From Chess & West: Secure Programming with Static Analysis

2 Introduction to Static Analysis

The refinement of techniques for the prompt discovery of error serves as well as any other as a hallmark of what we mean by science.

—J. Robert Oppenheimer

This chapter is about static analysis tools: what they are, what they're good for, and what their limitations are. Any tool that analyzes code without executing it is performing static analysis. For the purpose of detecting security problems, the variety of static analysis tools we are most interested in are the ones that behave a bit like a spell checker; they prevent well-understood varieties of mistakes from going unnoticed. Even good spellers use a spell checker because, invariably, spelling mistakes creep in no matter how good a speller you are. A spell checker won't catch every slip-up: If you type *mute* when you mean *moot,* a spell checker won't help. Static analysis tools are the same way. A clean run doesn't guarantee that your code is perfect; it merely indicates that it is free of certain kinds of common problems. Most practiced and professional writers find a spell checker to be a useful tool. Poor writers benefit from using a spell checker too, but the tool does not transform them into excellent writers! The same goes for static analysis: Good programmers can leverage static analysis tools to excellent effect, but bad programmers will still produce bad programs regardless of the tools they use.

Our focus is on static analysis tools that identify security defects, but we begin by looking at the range of problems that static analysis can help solve. Later in the chapter, we look at the fundamental problems that make static analysis difficult from both a theoretical standpoint and a practical standpoint, and explore the trade-offs that tools make to meet their objectives.

2.1 Capabilities and Limitations of Static Analysis

Security problems can result from the same kind of simple mistakes that lead a good speller to occasionally make a typo: a little bit of confusion, a momentary lapse, or a temporary disconnect between the brain and the keyboard. But security problems can also grow out of a lack of understanding about what secure programming entails. It is not unusual for programmers to be completely unaware of some of the ways that attackers will try to take advantage of a piece of code.

With that in mind, static analysis is well suited to identifying security problems for a number of reasons:

- Static analysis tools apply checks thoroughly and consistently, without any of the bias that a programmer might have about which pieces of code are "interesting" from a security perspective or which pieces of code are easy to exercise through dynamic testing. Ever asked someone to proofread your work and had them point out an obvious problem that you completely overlooked? Was your brain automatically translating the words on the page into the words you intended to write? Then you know how valuable an unbiased analysis can be.
- By examining the code itself, static analysis tools can often point to the root cause of a security problem, not just one of its symptoms. This is particularly important for making sure that vulnerabilities are fixed properly. More than once, we've heard the story where the security team reports: "The program contains a buffer overflow. We know it contains a buffer overflow because when we feed it the letter *a* 50 times in a row, it crashes." Only later does the security team find out that the program has been fixed by checking to see if the input consists of exactly the letter *a* 50 times in a row.
- Static analysis can find errors early in development, even before the program is run for the first time. Finding an error early not only reduces the cost of fixing the error, but the quick feedback cycle can help guide a programmer's work: A programmer has the opportunity to correct mistakes he or she wasn't previously aware could even happen. The attack scenarios and information about code constructs used by a static analysis tool act as a means of knowledge transfer.

- When a security researcher discovers a new variety of attack, static analysis tools make it easy to recheck a large body of code to see where the new attack might succeed. Some security defects exist in software for years before they are discovered, which makes the ability to review legacy code for newly discovered types of defects invaluable.

The most common complaint leveled against static analysis tools that target security is that they produce too much noise. Specifically, they produce too many *false positives,* also known as *false alarms*. In this context, a false positive is a problem reported in a program when no problem actually exists. A large number of false positives can cause real difficulties. Not only does wading through a long list of false positives feel a little like serving latrine duty, but a programmer who has to look through a long list of false positives might overlook important results that are buried in the list.

False positives are certainly undesirable, but from a security perspective, *false negatives* are much worse. With a false negative, a problem exists in the program, but the tool does not report it. The penalty for a false positive is the amount of time wasted while reviewing the result. The penalty for a false negative is far greater. Not only do you pay the price associated with having a vulnerability in your code, but you live with a false sense of security stemming from the fact that the tool made it appear that everything was okay.

All static analysis tools are guaranteed to produce some false positives or some false negatives. Most produce both. We discuss the reasons why later in this chapter. The balance a tool strikes between false positives and false negatives is often indicative of the purpose of the tool. The right balance is quite different for static analysis tools that are meant to detect garden-variety bugs and static analysis tools that specifically target security-relevant defects. The cost of missing a garden-variety bug is, relatively speaking, small—multiple techniques and processes can be applied to make sure that the most important bugs are caught. For this reason, code quality tools usually attempt to produce a low number of false positives and are more willing to accept false negatives. Security is a different story. The penalty for overlooked security bugs is high, so security tools usually produce more false positives to minimize false negatives.

For a static analysis tool to catch a defect, the defect must be visible in the code. This might seem like an obvious point, but it is important to understand that architectural risk analysis is a necessary compliment to static analysis. Although some elements of a design have an explicit representation

in the program (a hard-coded protocol identifier, for example), in many cases, it is hard to derive the design given only the implementation.

2.2 Solving Problems with Static Analysis

Static analysis is used more widely than many people realize, partially because there are many kinds of static analysis tools, each with different goals. In this section, we take a look at some of the different categories of static analysis tools, referring to commercial vendors and open source projects where appropriate, and show where security tools fit in. We cover:

- Type checking
- Style checking
- Program understanding
- Program verification
- Property checking
- Bug finding
- Security review

Type Checking

The most widely used form of static analysis, and the one that most programmers are familiar with, is type checking. Many programmers don't give type checking much thought. After all, the rules of the game are typically defined by the programming language and enforced by the compiler, so a programmer gets little say in when the analysis is performed or how the analysis works. Type checking is static analysis nonetheless. Type checking eliminates entire categories of programming mistakes. For example, it prevents programmers from accidentally assigning integral values to object variables. By catching errors at compile time, type checking prevents runtime errors.

Type checking is limited in its capacity to catch errors, though, and it suffers from false positives and false negatives just like all other static analysis techniques. Interestingly, programmers rarely complain about a type checker's imperfections. The Java statements in Example 2.1 will not compile because it is never legal to assign an expression of type `int` to a variable of type `short`, even though the programmer's intent is unambiguous. Example 2.2 shows the output from the Java compiler. This is an

example of a type-checking false positive. The problem can be fixed by introducing an explicit type cast, which is the programmer's way of overriding the default type inference behavior.

Example 2.1 A type-checking false positive: These Java statements do not meet type safety rules even though they are logically correct.

```
10  short s = 0;
11  int i = s;   /* the type checker allows this */
12  short r = i; /* false positive: this will cause a
13                  type checking error at compile time */
```

Example 2.2 Output from the Java compiler demonstrating the type-checking false positive.

```
$ javac bar.java
bar.java:12: possible loss of precision
found   : int
required: short
  short r = i; /* false positive: this will cause a
             ^
1 error
```

Type checking suffers from false negatives, too. The Java statements in Example 2.3 will pass type checking and compile without a hitch, but will fail at runtime. Arrays in Java are covariant, meaning that the type checker allows an `Object` array variable to hold a reference to a `String` array (because the `String` class is derived from the `Object` class), but at runtime Java will not allow the `String` array to hold a reference to an object of type `Object`. The type checker doesn't complain about the code in Example 2.3, but when the code runs, it throws an `ArrayStoreException`. This represents a type-checking false negative.

Example 2.3 These Java statements meet type-checking rules but will fail at runtime.

```
Object[] objs = new String[1];
objs[0] = new Object();
```

Style Checking

Style checkers are also static analysis tools. They generally enforce a pickier and more superficial set of rules than a type checker. Pure style checkers enforce rules related to whitespace, naming, deprecated functions, commenting, program structure, and the like. Because many programmers are fiercely attached to their own version of good style, most style checkers are quite flexible about the set of rules they enforce. The errors produced by style checkers often affect the readability and the maintainability of the code but do not indicate that a particular error will occur when the program runs.

Over time, some compilers have implemented optional style checks. For example, gcc's `-Wall` flag will cause the compiler to detect when a switch statement does not account for all possible values of an enumerated type. Example 2.4 shows a C function with a suspicious switch statement. Example 2.5 shows what gcc says about the function when `-Wall` is in effect.

Example 2.4 A C function with a switch statement that does not account for all possible values of an enumerated type.

```
 1 typedef enum { red, green, blue } Color;
 2
 3 char* getColorString(Color c) {
 4  char* ret = NULL;
 5  switch (c) {
 6    case red:
 7      printf("red");
 8  }
 9  return ret;
10 }
```

Example 2.5 The output from gcc using the `-Wall` flag.

```
enum.c:5: warning: enumeration value 'green' not handled in switch
enum.c:5: warning: enumeration value 'blue' not handled in switch
```

It can be difficult to adopt a style checker midway through a large programming project because different programmers likely have been adhering to somewhat different notions of the "correct" style. After a project has

begun, revisiting code purely to make adjustments to reduce output from the style checker will realize only marginal benefit and at the cost of great inconvenience. Going through a large body of code and correcting style-checker warnings is a little like painting a wooden house that's infested by termites. Style checking is easiest to adopt at the outset of a project.

Many open source and commercial style checkers are available. By far the most famous is the venerable tool *lint*. Many of the checks originally performed by lint have been incorporated into the various warning levels offered by popular compilers, but the phrase *lint-like* has stuck around as a pejorative term for describing style checkers. For style checking Java, we like the open source program PMD (http://pmd.sourceforge.net) because it makes it easy to choose the style rules you'd like to enforce and almost as easy to add your own rules. PMD also offers some rudimentary bug detection capability. Parasoft (http://www.parasoft.com) sells a combination bug finder/style checker for Java, C, and C++.

