations, security requires less than 10% of the schedule [Lipner, 2006]. Our experience with the code review phase of the security process is similar—after the backlog of security problems is cleared out, keeping pace with new development requires much less effort.

The Review Cycle

We begin with an overview of the code review cycle and then talk about each phase in detail. The four major phases in the cycle are:

1. Establish goals
2. Run the static analysis tool
3. Review code (using output from the tool)
4. Make fixes

Figure 3.1 shows a few potential back edges that make the cycle a little more complicated than a basic box step. The frequency with which the cycle is repeated depends largely upon the goals established in the first phase, but our experience is that if a first iteration identifies more than a handful of security problems, a second iteration likely will identify problems too.

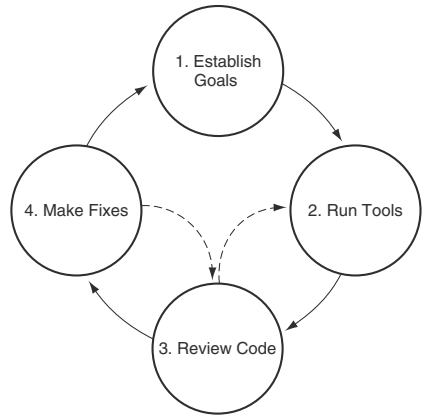

Figure 3.1 The code review cycle.

Later in the chapter, we discuss when to perform code review and who should do the reviewing, but we put forth a typical scenario here to set the stage. Imagine the first iteration of the cycle being carried out midway through the time period allocated for coding. Assume that the reviewers are programmers who have received security training.

1. Establish Goals

A well-defined set of security goals will help prioritize the code that should be reviewed and criteria that should be used to review it. Your goals should come from an assessment of the software risks you face. We sometimes hear sweeping high-level objectives along these lines:

- "If it can be reached from the Internet, it has to be reviewed before it's released."

  or

- "If it handles money, it has to be reviewed at least once a year."

We also talk to people who have more specific tactical objectives in mind. A short-term focus might come from a declaration:

- "We can't fail our next compliance audit. Make sure the auditor gives us a clean bill of health."

  or

- "We've been embarrassed by a series of cross-site scripting vulnerabilities. Make it stop."

You need to have enough high-level guidance to prioritize your potential code review targets. Set review priorities down to the level of individual programs. When you've gotten down to that granularity, don't subdivide any further; run static analysis on at least a whole program at a time. You might choose to review results in more detail or with greater frequency for parts of the program if you believe they pose more risk, but allow the tool's results to guide your attention, at least to some extent. At Fortify, we conduct line-by-line peer review for components that we deem to be high risk, but we always run tools against all of the code.

When we ask people what they're looking for when they do code review, the most common thing we hear is, "Uh, err, the OWASP Top Ten?" Bad answer. The biggest problem is the "?" at the end. If you're not too sure about what you're looking for, chances are good that you're not going to find it. The "OWASP Top Ten" part isn't so hot, either. Checking for the OWASP Top Ten is part of complying with the Payment Card Industry (PCI) Data Security Standard, but that doesn't make it the beginning and end of the kinds of problems you should be looking for. If you need inspiration, examine the results of previous code reviews for either the program you're planning to review or similar programs. Previously discovered errors have an uncanny way of slipping back in. Reviewing past results also gives you the opportunity to learn about what has changed since the previous review.

Make sure reviewers understand the purpose and function of the code being reviewed. A high-level description of the design helps a lot. It's also the right time to review the risk analysis results relevant to the code. If reviewers don't understand the risks before they begin, the relevant risks will inevitably be determined in an ad-hoc fashion as the review proceeds. The results will be less than ideal because the collective opinion about what is acceptable and what is unacceptable will evolve as the review progresses. The "I'll know a security problem when I see it" approach doesn't yield optimal results.

2. Run Static Analysis Tools

Run static analysis tools with the goals of the review in mind. To get started, you need to gather the target code, configure the tool to report the kinds of problems that pose the greatest risks, and disable checks that aren't relevant. The output from this phase will be a set of raw results for use during code review. Figure 3.2 illustrates the flow through phases 2 and 3.

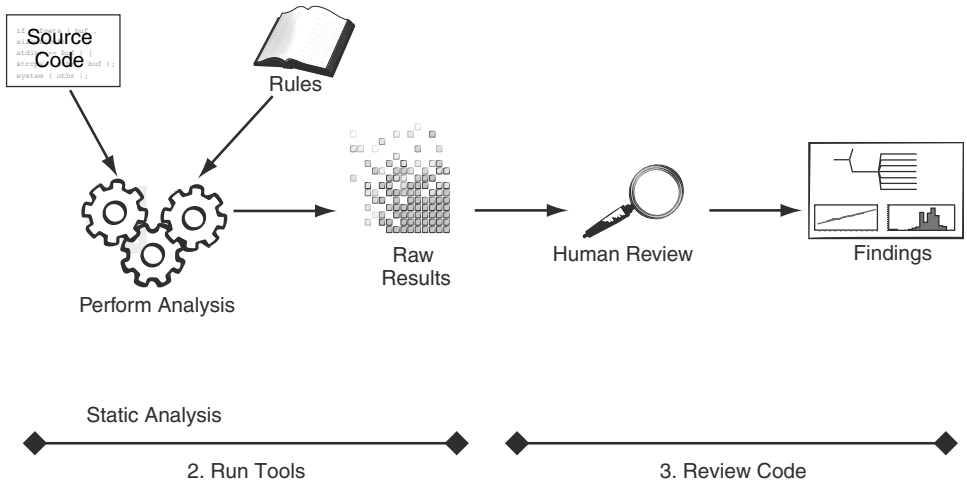

Figure 3.2 Steps 2 and 3: running the tool and reviewing the code.