Program Understanding

Program understanding tools help users make sense of a large codebase. Integrated development environments (IDEs) always include at least some program understanding functionality. Simple examples include "find all uses of this method" and "find the declaration of this global variable." More advanced analysis can support automatic program-refactoring features, such as renaming variables or splitting a single function into multiple functions.

Higher-level program understanding tools try to help programmers gain insight into the way a program works. Some try to reverse-engineer information about the design of the program based on an analysis of the implementation, thereby giving the programmer a big-picture view of the program. This is particularly useful for programmers who need to make sense out of a large body of code that they did not write, but it is a poor substitute for the original design itself.

The open source Fujaba tool suite (http://wwwcs.uni-paderborn.de/cs/fujaba/), pictured in Figure 2.1, enables a developer to move back and forth between UML diagrams and Java source code. In some cases, Fujaba can also infer design patterns from the source code it reads.

CAST Systems (http://www.castsoftware.com) focuses on cataloging and exploring large software systems.

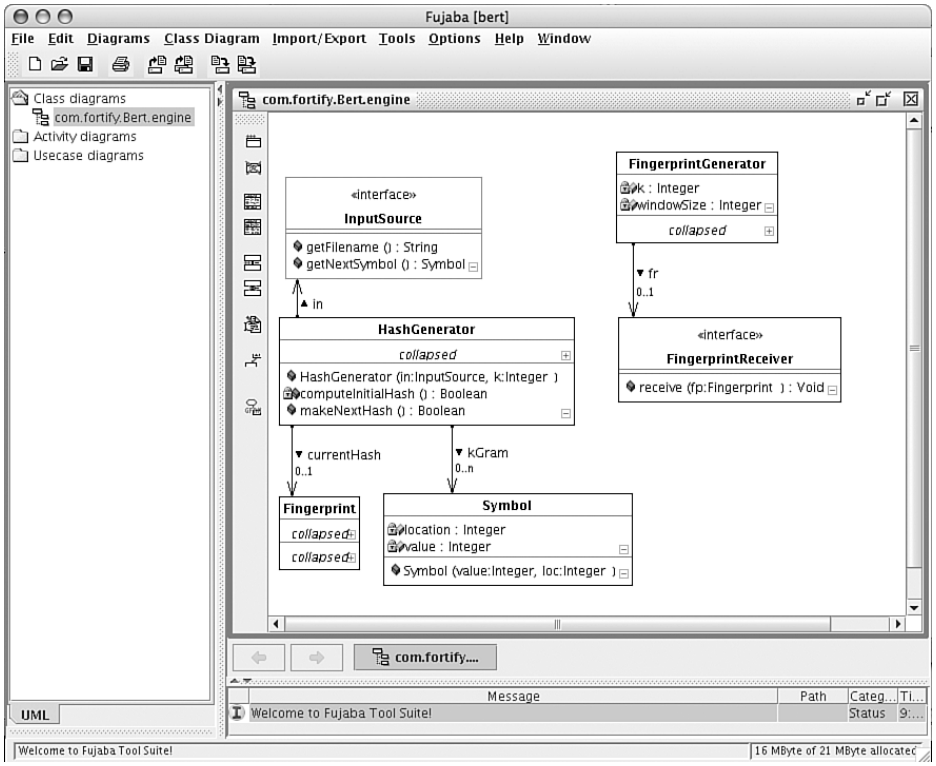

Figure 2.1 Fujaba enables programmers to move back and forth between a UML view and source code. Fujaba has a reverse-engineering capability that allows it to read source code and identify a limited set of design patterns.

Program Verification and Property Checking

A program verification tool accepts a specification and a body of code and then attempts to prove that the code is a faithful implementation of the specification. If the specification is a complete description of everything the program should do, the program verification tool can perform *equivalence checking* to make sure that the code and the specification exactly match.[1]

1. Equivalence checking is not used much for software, but in the world of hardware design, where a circuit might go through a long series of complex transformations on its way to becoming a piece of silicon, equivalence checking is widely used to make sure that a transformed design remains true to the original design.

Rarely do programmers have a specification that is detailed enough that it can be used for equivalence checking, and the job of creating such a specification can end up being more work than writing the code, so this style of formal verification does not happen very often. Even more limiting is the fact that, historically, equivalence checking tools have not been able to process programs of any significant size. See the sidebar "Formal Verification and the Orange Book" for a 1980s attempt at pulling formal verification toward the mainstream.

More commonly, verification tools check software against a *partial specification* that details only part of the behavior of a program. This endeavor sometimes goes by the name *property checking*. The majority of property checking tools work either by applying logical inference or by performing model checking. (We discuss analysis algorithms in more detail in Chapter 4.)

Many property checking tools focus on *temporal safety properties*. A temporal safety property specifies an ordered sequence of events that a program must *not* carry out. An example of a temporal safety property is "a memory location should not be read after it is freed." Most property-checking tools enable programmers to write their own specifications to check program-specific properties.

When a property checking tool discovers that the code might not match the specification, it traditionally explains its finding to the user by reporting a *counterexample:* a hypothetical event or sequence of events that takes place within the program that will lead to the property being violated. Example 2.6 gives a few C statements that contain a memory leak, and Example 2.7 shows how a property checking tool might go about reporting a violation of the property using a counterexample to tell the story of the leaking memory.

Example 2.6 A memory leak. If the first call to `malloc()` succeeds and the second fails, the memory allocated in the first call is leaked.

```
1  inBuf = (char*) malloc(bufSz);
2  if (inBuf == NULL)
3    return -1;
4  outBuf = (char*) malloc(bufSz);
5  if (outBuf == NULL)
6    return -1;
```

Example 2.7 A counterexample from a property checking tool running against the code in Example 2.6. The sequence of events describes a way in which the program can violate the property "allocated memory should always be freed".

```
Violation of property "allocated memory should always be freed":
  line 2: inBuf != NULL
  line 5: outBuf == NULL
  line 6: function returns (-1) without freeing inBuf
```

A property checking tool is said to be *sound with respect to the specification* if it will always report a problem if one exists. In other words, the tool will never suffer a false negative. Most tools that claim to be sound require that the program being evaluated meet certain conditions. Some disallow function pointers, while others disallow recursion or assume that two pointers never alias (point to the same memory location). Soundness is an important characteristic in an academic context, where anything less might garner the label "unprincipled." But for large real-world bodies of code, it is almost impossible to meet the conditions stipulated by the tool, so the soundness guarantee is not meaningful. For this reason, soundness is rarely a requirement from a practitioner's point of view.

In striving for soundness or because of other complications, a property checking tool might produce false positives. In the case of a false positive, the counterexample will contain one or more events that could not actually take place. Example 2.8 gives a second counterexample for a memory leak. This time, the property checker has gone wrong; it does not understand that, by returning `NULL`, `malloc()` is indicating that no memory has been allocated. This could indicate a problem with the way the property is specified, or it could be a problem with the way the property checker works.

Example 2.8 An errant counterexample from a property checking tool running against the code in Example 2.6. The tool does not understand that when `malloc()` returns `NULL`, no memory has been allocated, and therefore no memory needs to be freed.

```
Violation of property "allocated memory should always be freed":
  line 2: inBuf == NULL
  line 3: function returns (-1) without freeing inBuf
```

Praxis High Integrity Systems (http://www.praxis-his.com) offers a commercial program verification tool for a subset of the Ada programming

language [Barnes, 2003]. Escher Technologies (http://www.eschertech.com) has its own programming language that can be compiled into C++ or Java. Numerous university research projects exist in both the program verification and property checking realm; we discuss many of them in Chapter 4. Polyspace (http://www.polyspace.com) and Grammatech (http://www.gramatech.com) both sell property checking tools.

Formal Verification and the Orange Book

Formal verification, wherein a tool applies a rigorous mathematical approach to its verification task, has a long and storied history. One of the best-known calls for the application of formal methods for the purposes of verifying security properties of system designs was included as part of the Trusted Computer System Evaluation Criteria (TCSEC), more often known by its colloquial name "the Orange Book" [DOD, 1985]. The Orange Book was written to guide developers in the creation of secure systems for sale to the U.S. government and military. The TCSEC is no longer in use, but many of the concepts it contained formed the basis for the Common Criteria (ISO/IEC standard 15408), a system for specifying and measuring security requirements. The Common Criteria are primarily used by government and military agencies in the United States and Europe.

The Orange Book outlines a hierarchy of security features and assurances along with a qualification process for certifying a product at a particular ranking. The TCSEC covers a wide variety of subjects, including mechanisms that should be used to protect information in the system (access controls), identification and authentication of users, audit features, system specification, architecture, test and verification methods, covert channel analysis, documentation requirements, trusted product-delivery systems, and many others.

The TCSEC does not mandate the use of formal methods for any level of certification except the highest one: A1. A1 certification requires a formal demonstration that the system design meets the requirements of the security policy. Formally demonstrating that the design has been implemented without error was not required.

A1 certification entailed rigorously defining a system's security policy and formally demonstrating that the system design enforces the policy. By the few who attempted it, this was always achieved by hierarchically decomposing the design, showing that the highest level of abstraction meets the requirements of the security policy and that each lower level of abstraction meets the requirements specified by the next higher level.

Bug Finding

The purpose of a bug finding tool is not to complain about formatting issues, like a style checker, nor is it to perform a complete and exhaustive comparison of the program against a specification, as a program verification tool would. Instead, a bug finder simply points out places where the program will behave in a way that the programmer did not intend. Most bug finders are easy to use because they come prestocked with a set of "bug idioms" (rules) that describe patterns in code that often indicate bugs.

Example 2.9 demonstrates one such idiom, known as double-checked locking. The purpose of the code is to allocate at most one object while minimizing the number of times any thread needs to enter the `synchronized` block. Although it might look good, before Java 1.5, it does not work—earlier Java versions did not guarantee that only one object would be allocated [Bacon, 2007]. Example 2.10 shows how the open source tool FindBugs (http://www.findbugs.org) identifies the problem [Hovemeyer and Pugh, 2004].