To get good results, you should be able to compile the code being analyzed. For development groups operating in their own build environment, this is not much of an issue, but for security teams who've had the code thrown over the wall to them, it can be a really big deal. Where are all the header files? Which version of that library are you using? The list of snags and roadblocks can be lengthy. You might be tempted to take some shortcuts here. A static analysis tool can often produce at least some results even if the code doesn't compile. Don't cave. *Get the code into a compilable state before you analyze it.* If you get into the habit of ignoring parse errors and resolution warnings from the static analysis tool, you'll eventually miss out on important results.

This is also the right time to add custom rules to detect errors that are specific to the program being analyzed. If your organization has a set of secure coding guidelines, go through them and look for things you can encode as custom rules. A static analysis tool won't, by default, know what constitutes a security violation in the context of your code. Chances are good that you can dramatically improve the quality of the tool's results by customizing it for your environment.

Errors found during previous manual code reviews are particularly useful here, too. If a previously identified error can be phrased as a violation of some program invariant (never do X, or always do Y), write a rule to detect

similar situations. Over time, this set of rules will serve as a form of institutional memory that prevents previous security slip-ups from being repeated.

3. Review Code

Now it's time to review the code with your own eyes. Go through the static analysis results, but don't limit yourself to just analysis results. Allow the tool to point out potential problems, but don't allow it to blind you to other problems that you can find through your own inspection of the code. We routinely find other bugs right next door to a tool-reported issue. This "neighborhood effect" results from the fact that static analysis tools often report a problem when they become confused in the vicinity of a sensitive operation. Code that is confusing to tools is often confusing to programmers, too, although not always for the same reasons. Go through all the static analysis results; don't stop with just the high-priority warnings. If the list is long, partition it so that multiple reviewers can share the work.

Reviewing a single issue is a matter of verifying the assumptions that the tool made when it reported the issue. Do mitigating factors prevent the code from being vulnerable? Is the source of untrusted data actually untrusted? Is the scenario hypothesized by the tool actually feasible?[1] If you are reviewing someone else's code, it might be impossible for you to answer all these questions, and you should collaborate with the author or owner of the code. Some static analysis tools makes it easy to share results (for instance, by publishing an issue on an internal Web site), which simplifies this process.

Collaborative auditing is a form of peer review. Structured peer reviews are a proven technique for identifying all sorts of defects [Wiegers, 2002; Fagan, 1976]. For security-focused peer review, it's best to have a security specialist as part of the review team. Peer review and static analysis are complimentary techniques. When we perform peer reviews, we usually put one reviewer in charge of going through tool output.

If, during the review process, you identify a problem that wasn't found using static analysis, return to step 2: Write custom rules to detect other instances of the same problem and rerun the tools. Human eyes are great for spotting new varieties of defects, and static analysis excels at making sure that every instance of those new problems has been found. The back edge from step 3 to step 2 in Figure 3.1 represents this work.

---

1. Michael Howard outlines a structured process for answering questions such as these in a security and privacy article entitled "A Process for Performing Security Code Reviews" [Howard, 2006].

Code review results can take a number of forms: bugs entered into the bug database, a formal report suitable for consumption by both programmers and management, entries into a software security tracking system, or an informal task list for programmers. No matter what the form is, make sure the results have a permanent home so that they'll be useful during the next code review. Feedback about each issue should include a detailed explanation of the problem, an estimate of the risk it brings, and references to relevant portions of the security policy and risk assessment documents. This permanent collection of review results is good for another purpose, too: input for security training. You can use review results to focus training on real problems and topics that are most relevant to your code.

4. Make Fixes

Two factors control the way programmers respond to the feedback from a security review:

- Does security matter to them? If getting security right is a prerequisite for releasing their code, it matters. Anything less is shaky ground because it competes with adding new functionality, fixing bugs, and making the release date.
- Do they understand the feedback? Understanding security issues requires security training. It also requires the feedback to be written in an intelligible manner. Results stemming from code review are not concrete the way a failing test case is, so they require a more complete explanation of the risk involved.

If security review happens early enough in the development lifecycle, there will be time to respond to the feedback from the security review. Is there a large clump of issues around a particular module or a particular feature? It might be time to step back and look for design alternatives that could alleviate the problem. Alternatively, you might find that the best and most lasting fix comes in the form of additional security training.

When programmers have fixed the problems identified by the review, the fixes must be verified. The form that verification takes depends on the nature of the changes. If the risks involved are not small and the changes are nontrivial, return to the review phase and take another look at the code. The back edge from step 4 to step 3 in Figure 3.1 represents this work.