Example 2.9 Double-checked locking. The purpose is to minimize synchronization while guaranteeing that only one object will ever be allocated, but the idiom does not work.

```
1 if (this.fitz == null) {
2   synchronized (this) {
3     if (this.fitz == null) {
4       this.fitz = new Fitzer();
5     }
6   }
7 }
```

Example 2.10 FindBugs identifies the double-checked locking idiom.

```
M M DC: Possible doublecheck on Fizz.fitz in Fizz.getFitz()
        At Fizz.java:[lines 1-3]
```

Sophisticated bug finders can extend their built-in patterns by inferring requirements from the code itself. For example, if a Java program uses the same synchronization lock to restrict access to a particular member variable in 99 out of 100 places where the member variable is used, it is likely that the lock should also protect the 100th usage of the member variable.

Some bug finding tools use the same sorts of algorithms used by property checking tools, but bug finding tools generally focus on producing a low number of false positives even if that means a higher number of false negatives. An ideal bug finding tool is *sound with respect to a counterexample*. In other words, when it generates a bug report, the accompanying counterexample always represents a feasible sequence of events in the program. (Tools that are sound with respect to a counterexample are sometimes called *complete* in academic circles.)

We think FindBugs does an excellent job of identifying bugs in Java code. Coverity makes a bug finder for C and C++ (http://www.coverity.com). Microsoft's Visual Studio 2005 includes the `\analyze` option (sometimes called Prefast) that checks for common coding errors in C and C++. Klocwork (http://www.klocwork.com) offers a combination program understanding and bug finding static analysis tool that enables graphical exploration of large programs.

Security Review

Security-focused static analysis tools use many of the same techniques found in other tools, but their more focused goal (identifying security problems) means that they apply these techniques differently.

The earliest security tools, ITS4 [Viega et al., 2000], RATS [RATS, 2001], and Flawfinder [Wheeler, 2001], were little more than a glorified grep; for the most part, they scanned code looking for calls to functions such as `strcpy()` that are easy to misuse and should be inspected as part of a manual source code review. In this sense, they were perhaps most closely related to style checkers—the things they pointed out would not necessarily cause security problems, but they were indicative of a heightened reason for security concern. Time after time, these tools have been indicted for having a high rate of false positives because people tried to interpret the tool output as a list of bugs rather than as an aid for use during code review.

Modern security tools are more often a hybrid of property checkers and bug finders. Many security properties can be succinctly expressed as program properties. For a property checker, searching for potential buffer overflow vulnerabilities could be a matter of checking the property "the program does not access an address outside the bounds of allocated memory". From the bug finding domain, security tools adopt the notion that developers often continue to reinvent the same insecure method of solving a problem, which can be described as an insecure programming idiom.

As we noted earlier, even though bug finding techniques sometimes prove useful, security tools generally cannot inherit the bug finding tools' tendency to minimize false positives at the expense of allowing false negatives. Security tools tend to err on the side of caution and point out bits of code that should be subject to manual review even if the tool cannot prove that they represent exploitable vulnerabilities. This means that the output from a security tool still requires human review and is best applied as part of a code review process. (We discuss the process for applying security tools in Chapter 3, "Static Analysis as Part of Code Review.") Even so, the better a security tool is, the better job it will do at minimizing "dumb" false positives without allowing false negatives to creep in.

Example 2.11 illustrates this point with two calls to `strcpy()`. Using `strcpy()` is simply not a good idea, but the first call cannot result in a buffer overflow; the second call will result in an overflow if `argv[0]` points to a very long string. (The value of `argv[0]` is usually, but not always, the name of the program.[2]) A good security tool places much more emphasis on the second call because it represents more than just a bad practice; it is potentially exploitable.

Example 2.11 A C program with two calls to `strcpy()`. A good security tool will categorize the first call as safe (though perhaps undesirable) and the second call as dangerous.

```
int main(int argc, char* argv[]) {
  char buf1[1024];
  char buf2[1024];
  char* shortString = "a short string";
  strcpy(buf1, shortString); /* safe use of strcpy */
  strcpy(buf2, argv[0]);     /* dangerous use of strcpy */
  ...
```

Fortify Software (http://www.fortify.com) and Ounce Labs (http://www.ouncelabs.com) make static analysis tools that specifically

2. Under filesystems that support symbolic links, an attacker can make a symbolic link to a program, and then `argv[0]` will be the name of the symbolic link. In POSIX environments, an attacker can write a wrapper program that invokes a function such as `execl()` that allows the attacker to specify a value for `argv[0]` that is completely unrelated to the name of the program being invoked. Both of these scenarios are potential means of attacking a privileged program. See Chapter 12 for more about attacks such as these.

target security. Both of us are particularly fond of Fortify because we've put a lot of time and effort into building Fortify's static analysis tool set. (Brian is one of Fortify's founders, and Jacob manages Fortify's Security Research Group.) A third company, Secure Software, sold a static analysis tool aimed at security, but in early 2007, Fortify acquired Secure's intellectual property. Go Fortify!

2.3 A Little Theory, a Little Reality

Static analysis is a computationally undecidable problem. Naysayers sometimes try to use this fact to argue that static analysis tools are not useful, but such arguments are specious. To understand why, we briefly discuss undecidability. After that, we move on to look at the more practical issues that make or break a static analysis tool.

In the mid-1930s, Alan Turing, as part of his conception of a general-purpose computing machine, showed that algorithms cannot be used to solve all problems. In particular, Turing posed the *halting problem,* the problem of determining whether a given algorithm terminates (reaches a final state). The proof that the halting problem is undecidable boils down to the fact that the only way to know for sure what an algorithm will do is to carry it out. In other words, the only guaranteed way to know what a program will do is to run it. If indeed an algorithm does not terminate, a decision about whether the algorithm terminates will never be reached. This notion of using one algorithm to analyze another (the essence of static analysis) is part of the foundation of computer science. For further reading on computational theory, we recommend Sipser's *Introduction to the Theory of Computation, Second Edition* [Sipser, 2005].

In 1953, Henry Rice posed what has come to be known as *Rice's theorem.* The implication of Rice's theorem is that static analysis cannot perfectly determine any nontrivial property of a general program. Consider the following two lines of pseudocode:

```
if program p halts
  call unsafe()
```

It is easy to see that, to determine whether this code ever calls the function `unsafe()`, a static analysis tool must solve the halting problem.

Example 2.12 gives a deeper demonstration of the fundamental difficulties that force all static analysis tools to produce at least some false posi-

tives or false negatives. Imagine the existence of a function `is_safe()` that always returns `true` or `false`. It takes a program as an argument, and returns `true` if the program is safe and `false` if the program is not safe. This is exactly the behavior we'd like from a static analysis tool. We can informally show that `is_safe()` cannot possibly fulfill its promise.

Example 2.12 shows the function `bother()` that takes a function as its argument and calls `unsafe()` only if its argument is safe. The example goes on to call the function `bother()` on itself recursively. Assume that `is_safe()` itself is safe. What should the outcome be? If `is_safe()` declares that `bother()` is safe, `bother` will call `unsafe()`. Oops. If `is_safe()` declares that `bother()` is unsafe, `bother()` will not call `unsafe()`, and, therefore, it is safe. Both cases lead to a contradiction, so `is_safe()` cannot possibly behave as advertised.

Example 2.12 Perfectly determining any nontrivial program property is impossible in the general case. `is_safe()` cannot behave as specified when the function `bother()` is called on itself.

```
bother(function f) {
  if ( is_safe(f) )
    call unsafe();
}

b = bother;
bother(b);
```

Success Criteria

In practice, the important thing is that static analysis tools provide useful results. The fact that they are imperfect does not prevent them from having significant value. In fact, the undecidable nature of static analysis is not really the major limiting factor for static analysis tools. To draw an analogy from the physical world, the speed of light places a potential limitation on the maximum speed of a new car, but many engineering difficulties limit the speed of cars well before the speed of light becomes an issue. The major practical factors that determine the utility of a static analysis tool are:

- The ability of the tool to make sense of the program being analyzed
- The trade-offs the tool makes between precision and scalability

- The set of errors that the tool checks for
- The lengths to which the tool's creators go to in order to make the tool easy to use

Making Sense of the Program

Ascribing meaning to a piece of source code is a challenging proposition. It requires making sense of the program text, understanding the libraries that the program relies on, and knowing how the various components of the program fit together.

Different compilers (or even different versions of the same compiler) interpret source code in different ways, especially where the language specification is ambiguous or allows the compiler leeway in its interpretation of the code. A static analysis tool has to know the rules the compiler plays by to parse the code the same way the compiler does. Each corner case in the language represents another little problem for a static analysis tool. Individually, these little problems are not too hard to solve, but taken together, they make language parsing a tough job. To make matters worse, some large organizations create their own language dialects by introducing new syntax into a language. This compounds the parsing problem.

After the code is parsed, a static analysis tool must understand the effects of library or system calls invoked by the program being analyzed. This requires the tool to include a model for the behavior of library and system functions. Characterizing all the relevant libraries for any widely used programming language involves understanding thousands of functions and methods. The quality of the tool's program model—and therefore the quality of its analysis results—is directly related to the quality of its library characterizations.

Although most static analysis research has focused on analyzing a single program at a time, real software systems almost always consist of multiple cooperating programs or modules, which are frequently written in different programming languages. If a static analysis tool can analyze multiple languages simultaneously and make sense of the relationships between the different modules, it can create a system model that more accurately represents how, when, and under what conditions the different pieces of code will run.