Steer Clear of the Exploitability Trap

Security review should not be about creating flashy exploits, but all too often, review teams get pulled down into exploit development. To understand why, consider the three possible verdicts that a piece of code might receive during a security review:

- Obviously exploitable
- Ambiguous
- Obviously secure

No clear dividing line exists between these cases; they form a spectrum. The endpoints on the spectrum are less trouble than the middle; obviously exploitable code needs to be fixed, and obviously secure code can be left alone. The middle case, ambiguous code, is the difficult one. Code might be ambiguous because its logic is hard to follow, because it's difficult to determine the cases in which the code will be called, or because it's hard to see how an attacker might be able to take advantage of the problem.

The danger lies in the way reviewers treat the ambiguous code. If the onus is on the reviewer to prove that a piece of code is exploitable before it will be fixed, the reviewer will eventually make a mistake and overlook an exploitable bug. When a programmer says, "I won't fix that unless you can prove it's exploitable," you're looking at the exploitability trap. (For more ways programmers try to squirm out of making security fixes, see the sidebar "Five Lame Excuses for Not Fixing Bad Code.")

The exploitability trap is dangerous for two reasons. First, developing exploits is time consuming. The time you put into developing an exploit would almost always be better spent looking for more problems. Second, developing exploits is a skill unto itself. What happens if you can't develop an exploit? Does it mean the defect is not exploitable, or that you simply don't know the right set of tricks for exploiting it?

Don't fall into the exploitability trap: Get the bugs fixed!

If a piece of code isn't obviously secure, make it obviously secure. Sometimes this approach leads to a redundant safety check. Sometimes it leads to a comment that provides a verifiable way to determine that the code is okay. And sometimes it plugs an exploitable hole. Programmers aren't always wild about the idea of changing a piece of code when no error can be demonstrated because any change brings with it the possibility of introducing a new bug. But the alternative—shipping vulnerabilities—is even less attractive.

Beyond the risk that an overlooked bug might eventually lead to a new exploit is the possibility that the bug might not even need to be exploitable

to cause damage to a company's reputation. For example, a "security researcher" who finds a new buffer overflow might be able to garner fame and glory by publishing the details, even if it is not possible to build an attack around the bug [Wheeler, 2005]. Software companies sometimes find themselves issuing security patches even though all indications are that a defect isn't exploitable.

Five Lame Excuses for Not Fixing Bad Code

Programmers who haven't figured out software security come up with some inspired reasons for not fixing bugs found during security review. "I don't think that's exploitable" is the all-time winner. All the code reviewers we know have their own favorite runners-up, but here are our favorite specious arguments for ignoring security problems:

1. **"I trust system administrators."**

Even though I know they've misconfigured the software before, I know they're going to get it right this time, so I don't need code that verifies that my program is configured reasonably.

2. **"You have to authenticate before you can access that page."**

How on earth would an attacker ever get a username and a password? If you have a username and a password, you are, by definition, a good guy, so you won't attack the system.

3. **"No one would ever think to do that!"**

The user manual very clearly states that names can be no longer than 26 characters, and the GUI prevents you from entering any more than 26 characters. Why would I need to perform a bounds check when I read a saved file?

4. **"That function call can never fail."**

I've run it a million times on my Windows desktop. Why would it fail when it runs on the 128 processor Sun server?

5. **"We didn't intend for that to be production-ready code."**

Yes, we know it's been part of the shipping product for several years now, but when it was written, we didn't expect it to be production ready, so you should review it with that in mind.

3.2 Adding Security Review to an Existing Development Process[2]

It's easy to talk about integrating security into the software development process, but it can be a tough transition to make if programmers are in the habit of ignoring security. Evaluating and selecting a static analysis tool can be the easiest part of a software security initiative. Tools can make programmers more efficient at tackling the software security problem, but tools alone cannot solve the problem. In other words, static analysis should be used as part of a secure development lifecycle, not as a replacement for a secure development lifecycle.

Any successful security initiative requires that programmers buy into the idea that security is important. In traditional hierarchical organizations, that usually means a dictum from management on the importance of security, followed by one or more signals from management that security really should be taken seriously. The famous 2002 memo from Bill Gates titled "Trustworthy Computing" is a perfect example of the former. In the memo, Gates wrote:

> So now, when we face a choice between adding features and resolving security issues, we need to choose security.

Microsoft signaled that it really was serious about security when it called a halt to Windows development in 2002 and had the entire Windows division (upward of 8,000 engineers) participate in a security push that lasted for more than two months [Howard and Lipner, 2006].

Increasingly, the arrival of a static analysis tool is part of a security push. For that reason, adoption of static analysis and adoption of an improved process for security are often intertwined. In this section, we address the hurdles related to tool adoption. Before you dive in, read the adoption success stories in the sidebar "Security Review Times Two."

Security Review Times Two

Static analysis security tools are new enough that, to our knowledge, no formal studies have been done to measure their impact on the software built by large organizations. But as part of our work at Fortify, we've watched closely as our customers have rolled out our tools to their development teams and security organizations. Here we describe

2. This section began as an article in *IEEE Security & Privacy Magazine*, co-authored with Pravir Chandra and John Steven [Chandra, Chess, Steven, 2006].

the results we've seen at two large financial services companies. Because the companies don't want their names to be used, we'll call them "East Coast" and "West Coast."

East Coast

A central security team is charged with doing code review. Before adopting a tool, the team reviewed 10 million lines of code per year. With Fortify, they are now reviewing 20 million lines of code per year. As they have gained familiarity with static analysis, they have written custom rules to enforce larger portions of their security policy. The result is that, as the tools do more of the review work, the human reviewers continue to become more efficient. In the coming year, they plan to increase the rate of review to 30 million lines of code per year without growing the size of the security team.

Development groups at the company are starting to adopt the tool, too; more than 100 programmers use the tool as part of the development process, but the organization has not yet measured the impact of developer adoption on the review process.

West Coast

A central security team is charged with reviewing all Internet-facing applications before they go to production. In the past, it took the security team three to four weeks to perform a review. Using static analysis, the security team now conducts reviews in one to two weeks. The security team expects to further reduce the review cycle time by implementing a process wherein the development team can run the tool and submit the results to the security team. (This requires implementing safeguards to ensure that the development team runs the analysis correctly.) The target is to perform code review for most projects in one week.

The security team is confident that, with the addition of source code analysis to the review process, they are now finding 100% of the issues in the categories they deem critical (such as cross-site scripting). The previous manual inspection process did not allow them to review every line of code, leaving open the possibility that some critical defects were being overlooked.

Development teams are also using static analysis to perform periodic checks before submitting their code to the security team. Several hundred programmers have been equipped with the tool. The result is that the security team now finds critical defects only rarely. (In the past, finding critical defects was the norm.) This has reduced the number of schedule slips and the number of "risk-managed deployments" in which the organization is forced to field an application with known vulnerabilities. The reduction in critical defects also significantly improves policy enforcement because when a security problem does surface, it now receives appropriate attention.

As a side benefit, development teams report that they routinely find non-security defects as a result of their code review efforts.

Adoption Anxiety

All the software development organizations we've ever seen are at least a little bit chaotic, and changing the behavior of a chaotic system is no mean feat. At first blush, adopting a static analysis tool might not seem like much of a problem. Get the tool, run the tool, fix the problems, and you're done. Right? Wrong. It's unrealistic to expect attitudes about security to change just because you drop off a new tool. Adoption is not as easy as leaving a screaming baby on the doorstep. Dropping off the tool and waving goodbye will lead to objections like the ones in Table 3.1.

Table 3.1 Commonly voiced objections to static analysis and their true meaning.

Objection	Translation
"It takes too long to run."	"I think security is optional, and since it requires effort, I don't want to do it."
"It has too many false positives."	"I think security is optional, and since it requires effort, I don't want to do it."
"It doesn't fit in to the way I work."	"I think security is optional, and since it requires effort, I don't want to do it."

In our experience, three big questions must be answered to adopt a tool successfully. An organization's size, along with the style and maturity of its development processes, all play heavily into the answers to these questions. None of them has a one-size-fits-all answer, so we consider the range of likely answers to each. The three questions are:

- Who runs the tool?
- When is the tool run?
- What happens to the results?

Who Runs the Tool?

Ideally, it wouldn't matter who actually runs the tool, but a number of practical considerations make it an important question, such as access to the code. Many organizations have two obvious choices: the security team or the programmers.

The Security Team

For this to work, you must ensure that your security team has the right skill set—in short, you want security folks with software development chops. Even if you plan to target programmers as the main consumers of the information generated by the tool, having the security team participate is a huge asset. The team brings risk management experience to the table and can often look at big-picture security concerns, too. But the security team didn't write the code, so team members won't have as much insight into it as the developers who did. It's tough for the security team to go through the code alone. In fact, it can be tricky to even get the security team set up so that they can compile the code. (If the security team isn't comfortable compiling other people's code, you're barking up the wrong tree.) It helps if you already have a process in place for the security team to give code-level feedback to programmers.

The Programmers

Programmers possess the best knowledge about how their code works. Combine this with the vulnerability details provided by a tool, and you've got a good reason to allow development to run the operation. On the flip side, programmers are always under pressure to build a product on a deadline. It's also likely that, even with training, they won't have the same level of security knowledge or expertise as members of the security team. If the programmers will run the tool, make sure they have time built into their schedule for it, and make sure they have been through enough security training that they'll be effective at the job. In our experience, not all programmers will become tool jockeys. Designate a senior member of each team to be responsible for running the tool, making sure the results are used appropriately, and answering tool-related questions from the rest of the team.

All of the Above

A third option is to have programmers run the tools in a mode that produces only high-confidence results, and use the security team to conduct more thorough but less frequent reviews. This imposes less of a burden on the programmers, while still allowing them to catch some of their own mistakes. It also encourages interaction between the security team and the development team. No question about it, joint teams work best. Every so

often, buy some pizzas and have the development team and the security team sit down and run the tool together. Call it eXtreme Security, if you like.

When Is the Tool Run?

More than anything else, deciding when the tool will be run determines the way the organization approaches security review. Many possible answers exist, but the three we see most often are these: while the code is being written, at build time, and at major milestones. The right answer depends on how the analysis results will be consumed and how much time it takes to run the tool.