Finally, modern software systems are increasingly driven by a critical aspect of their environment: configuration files. The better a tool can make sense of a program's configuration information, the better the model

it can create. Popular buzzwords, including *system-oriented architecture, aspect-oriented programming,* and *dependency injection,* all require understanding configuration information to accurately model the behavior of the program.

Configuration information is useful for another purpose, too. For Web-based applications, the program's configuration often specifies the binding between the code and the URIs used to access the code. If a static analysis tool understands this binding, its output can include information about which URIs and which input parameters are associated with each vulnerability. In some cases, a dynamic testing tool can use this information to create an HTTP request to the Web application that will demonstrate the vulnerability. Working in the other direction, when a dynamic testing tool finds a vulnerability, it can use static analysis results to provide a root-cause analysis of the vulnerability.

Not all static analysis results are easy to generate tests for, however. Some depend on very precise timing, and others require manipulation of input sources other than the HTTP request. Just because it is hard to generate a dynamic test for a static analysis result does not mean the result is invalid. Conversely, if it is easy to generate a dynamic test for a static analysis result, it is reasonable to assume that it would be easy for an attacker to generate the same test.

Trade-Offs Between Precision, Depth, and Scalability

The most precise methods of static analysis, in which all possible values and all eventualities are tracked with unyielding accuracy, are currently capable of analyzing thousands or tens of thousands of lines of code before the amount of memory used and the execution time required become unworkable. Modern software systems often involve millions or tens of millions of lines of code, so maximum precision is not a realistic possibility in many circumstances. On the other end of the spectrum, a simple static analysis algorithm, such as one that identifies the use of dangerous or deprecated functions, is capable of processing an effectively unlimited amount of code, but the results provide only limited value. Most static analysis tools sacrifice some amount of precision to achieve better scalability. Cutting-edge research projects often focus on finding better trade-offs. They look for ways to gain scalability by sacrificing precision in such a way that it will not be missed.

The depth of analysis a tool performs is often directly proportional to the scope of the analysis (the amount of the program that the tool examines

at one time). Looking at each line one at a time makes for fast processing, but the lack of context necessarily means that the analysis will be superficial. At the other extreme, analyzing an entire program or an entire system provides much better context for the analysis but is expensive in terms of time, memory, or both. In between are tools that look at individual functions or modules one at a time.

From a user's perspective, static analysis tools come in several speed grades. The fastest tools provide almost instantaneous feedback. These tools could be built into an IDE the same way an interactive spell checker is built into Microsoft Word, or they could run every time the compiler runs. With the next rung up, users might be willing to take a coffee break or get lunch while the tool runs. A programmer might use such a tool once a day or just before committing code to the source repository. At the top end, tools give up any pretense at being interactive and run overnight or over a weekend. Such tools are best suited to run as part of a nightly build or a milestone build. Naturally, the greater the depth of the analysis, the greater the runtime of the tool.

To give a rough sense of the trade-offs that tools make, Figure 2.2 considers the bug finding and security tools discussed earlier in the chapter and plots their execution time versus the scope of the analysis they perform.

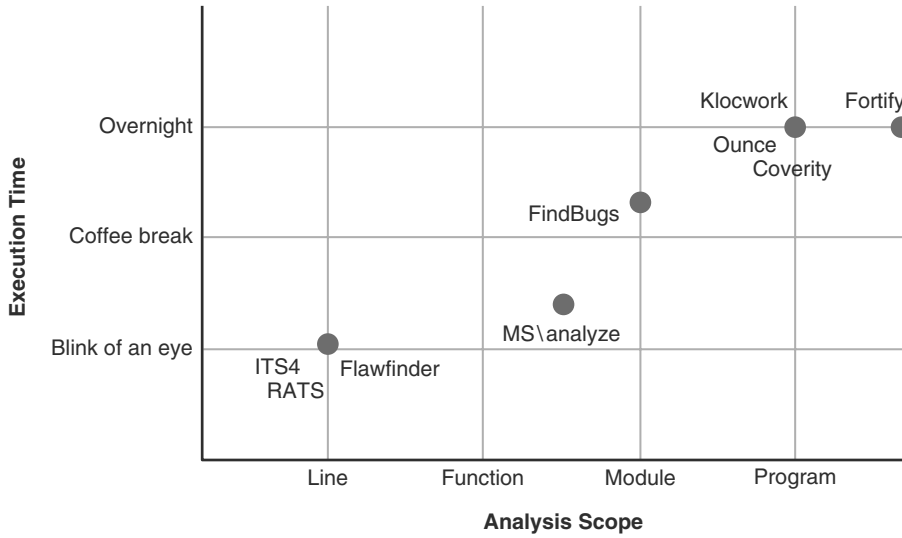

Figure 2.2 Analysis scope vs. execution time for the bug finding and security tools discussed in Section 2.1.

Finding the Right Stuff

Static analysis tools must be armed with the right set of defects to search for. What the "right set" consists of depends entirely upon the purpose of the software being analyzed. Clients fail differently than servers. Operating systems fail differently than desktop applications. The makers of a static analysis tool must somehow take this context into account. With research tools, the most common approach is to build a tool that targets only a small number of scenarios. Commercial tools sometimes ask the user to select the scenario at hand to make decisions about what to report.

Even with a limited purview, the most valuable things to search for are often specific to the particular piece of software being evaluated. Finding these defects requires the tool to be extensible; users must be able to add their own *custom rules*. For example, detecting locations where private data are made public or otherwise mismanaged by a program requires adding custom rules that tell the analysis tool which pieces of data are considered private.

Just as a good program model requires a thorough characterization of the behavior of libraries and system interfaces, detecting defects requires a thorough set of rules that define where and under what circumstances the defects can occur. The size of the rule set is the first and most obvious means of comparing the capabilities of static analysis tools [McGraw, 2006], but counting the number of rules that a tool has does not tell the whole story, especially if a single rule can be applied in a variety of circumstances or can contain wildcards that match against entire families of functions. Comparing static analysis tools based on the size of their rule sets is like comparing operating systems based on the number of lines of source code they are built from.

The best way to compare static analysis tools is by using them to analyze the same code and comparing the results, but choosing the right code for comparing tools is no small problem in and of itself. A number of attempts at creating static analysis benchmarks have arisen in the last few years:

- Benjamin Livshits has put together two benchmarks for static analysis tools. SecuriBench (http://suif.stanford.edu/~livshits/securibench/) is a collection of open source Web-based Java programs that contain known security defects. SecuriBench Micro (http://suif.stanford.edu/~livshits/work/securibench-micro/) is a set of small hand-crafted Java programs that are intentionally written to stress different aspects of a static analysis tool.
- Zitser, Lippman, and Leek have assembled a small collection of vulnerable programs derived from real-world vulnerable programs for the

purpose of testing static analysis tools [Zitser, 2004]. Kratkiewicz later created a set of scripts to generate vulnerable programs with different characteristics [Kratkiewicz, 2005].

- The SAMATE group at NIST (http://samate.nist.gov) is in the process of creating a publicly available reference data set for the purpose of benchmarking static analysis tools.
- Tim Newsham and Brian Chess have developed the Analyzer Benchmark (ABM), which consists of a mix of small hand-crafted programs and large real-world programs meant for characterizing static analysis tools [Newsham and Chess, 2005]. All of the ABM test cases have been donated to the NIST SAMATE project.
- The Department of Homeland Security's Build Security In site (https://buildsecurityin.us-cert.gov) hosts a set of sample programs developed by Cigital that are meant to help evaluate static analysis tools.

Beyond selecting test cases, benchmarking static analysis tools is difficult because there is no widely agreed-upon yardstick for comparing results. It's hard to reach a general consensus about whether one possible trade-off between false positives and false negatives is better than another. If you need to perform a tool evaluation, our best advice is to run all the tools against a real body of code that you understand well. If possible, use a program that contains known vulnerabilities. Compare results in light of your particular needs.

Ease of Use

A static analysis tool is always the bearer of bad news. It must convince a programmer that the code does something unexpected or incorrect. If the way the tool presents its findings is not clear and convincing, the programmer is not likely to take heed.

Static analysis tools have greater usability problems than that, though. If a tool does not present good error information when it is invoked incorrectly or when it cannot make sense of the code, users will not understand the limitations of the results they receive. In general, finding the source code that needs to be analyzed is a hard job because not all source files are relevant under all circumstances, and languages such as C and C++ allow code to be included or not included using preprocessor directives. The best way for a tool to identify the code that actually needs to be analyzed is for it to integrate smoothly with the program's build system. Popular build tools include Make, Ant, Maven, and a whole manner of integrated development environments such as Microsoft's Visual Studio and the open source

program Eclipse. Integrating within a programmer's development environment also provides a forum for presenting results.

In an industrial setting, a source code analysis tool must fit in as part of the software development process. (This is the topic of Chapter 3.) Because the same codebase can grow and evolve over a period of months, years or decades, the tool should make it easy to review results through multiple revisions of the code. It should allow users to suppress false positives so that they don't have to review them again in the future or to look at only issues that have been introduced since the last review.

All static analysis tools make trade-offs between false positives and false negatives. Better tools allow the user to control the trade-offs they make, to meet the specific needs of the user.

Analyzing the Source vs. Analyzing Compiled Code

Most static analysis tools examine a program as the compiler sees it (by looking at the source code), but some examine the program as the runtime environment sees it (by looking at the bytecode or the executable). Looking at compiled code offers two advantages:

- The tool does not need to guess at how the compiler will interpret the code because the compiler has already done its work. Removing the compiler removes ambiguity.
- Source code can be hard to come by. In some circumstances, it is easier to analyze the bytecode or the executable simply because it is more readily available.

A few distinct disadvantages exist, too:

- Making sense out of compiled code can be tough. A native executable can be difficult to decode. This is particularly true for formats that allow variable-width instructions, such as Intel x86, because the meaning of program changes depending upon where decoding begins. Some static analysis tools use information gleaned from dynamic analysis tools such as IDA Pro to counter this problem. Even a properly decoded binary lacks the type information that is present in source code. The lack of type information makes analysis harder. Optimizations performed by the compiler complicate matters further.

  Languages such as Java that are compiled into bytecode do not have the same decoding problem, and type information is present in the bytecode, too. But even without these problems, the transformations performed by the compiler can throw away or obscure information about

the programmer's intent. The compilation process for JavaServer Pages (JSPs) illustrates this point. The JavaServer Pages format allows a programmer to combine an HTML-like markup language with Java code. At runtime, the JSP interpreter compiles the JSP source file into Java source code and then uses the standard Java compiler to translate the Java source code into bytecode.

The three lines of JSP markup shown in Example 2.13 are relatively straightforward: They echo the value of a URL parameter. (This is a cross-site scripting vulnerability. For more information about cross-site scripting, see Chapter 9, "Web Applications.") The JSP compiler translates these three lines of markup into more than 50 lines of Java source code, as shown in Example 2.14. The Java source code contains multiple conditionals, a loop, and several return statements, even though none of these constructs is evident in the original JSP markup. Although it is possible to understand the behavior of the JSP by analyzing the Java source, it is significantly more difficult. Taken together, these examples demonstrate the kinds of challenges that looking at compiled code can introduce.

This problem is even worse for C and C++ programs. For many kinds of program properties, analyzing the implementation of a function does not reveal the semantics of the function. Consider a function that allows the program to send a string as a SQL query to a remote database. The executable code might reveal some transformations of a string, then some packets sent out over the network, and then some packets received back from the network. These operations would not explain that the string will be interpreted as a SQL query.

- Analyzing a binary makes it harder to do a good job of reporting useful findings. Most programmers would like to see findings written in terms of source code, and that requires a binary analysis tool to map its analysis from the executable back to the source. If the binary includes debugging information, this mapping might not be too difficult, but if the binary does not have debugging information or if the compiler has optimized the code to the point that it does not easily map back to the source, it will be hard to make sense of the analysis results.

For bytecode formats such as Java, there is no clear right answer. The source code contains more information, but the bytecode is easier to come by. For native programs, the disadvantages easily outweigh the advantages; analyzing the source is easier and more effective.

Example 2.13 A small example of Java Server Page (JSP) markup. The code echoes the value of a URL parameter (a cross-site scripting vulnerability).

```
<fmt:message key="hello">
    <fmt:param value="${param.test}"/>
</fmt:message>
```

Example 2.14 Three lines of JSP markup are transformed into more than 50 lines of Java code. The Java code is much harder to understand than the JSP markup. Translating JSP into Java before analyzing it makes the analysis job much harder.

```
if (_fmt_message0 == null)
  _fmt_message0 =
    new org.apache.taglibs.standard.tag.el.fmt.
        MessageTag_fmt_message0.setPageContext(pageContext);
_fmt_message0.setParent((javax.servlet.jsp.tagext.Tag)null);
_activeTag = _fmt_message0;
_fmt_message0.setKey(
    weblogic.utils.StringUtils.valueOf("hello"));
_int0 = _fmt_message0.doStartTag();
if (_int0 != Tag.SKIP_BODY) {
  if (_int0 == BodyTag.EVAL_BODY_BUFFERED) {
    out = pageContext.pushBody();
    _fmt_message0.setBodyContent((BodyContent)out);
    _fmt_message0.doInitBody();
  }
  do {
    out.print("\r\n ");
    if (_fmt_param0 == null)
      _fmt_param0 =
        new org.apache.taglibs.standard.tag.el.fmt.ParamTag();
    _fmt_param0.setPageContext(pageContext);
    _fmt_param0.setParent(
      (javax.servlet.jsp.tagext.Tag)_fmt_message0);
    _activeTag = _fmt_param0;
    _fmt_param0.setValue(
      weblogic.utils.StringUtils.valueOf("${param.test}"));
    _int1 = _fmt_param0.doStartTag();
    weblogic.servlet.jsp.StandardTagLib.fakeEmptyBodyTag(
      pageContext, _fmt_param0, _int1, true);
    if (_fmt_param0.doEndTag() == Tag.SKIP_PAGE) {
      _activeTag = null;
      _releaseTags(_fmt_param0);
      return;
    }
    _activeTag = _fmt_param0.getParent();
    _fmt_param0.release();
    out.print("\r\n ");
```

```
  } while ( _fmt_message0.doAfterBody() ==
            IterationTag.EVAL_BODY_AGAIN);
  if (_int0 == BodyTag.EVAL_BODY_BUFFERED)
    out = pageContext.popBody();
}
if (_fmt_message0.doEndTag() == Tag.SKIP_PAGE) {
  _activeTag = null;
  _releaseTags(_fmt_message0);
  return;
}
_activeTag = _fmt_message0.getParent();
_fmt_message0.release();
_writeText(response, out, _wl_block2, _wl_block2Bytes);
if (_fmt_message0 == null)
  _fmt_message0 =
    new org.apache.taglibs.standard.tag.el.fmt.MessageTag();
```

Summary

Static analysis is useful for many purposes, but it is especially useful for security because it provides a means of thorough analysis that is not otherwise feasible. Table 2.1 lists the static analysis tools discussed in the chapter.

All static analysis tools produce at least some false positives or some false negatives, but most produce both. For security purposes, false negatives are more troublesome than false positives, although too many false positives can lead a reviewer to overlook true positives.

Practical challenges for static analysis tools include the following:

- Making sense of the program (building an accurate program model)
- Making good trade-offs between precision, depth, and scalability
- Looking for the right set of defects
- Presenting easy-to-understand results and errors
- Integrating easily with the build system and integrated development environments

Static analysis tools can analyze source or compiled code. For bytecode languages such as Java, the two approaches are on roughly equal footing. For C and C++ programs, analyzing compiled code is harder and produces inferior results.

Table 2.1 Static analysis tools discussed in the chapter.

Type of Tool/Vendors	Web Site
Style Checking	
PMD	http://pmd.sourceforge.net
Parasoft	http://www.parasoft.com
Program Understanding	
Fujaba	http://wwwcs.uni-paderborn.de/cs/fujaba/
CAST	http://www.castsoftware.com
Program Verification	
Praxis High Integrity Systems	http://www.praxis-his.com
Escher Technologies	http://www.eschertech.com
Property Checking	
Polyspace	http://www.polyspace.com
Grammatech	http://www.gramatech.com
Bug Finding	
FindBugs	http://www.findbugs.org
Coverity	http://www.coverity.com
Visual Studio 2005 \analyze	http://msdn.microsoft.com/vstudio/
Klocwork	http://www.klocwork.com
Security Review	
Fortify Software	http://www.fortify.com
Ounce Labs	http://www.ouncelabs.com

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.

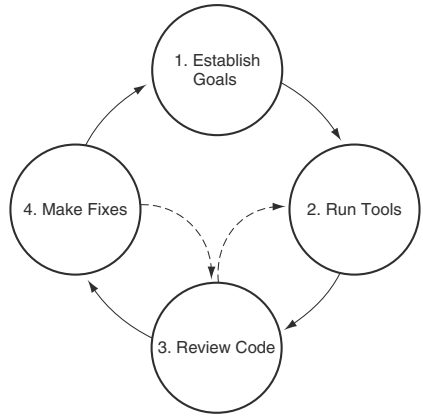

Figure 3.1 The code review cycle.

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.

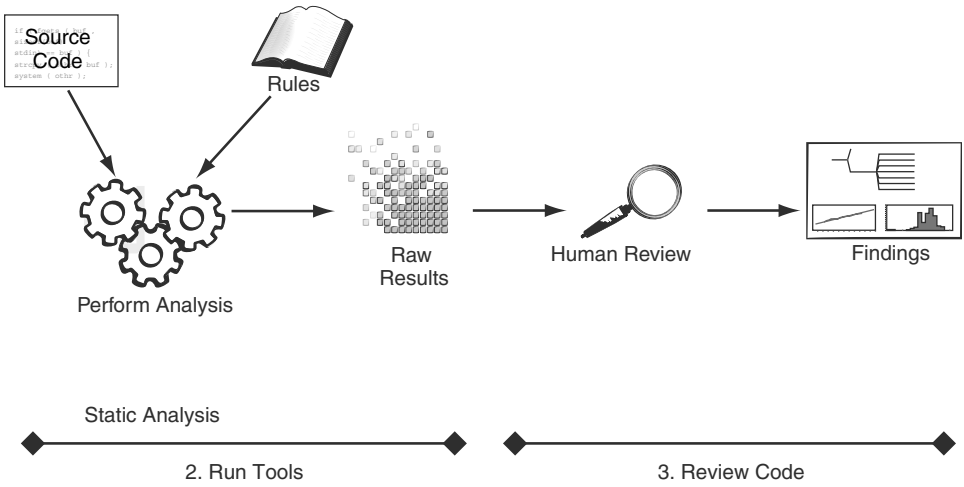

Figure 3.2 Steps 2 and 3: running the tool and reviewing the code.

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.

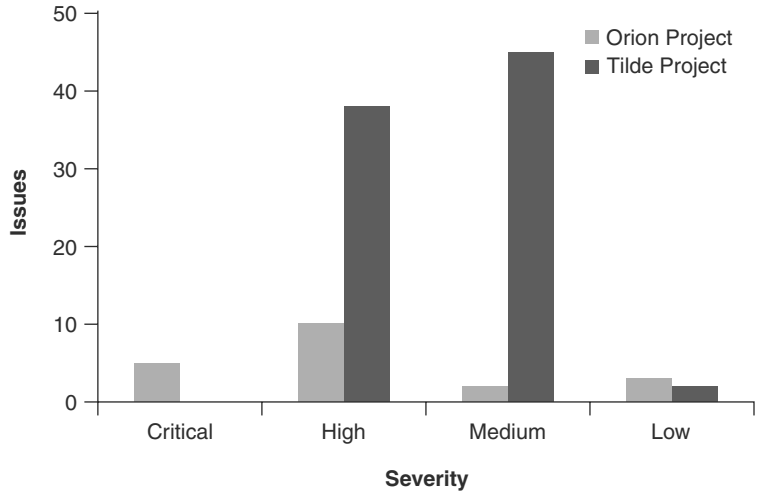

Figure 3.3 Source code analysis results broken down by severity for two subprojects.