While the Code Is Being Written

Studies too numerous to mention have shown that the cost of fixing a bug increases over time, so it makes sense to check new code promptly. One way to accomplish this is to integrate the source code analysis tool into the programmer's development environment so that the programmer can run on-demand analysis and gain expertise with the tool over time. An alternate method is to integrate scanning into the code check-in process, thereby centralizing control of the analysis. (This approach costs the programmers in terms of analysis freedom, but it's useful when desktop integration isn't feasible.) If programmers will run the tool a lot, the tool needs to be fast and easy to use. For large projects, that might mean asking each developer to analyze only his or her portion of the code and then running an analysis of the full program at build time or at major milestones.

At Build Time

For most organizations, software projects have a well-defined build process, usually with regularly scheduled builds. Performing analysis at build time gives code reviewers a reliable report to use for direct remediation, as well as a baseline for further manual code inspection. Also, by using builds as a timeline for source analysis, you create a recurring, consistent measure of the entire project, which provides perfect input for analysis-driven metrics. This is a great way to get information to feed a training program.

At Major Milestones

Organizations that rely on heavier-weight processes have checkpoints at project milestones, generally near the end of a development cycle or at some large interval during development. These checkpoints sometimes include

security-related tasks such as a design review or a penetration test. Logically extending this concept, checkpoints seem like a natural place to use a static analysis tool. The down side to this approach is that programmers might put off thinking about security until the milestone is upon them, at which point other milestone obligations can push security off to the sidelines. If you're going to wait for milestones to use static analysis, make sure you build some teeth into the process. The consequences for ignoring security need to be immediately obvious and known to all ahead of time.

What Happens to the Results?

When people think through the tool adoption process, they sometimes forget that most of the work comes after the tool is run. It's important to decide ahead of time how the actual code review will be performed.

Output Feeds a Release Gate

The security team processes and prioritizes the tool's output as part of a checkpoint at a project milestone. The development team receives the prioritized results along with the security team's recommendations about what needs to be fixed. The development team then makes decisions about which problems to fix and which to classify as "accepted risks." (Development teams sometimes use the results from a penetration test the same way.) The security team should review the development team's decisions and escalate cases where it appears that the development team is taking on more risk than it should. If this type of review can block a project from reaching a milestone, the release gate has real teeth. If programmers can simply ignore the results, they will have no motivation to make changes.

The gate model is a weak approach to security for the same reason that penetration testing is a weak approach to security: It's reactive. Even though the release gate is not a good long-term solution, it can be an effective stepping stone. The hope is that the programmers will eventually get tired of having their releases waylaid by the security team and decide to take a more proactive approach.

A Central Authority Doles Out Individual Results

A core group of tool users can look at the reported problems for one or more projects and pick the individual issues to send to the programmers responsible for the code in question. In such cases, the static analysis tools should report everything it can; the objective is to leave no stone unturned.

False positives are less of a concern because a skilled analyst processes the results prior to the final report. With this model, the core group of tool users becomes skilled with the tools in short order and becomes adept at going through large numbers of results.

A Central Authority Sets Pinpoint Focus

Because of the large number of projects that might exist in an organization, a central distribution approach to results management can become constrained by the number of people reviewing results, even if reviewers are quite efficient. However, it is not unusual for a large fraction of the acute security pain to be clustered tightly around just a small number of types of issues. With this scenario, the project team will limit the tool to a small number of specific problem types, which can grow or change over time according to the risks the organization faces. Ultimately, defining a set of in-scope problem types works well as a centrally managed policy, standard, or set of guidelines. It should change only as fast as the development team can adapt and account for all the problems already in scope. On the whole, this approach gives people the opportunity to become experts incrementally through hands-on experience with the tool over time.

Start Small, Ratchet Up

Security tools tend to come preconfigured to detect as much as they possibly can. This is really good if you're trying to figure out what a tool is capable of detecting, but it can be overwhelming if you're assigned the task of going through every issue. No matter how you answer the adoption questions, our advice here is the same: Start small. Turn off most of the things the tool detects and concentrate on a narrow range of important and well-understood problems. Broaden out only when there's a process in place for using the tool and the initial batch of problems is under control. No matter what you do, a large body of existing code won't become perfect overnight. The people in your organization will thank you for helping them make some prioritization decisions.

3.3 Static Analysis Metrics

Metrics derived from static analysis results are useful for prioritizing remediation efforts, allocating resources among multiple projects, and getting feedback on the effectiveness of the security process. Ideally, one could use

metrics derived from static analysis results to help quantify the amount of risk associated with a piece of code, but using tools to measure risk is tricky. The most obvious problem is the unshakable presence of false positives and false negatives, but it is possible to compensate for them. By manually auditing enough results, a security team can predict the rate at which false positives and false negatives occur for a given project and extrapolate the number of true positives from a set of raw results. A deeper problem with using static analysis to quantify risk is that there is no good way to sum up the risk posed by a set of vulnerabilities. Are two buffer overflows twice as risky as a single buffer overflow? What about ten? Code-level vulnerabilities identified by tools simply do not sum into an accurate portrayal of risk. See the sidebar "The Density Deception" to understand why.

Instead of trying to use static analysis output to directly quantify risk, use it as a tactical way to focus security efforts and as an indirect measure of the process used to create the code.

The Density Deception

In the quality assurance realm, it's normal to compute the *defect density* for a piece of code by dividing the number of known bugs by the number of lines of code. Defect density is often used as a measure of quality. It might seem intuitive that one could use static analysis output to compute a "vulnerability density" to measure the amount of risk posed by the code. It doesn't work. We use two short example programs with some blatant vulnerabilities to explain why. First up is a straight-line program:

```
 1 /* This program computes Body Mass Index (BMI). */
 2 int main(int argc, char** argv)
 3 {
 4   char heightString[12];
 5   char weightString[12];
 6   int height, weight;
 7   float bmi;
 8
 9   printf("Enter your height in inches: ");
10   gets(heightString);
11   printf("Enter your weight in pounds: ");
12   gets(weightString);
13   height = atoi(heightString);
14   weight = atoi(weightString);
15   bmi = ((float)weight/((float)height*height)) * 703.0;
16
17   printf("\nBody mass index is %2.2f\n\n", bmi);
18 }
```

Continues

Continued

The program has 18 lines, and any static analysis tool will point out two glaring buffer overflow vulnerabilities: the calls to `gets()` on lines 10 and 12. Divide 2 by 18 for a vulnerability density of 0.111. Now consider another program that performs exactly the same computation:

```
 1 /* This program computes Body Mass Index (BMI). */
 2 int main(int argc, char** argv)
 3 {
 4   int height, weight;
 5   float bmi;
 6
 7   height = getNumber("Enter your height in inches");
 8   weight = getNumber("Enter your weight in pounds");
 9   bmi = ((float)weight/((float)height*height)) * 703.0;
10
11   printf("\nBody mass index is %2.2f\n\n", bmi);
12 }
13
14 int getNumber(char* prompt) {
15   char buf[12];
16   printf("%s: ", prompt);
17   return atoi(gets(buf));
18 }
```

This program calls `gets()`, too, but it uses a separate function to do it. The result is that a static analysis tool will report only one vulnerability (the call to `gets()` on line 17). Divide 1 by 18 for a vulnerability density of 0.056. Whoa. The second program is just as vulnerable as the first, but its vulnerability density is 50% smaller! The moral to the story is that the way the program is written has a big impact on the vulnerability density. This makes vulnerability density completely meaningless when it comes to quantifying risk. (Stay tuned. Even though vulnerability density is terrible in this context, the next section describes a legitimate use for it.)

Metrics for Tactical Focus

Many simple metrics can be derived from static analysis results. Here we look at the following:

- Measuring vulnerability density
- Comparing projects by severity
- Breaking down results by category
- Monitoring trends

Measuring Vulnerability Density

We've already thrown vulnerability density under the bus, so what more is there to talk about? Dividing the number of static analysis results by the number of lines of code is an awful way to measure risk, but it's a good way to measure the amount of work required to do a complete review. Comparing vulnerability density across different modules or different projects helps formulate review priorities. Track issue density over time to gain insight into whether tool output is being taken into consideration.

Comparing Projects by Severity

Static analysis results can be applied for project comparison purposes. Figure 3.3 shows a comparison between two modules, with the source code analysis results grouped by severity. The graph suggests a plan of action: Check out the critical issues for the first module, and then move on to the high-severity issues for the second.

Comparing projects side by side can help people understand how much work they have in front of them and how they compare to their peers. When you present project comparisons, name names. Point fingers. Sometimes programmers need a little help accepting responsibility for their code. Help them.

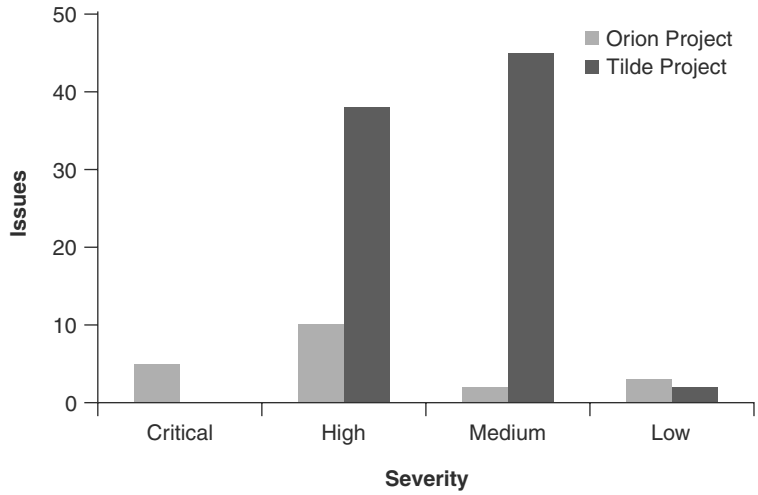

Figure 3.3 Source code analysis results broken down by severity for two subprojects.

Breaking Down Results by Category

Figure 3.4 presents results for a single project grouped by category. The pie chart gives a rough idea about the amount of remediation effort required to address each type of issue. It also suggests that log forging and cross-site scripting are good topics for an upcoming training class.