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.

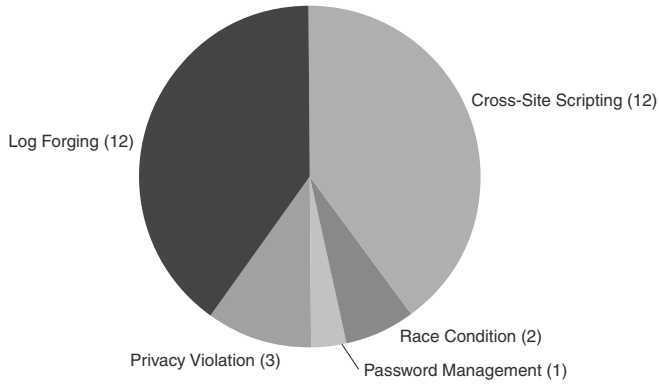

Figure 3.4 Source code analysis results broken down by category.

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.

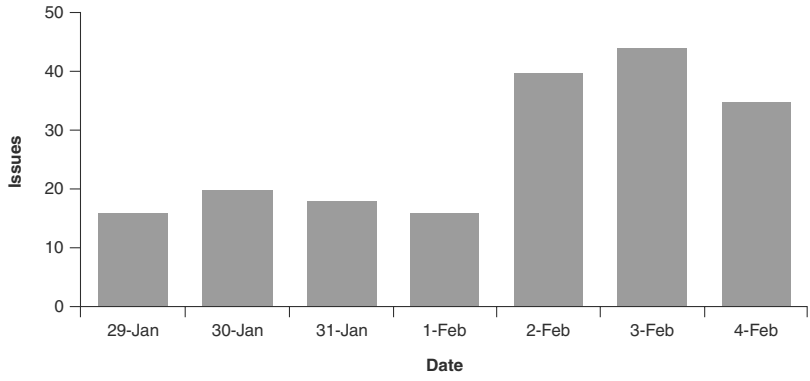

Figure 3.5 Source code analysis results from a series of nightly builds. The spike in issues on February 2 reflects the incorporation of a module originally written by a different team.

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.

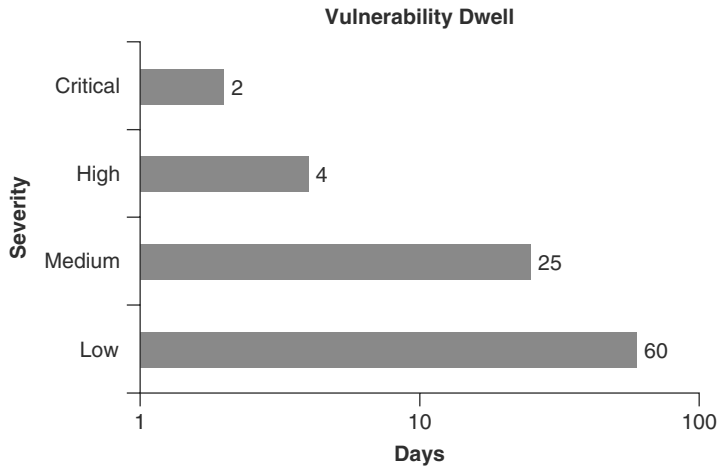

Figure 3.6 Vulnerability dwell as a function of severity. When a vulnerability is identified, vulnerability dwell measures how long it remains in the code. (The x-axis uses a log scale.)

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).

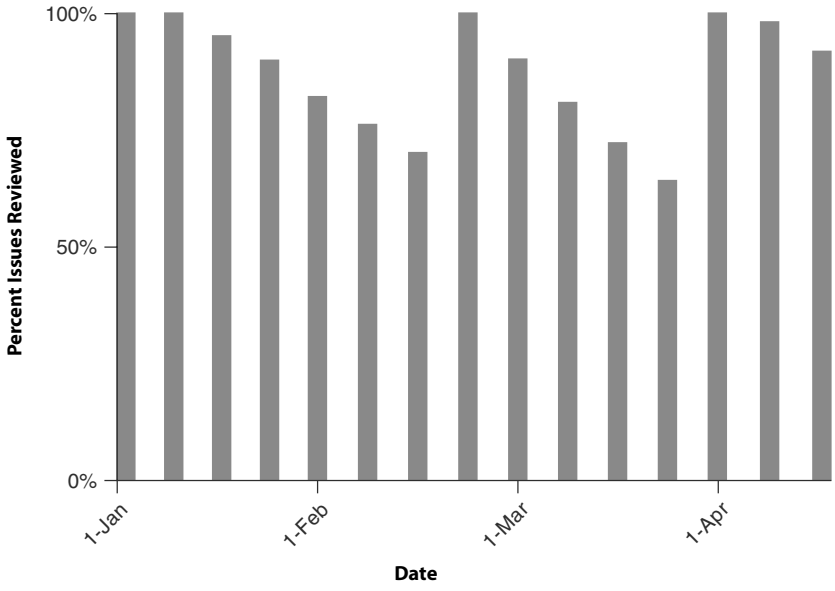

Figure 3.7 Audit coverage over time. After all static analysis results are reviewed, the code continues to evolve and the percentage of reviewed issues begins to decline.

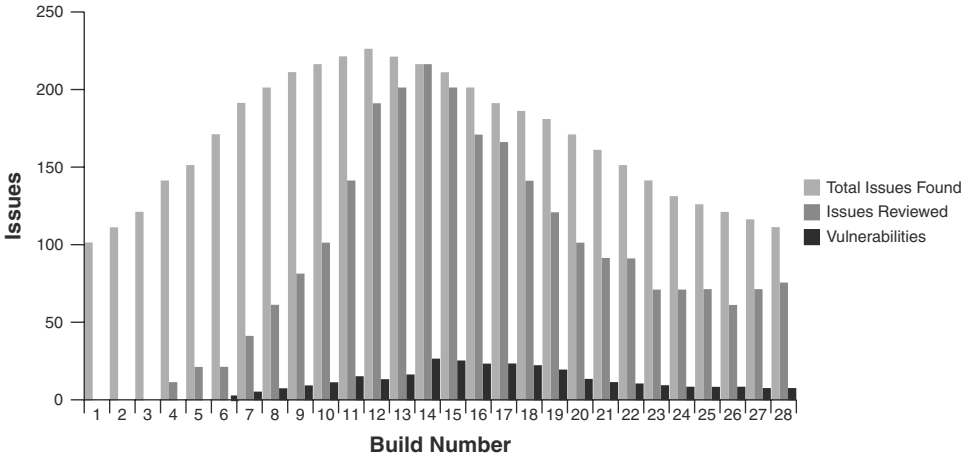

Figure 3.8 Audit history: the total number of static analysis results, the number of reviewed results, and the number of identified vulnerabilities present in the project.

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.

4 Static Analysis Internals

Those who say it cannot be done should not interrupt the people doing it.

—Chinese proverb

This chapter is about what makes static analysis tools tick. We look at the internal workings of advanced static analysis tools, including data structures, analysis techniques, rules, and approaches to reporting results. Our aim is to explain enough about what goes into a static analysis tool that you can derive maximum benefit from the tools you use. For readers interested in creating their own tools, we hope to lay enough groundwork to provide a reasonable starting point.

Regardless of the analysis techniques used, all static analysis tools that target security function in roughly the same way, as shown in Figure 4.1. They all accept code, build a model that represents the program, analyze that model in combination with a body of security knowledge, and finish by presenting their results back to the user. This chapter walks through the process and takes a closer look at each step.

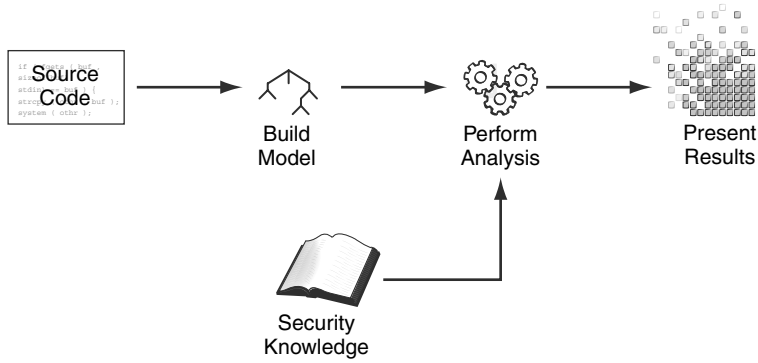

Figure 4.1 A block diagram for a generic static analysis security tool. At a high level, almost all static analysis security tools work this way.

4.1 Building a Model

The first thing a static analysis tool needs to do is transform the code to be analyzed into a *program model,* a set of data structures that represent the code. As you would expect, the model a tool creates is closely linked to the kind of analysis it performs, but generally static analysis tools borrow a lot from the compiler world. In fact, many static analysis techniques were developed by researchers working on compilers and compiler optimization problems. If you are interested in an in-depth look at compilers, we recommend both the classic textbook *Compilers: Principles, Techniques, and Tools* (often called "the dragon book"), by Aho, Sethi, and Ullman [Aho et al., 2006], and Appel's *Modern Compiler Implementation* series (often called "the tiger books") [Appel, 1998].