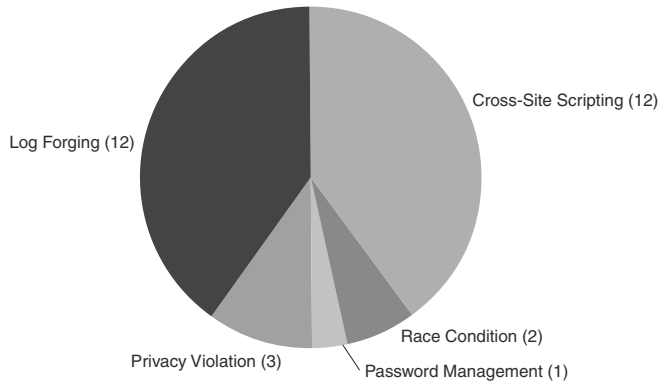

Figure 3.4 Source code analysis results broken down by category.

Source code analysis results can also point out trends over time. Teams that are focused on security will decrease the number of static analysis findings in their code. A sharp increase in the number of issues found deserves attention. Figure 3.5 shows the number of issues found during a series of nightly builds. For this particular project, the number of issues found on February 2 spikes because the development group has just taken over a module from a group that has not been focused on security.

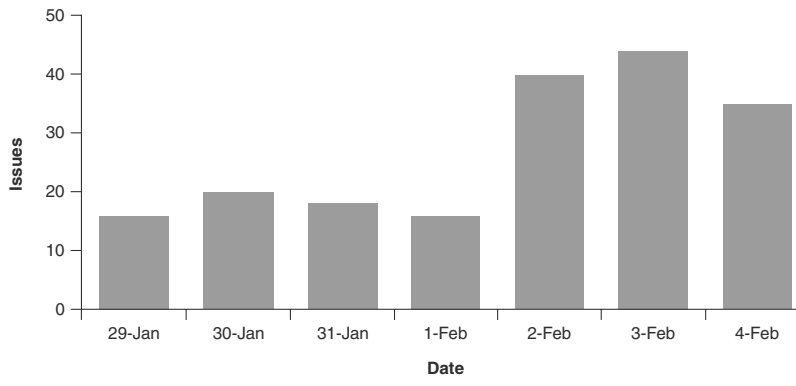

Figure 3.5 Source code analysis results from a series of nightly builds. The spike in issues on February 2 reflects the incorporation of a module originally written by a different team.

Process Metrics

The very presence of some types of issues can serve as an early indicator of more widespread security shortcomings [Epstein, 2006]. Determining the kinds of issues that serve as bellwether indicators requires some experience with the particular kind of software being examined. In our experience, a large number of string-related buffer overflow issues is a sign of trouble for programs written in C.

More sophisticated metrics leverage the capacity of the source code analyzer to give the same issue the same identifier across different builds. (See Chapter 4, "Static Analysis Internals," for more information on issue identifiers.) By following the same issue over time and associating it with the feedback provided by a human auditor, the source code analyzer can provide insight into the evolution of the project. For example, static analysis results can reveal the way a development team responds to security vulnerabilities. After an auditor identifies a vulnerability, how long, on average, does it take for the programmers to make a fix? We call this *vulnerability dwell*. Figure 3.6 shows a project in which the programmers fix critical vulnerabilities within two days and take progressively longer to address less severe problems.

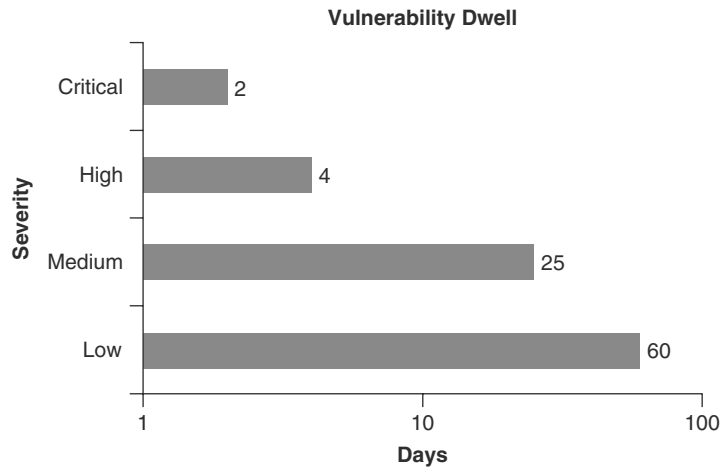

Figure 3.6 Vulnerability dwell as a function of severity. When a vulnerability is identified, vulnerability dwell measures how long it remains in the code. (The x-axis uses a log scale.)

Static analysis results can also help a security team decide when it's time to audit a piece of code. The rate of auditing should keep pace with the rate of development. Better yet, it should keep pace with the rate at which potential security issues are introduced into the code. By tracking individual issues over time, static analysis results can show a security team how many unreviewed issues a project contains. Figure 3.7 presents a typical graph. At the point the project is first reviewed, audit coverage goes to 100%. Then, as the code continues to evolve, the audit coverage decays until the project is audited again.