We now take a brief tour of the most important techniques and data structures that compilers and static analysis tools share.

Lexical Analysis

Tools that operate on source code begin by transforming the code into a series of tokens, discarding unimportant features of the program text such as whitespace or comments along the way. The creation of the token stream is called *lexical analysis.* Lexing rules often use regular expressions to identify tokens. Example 4.1 gives a simple set of lexing rules that could be used to process the following C program fragment:

```
if (ret) // probably true
  mat[x][y] = END_VAL;
```

This code produces the following sequence of tokens:

```
IF LPAREN ID(ret) RPAREN ID(mat) LBRACKET ID(x) RBRACKET LBRACKET
ID(y) RBRACKET EQUAL ID(END_VAL) SEMI
```

Notice that most tokens are represented entirely by their token type, but to be useful, the ID token requires an additional piece of information: the name of the identifier. To enable useful error reporting later, tokens should carry at least one other kind of information with them: their position in the source text (usually a line number and a column number).

For the simplest of static analysis tools, the job is nearly finished at this point. If all the tool is going to do is match the names of dangerous functions, the analyzer can go through the token stream looking for identifiers,

match them against a list of dangerous function names, and report the results. This is the approach taken by ITS4, RATS, and Flawfinder.

Example 4.1 Sample lexical analysis rules.

```
if                          { return IF; }
(                           { return LPAREN; }
)                           { return RPAREN; }
[                           { return LBRACKET; }
]                           { return LBRACKET; }
=                           { return EQUAL; }
;                           { return SEMI; }
/[ \t\n]+/                  { /* ignore whitespace */ }
/\/\/.*/                    { /* ignore comments */ }
/[a-zA-Z][a-zA-Z0-9]*"/     { return ID; }
```

Parsing

A language parser uses a *context-free grammar* (CFG) to match the token stream. The grammar consists of a set of productions that describe the symbols (elements) in the language. Example 4.2 lists a set of productions that are capable of parsing the sample token stream. (Note that the definitions for these productions would be much more involved for a full-blown language parser.)

Example 4.2 Production rules for parsing the sample token stream.

```
stmt := if_stmt | assign_stmt
if_stmt := IF LPAREN expr RPAREN stmt
expr := lval
assign_stmt := lval EQUAL expr SEMI
lval = ID | arr_access
arr_access := ID arr_index+
arr_idx := LBRACKET expr RBRACKET
```

The parser performs a *derivation* by matching the token stream against the production rules. If each symbol is connected to the symbol from which it was derived, a *parse tree* is formed. Figure 4.2 shows a parse tree created using the production rules from Example 4.2. We have omitted terminal symbols that do not carry names (`IF`, `LPAREN`, `RPAREN`, etc.), to make the salient features of the parse tree more obvious.

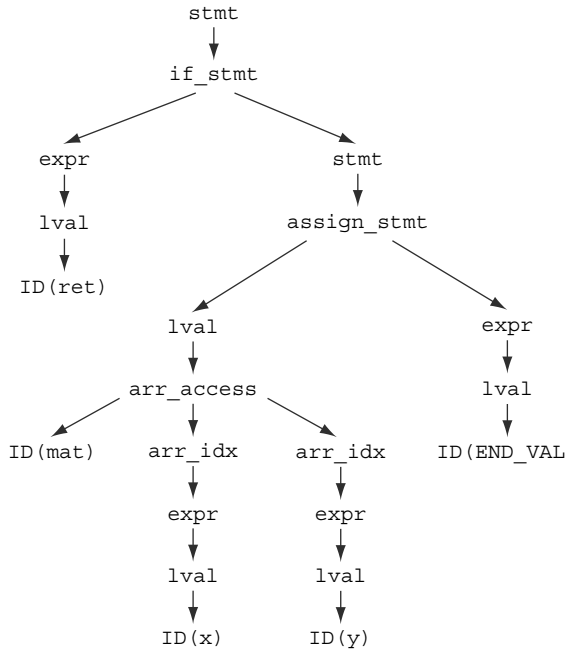

Figure 4.2 A parse tree derived from the sequence of tokens.

If you would like to build your own parser, the venerable UNIX programs Lex and Yacc have been the traditional way to start in C; if you can choose any language you like, we prefer JavaCC (https://javacc.dev.java.net) because it's all-around easier to use and comes complete with a grammar for parsing Java. For C and C++, the Edison Design Group (EDG) (http://www.edg.com) sells an excellent front end. EDG sometimes makes its toolkit available for free for academic use. The open source Elsa C and C++ parser from U.C. Berkeley is another good option (http:// www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/).

Abstract Syntax

It is feasible to do significant analysis on a parse tree, and certain types of stylistic checks are best performed on a parse tree because it contains the most direct representation of the code just as the programmer wrote it. However, performing complex analysis on a parse tree can be inconvenient for a number of reasons. The nodes in the tree are derived directly from the grammar's production rules, and those rules can introduce nonterminal

symbols that exist purely for the purpose of making parsing easy and nonambiguous, rather than for the purpose of producing an easily understood tree; it is generally better to abstract away both the details of the grammar and the syntactic sugar present in the program text. A data structure that does these things is called an *abstract syntax tree (AST)*. The purpose of the AST is to provide a standardized version of the program suitable for later analysis. The AST is usually built by associating tree construction code with the grammar's production rules.

Figure 4.3 shows an AST for our example. Notice that the `if` statement now has an empty `else` branch, the predicate tested by the `if` is now an explicit comparison to zero (the behavior called for by C), and array access is uniformly represented as a binary operation.

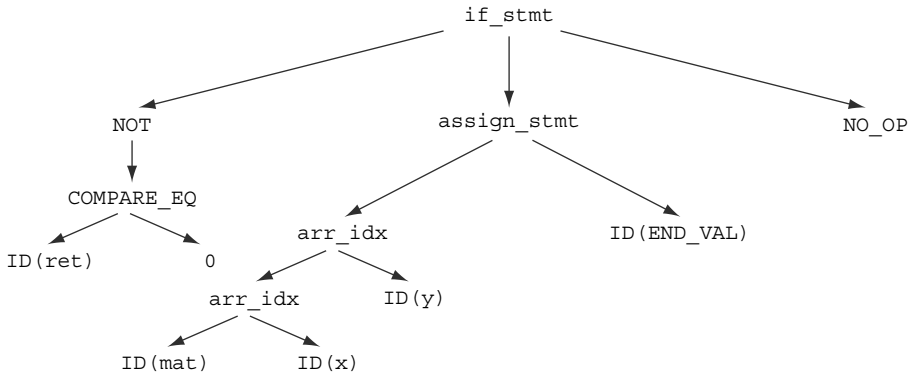

Figure 4.3 An abstract syntax tree.

Depending on the needs of the system, the AST can contain a more limited number of constructs than the source language. For example, method calls might be converted to function calls, or `for` and `do` loops might be converted to `while` loops. Significant simplification of the program in this fashion is called *lowering*. Languages that are closely related, such as C and C++, can be lowered into the same AST format, although such lowering runs the risk of distorting the programmer's intent. Languages that are syntactically similar, such as C++ and Java, might share many of the same AST node types but will almost undoubtedly have special kinds of nodes for features found in only one language.

Semantic Analysis

As the AST is being built, the tool builds a *symbol table* alongside it. For each identifier in the program, the symbol table associates the identifier with its type and a pointer to its declaration or definition.

With the AST and the symbol table, the tool is now equipped to perform type checking. A static analysis tool might not be required to report type-checking errors the way a compiler does, but type information is critically important for the analysis of an object-oriented language because the type of an object determines the set of methods that the object can invoke. Furthermore, it is usually desirable to at least convert implicit type conversions in the source text into explicit type conversions in the AST. For these reasons, an advanced static analysis tool has to do just as much work related to type checking as a compiler does.

In the compiler world, symbol resolution and type checking are referred to as *semantic analysis* because the compiler is attributing meaning to the symbols found in the program. Static analysis tools that use these data structures have a distinct advantage over tools that do not. For example, they can correctly interpret the meaning of overloaded operators in C++ or determine that a Java method named `doPost()` is, in fact, part of an implementation of `HttpServlet`. These capabilities enable a tool to perform useful checks on the structure of the program. We use the term *structural analysis* for these kinds of checks. For more, see the sidebar "Checking Structural Rules."

After semantic analysis, compilers and more advanced static analysis tools part ways. A modern compiler uses the AST and the symbol and type information to generate an *intermediate representation,* a generic version of machine code that is suitable for optimization and then conversion into platform-specific object code. The path for static analysis tools is less clear cut. Depending on the type of analysis to be performed, a static analysis tool might perform additional transformations on the AST or might generate its own variety of intermediate representation suitable to its needs.

If a static analysis tool uses its own intermediate representation, it generally allows for at least assignment, branching, looping, and function calls (although it is possible to handle function calls in a variety of ways—we discuss function calls in the context of interprocedural analysis in Section 4.4). The intermediate representation that a static analysis tool uses is usually a higher-level view of the program than the intermediate representation that a compiler uses. For example, a C language compiler likely will convert all references to structure fields into byte offsets into the structure for its intermediate representation, while a static analysis tool more likely will continue to refer to structure fields by their names.

Checking Structural Rules

While he was an intern at Fortify, Aaron Siegel created a language for describing structural program properties. In the language, a property is defined by a target AST node type and a predicate that must be true for an instance of that node type to match. Two examples follow.