Another view of this same data gives a more comprehensive view of the project. An audit history shows the total number of results, number of results reviewed, and number of vulnerabilities identified in each build. This view takes into account not just the work of the code reviewers, but the effect the programmers have on the project. Figure 3.8 shows results over roughly one month of nightly builds. At the same time the code review is taking place, development is in full swing, so the issues in the code continue to change. As the auditors work, they report vulnerabilities (shown in black).

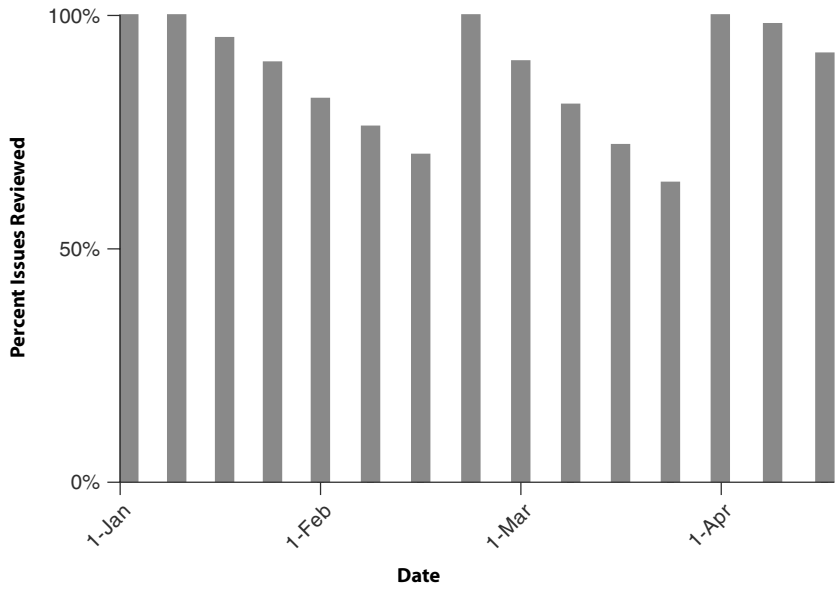

Figure 3.7 Audit coverage over time. After all static analysis results are reviewed, the code continues to evolve and the percentage of reviewed issues begins to decline.

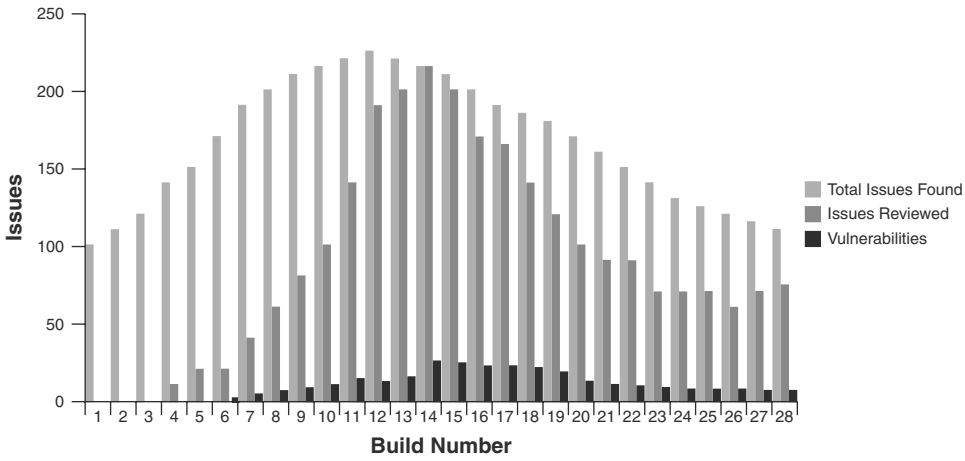

Figure 3.8 Audit history: the total number of static analysis results, the number of reviewed results, and the number of identified vulnerabilities present in the project.

Around build 14, the auditors have looked at all the results, so the total number of results is the same as the number reviewed. Development work is not yet complete, though, and soon the project again contains unreviewed results. As the programmers respond to some of the vulnerabilities identified by the audit team, the number of results begins to decrease and some of the identified vulnerabilities are fixed. At the far-right side of the graph, the growth in the number of reviewed results indicates that reviewers are beginning to look at the project again.

Summary

Building secure systems takes effort, especially for organizations that aren't used to paying much attention to security. Code review should be part of the software security process. When used as part of code review, static analysis tools can help codify best practices, catch common mistakes, and generally make the security process more efficient and consistent. But to achieve these benefits, an organization must have a well-defined code review process. At a high level, the process consists of four steps: defining goals, running tools, reviewing the code, and making fixes. One symptom of an ineffective process is a frequent descent into a debate about exploitability.

To incorporate static analysis into the existing development process, an organization needs a tool adoption plan. The plan should lay out who will run the tool, when they'll run it, and what will happen to the results. Static analysis tools are process agnostic, but the path to tool adoption is not. Take style and culture into account as you develop an adoption plan.

By tracking and measuring the security activities adopted in the development process, an organization can begin to sharpen its software security focus. The data produced by source code analysis tools can be useful for this purpose, giving insight into the kinds of problems present in the code, whether code review is taking place, and whether the results of the review are being acted upon in a timely fashion.