In Java, database connections should not be shared between threads; therefore, database connections should not be stored in static fields. To find static fields that hold database connections, we write

```
Field: static and type.name == "java.sql.Connection"
```

In C, the statement

```
buf = realloc(buf, 256);
```

causes a memory leak if `realloc()` fails and returns `null`. To flag such statements, we write

```
FunctionCall c1: (
  c1.function is [name == "realloc"] and
  c1 in [AssignmentStatement: rhs is c1 and
         lhs == c1.arguments[0]
        ]
)
```

Tracking Control Flow

Many static analysis algorithms (and compiler optimization techniques) explore the different execution paths that can take place when a function is executed. To make these algorithms efficient, most tools build a *control flow graph* on top of the AST or intermediate representation. The nodes in a control flow graph are *basic blocks*: sequences of instructions that will always be executed starting at the first instruction and continuing to the last instruction, without the possibility that any instructions will be skipped. Edges in the control flow graph are directed and represent potential control flow paths between basic blocks. Back edges in a control flow graph represent potential loops. Consider the C fragment in Example 4.3.

Example 4.3 A C program fragment consisting of four basic blocks.

```
if (a > b) {
  nConsec = 0;
} else {
  s1 = getHexChar(1);
  s2 = getHexChar(2);
}
return nConsec;
```

Figure 4.4 presents the control flow graph for the fragment. The four basic blocks are labeled `bb0` through `bb3`. In the example, the instructions in each basic block are represented in source code form, but a basic block data structure in a static analysis tool would likely hold pointers to AST nodes or the nodes for the tool's intermediate representation.

When a program runs, its control flow can be described by the series of basic blocks it executes. A *trace* is a sequence of basic blocks that define a path through the code. There are only two possible execution paths through the code in Example 4.3 (the branch is either taken or not). These paths are represented by two unique traces through the control flow graph in Figure 4.4: `[ bb0, bb1, bb3 ]` and `[ bb0, bb2, bb3 ]`.

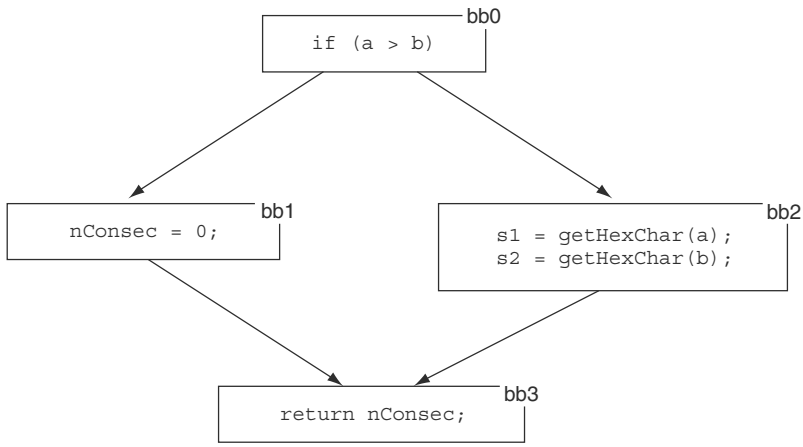

Figure 4.4 A control flow graph with four basic blocks.

A *call graph* represents potential control flow between functions or methods. In the absence of function pointers or virtual methods, constructing a

call graph is simply a matter of looking at the function identifiers referenced in each function. Nodes in the graph represent functions, and directed edges represent the potential for one function to invoke another. Example 4.4 shows a program with three functions, and Figure 4.5 shows its call graph. The call graph makes it clear that `larry` can call `moe` or `curly`, and `moe` can call `curly` or call itself again recursively.

Example 4.4 A short program with three functions.

```
int larry(int fish) {
  if (fish) {
    moe(1);
  } else  {
    curly();
  }

}

int moe(int scissors) {
  if (scissors) {
    curly();
    moe(0);
  } else {
    curly();
  }
}

int curly() {
  /* empty */
}
```

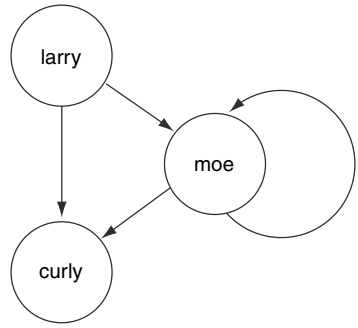

Figure 4.5 The call graph for the program in Example 4.4.

When function pointers or virtual methods are invoked, the tool can use a combination of dataflow analysis (discussed next) and data type analysis to limit the set of potential functions that can be invoked from a call site. If the program loads code modules dynamically at runtime, there is no way to be sure that the control flow graph is complete because the program might run code that is not visible at the time of analysis.

For software systems that span multiple programming languages or consist of multiple cooperating processes, a static analysis tool will ideally stitch together a control flow graph that represents the connections between the pieces. For some systems, the configuration files hold the data needed to span the call graphs in different environments.

Tracking Dataflow

Dataflow analysis algorithms examine the way data move through a program. Compilers perform dataflow analysis to allocate registers, remove dead code, and perform many other optimizations.

Dataflow analysis usually involves traversing a function's control flow graph and noting where data values are generated and where they are used. Converting a function to *Static Single Assignment (SSA)* form is useful for many dataflow problems. A function in SSA form is allowed to assign a value to a variable only once. To accommodate this restriction, new variables must be introduced into the program. In the compiler literature, the new variables are usually represented by appending a numeric subscript to the original variable's name, so if the variable `x` is assigned three times, the rewritten program will refer to variables x_1, x_2, and x_3.

SSA form is valuable because, given any variable in the program, it is easy to determine where the value of the variable comes from. This property has many applications. For example, if an SSA variable is ever assigned a constant value, the constant can replace all uses of the SSA variable. This technique is called *constant propagation*. Constant propagation by itself is useful for finding security problems such as hard-coded passwords or encryption keys.

Example 4.5 lists an excerpt from the Tiny Encryption Algorithm (TEA). The excerpt first appears as it normally would in the program source text and next appears in SSA form. This example is simple because it is straight-line code without any branches.

Example 4.5 An excerpt from the TEA encryption algorithm, both in its regular source code form and in static single assignment form.

Regular source code form:

```
sum = sum + delta ;
sum = sum & top;
y = y + (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1];
y = y & top;
z = z + (y<<4)+k[2] ^ y+sum ^ (y>>5)+k[3];
z = z & top;
```

SSA form:

```
sum2 = sum1 + delta1 ;
sum3 = sum2 & top1;
y2 = y1 + (z1<<4)+k[0]1 ^ z1+sum3 ^ (z1>>5)+k[1]1;
y3 = y2 & top1;
z2 = z1 + (y3<<4)+k[2]1 ^ y3+sum3 ^ (y3>>5)+k[3]1;
z3 = z2 & top1;
```

If a variable is assigned different values along different control flow paths, in SSA form, the variable must be reconciled at the point where the control flow paths merge. SSA accomplishes this merge by introducing a new version of the variable and assigning the new version the value from one of the two control flow paths. The notational shorthand for this merge point is called a ϕ-*function*. The ϕ-function stands in for the selection of the appropriate value, depending upon the control flow path that is executed. Example 4.6 gives an example in which conversion to SSA form requires introducing a ϕ function.

Example 4.6 Another example of conversion to SSA form. The code is shown first in its regular source form and then in its SSA form. In this case, two control flow paths merge at the bottom of the `if` block, and the variable `tail` must be reconciled using a ϕ-function.

Regular source form:

```
if (bytesRead < 8) {
  tail = (byte) bytesRead;
}
```

SSA form:

```
if (bytesRead1 < 8) {
  tail2 = (byte) bytesRead1;
}
tail3 = φ(tail1, tail2);
```

Taint Propagation

Security tools need to know which values in a program an attacker could potentially control. Using dataflow to determine what an attacker can control is called *taint propagation*. It requires knowing where information enters the program and how it moves through the program. Taint propagation is the key to identifying many input validation and representation defects. For example, a program that contains an exploitable buffer overflow vulnerability almost always contains a dataflow path from an input function to a vulnerable operation. We discuss taint propagation further when we look at analysis algorithms and then again when we look at rules.

The concept of tracking tainted data is not restricted to static analysis tools. Probably the most well-known implementation of dynamic taint propagation is Perl's taint mode, which uses a runtime mechanism to make sure that user-supplied data are validated against a regular expression before they are used as part of a sensitive operation.

Pointer Aliasing

Pointer alias analysis is another dataflow problem. The purpose of alias analysis is to understand which pointers could possibly refer to the same memory location. Alias analysis algorithms describe pointer relationships with terms such as "must alias," "may alias," and "cannot alias." Many compiler optimizations require some form of alias analysis for correctness. For example, a compiler would be free to reorder the following two statements only if the pointers `p1` and `p2` do not refer to the same memory location:

```
*p1 = 1;
*p2 = 2;
```

For security tools, alias analysis is important for performing taint propagation. A flow-sensitive taint-tracking algorithm needs to perform alias analysis to understand that data flow from `getUserInput()` to `processInput()` in the following code:

```
p1 = p2;
*p1 = getUserInput();
processInput(*p2);
```

It is common for static analysis tools to assume that pointers—at least pointers that are passed as function arguments—do not alias. This assumption seems to hold often enough for many tools to produce useful results, but it could cause a tool to overlook important results.

4.2 Analysis Algorithms

The motivation for using advanced static analysis algorithms is to improve context sensitivity—to determine the circumstances and conditions under which a particular piece of code runs. Better context sensitivity enables a better assessment of the danger the code represents. It's easy to point at all calls to `strcpy()` and say that they should be replaced, but it's much harder to call special attention to only the calls to `strcpy()` that might allow an attacker to overflow a buffer.

Any advanced analysis strategy consists of at least two major pieces: an *intraprocedural analysis* component for analyzing an individual function, and an *interprocedural analysis* component for analyzing interaction between functions. Because the names *intraprocedural* and *interprocedural* are so similar, we use the common vernacular terms *local analysis* to mean intraprocedural analysis, and *global analysis* to mean interprocedural analysis. Figure 4.6 diagrams the local analysis and global analysis components, and associates the major data structures commonly used by each.

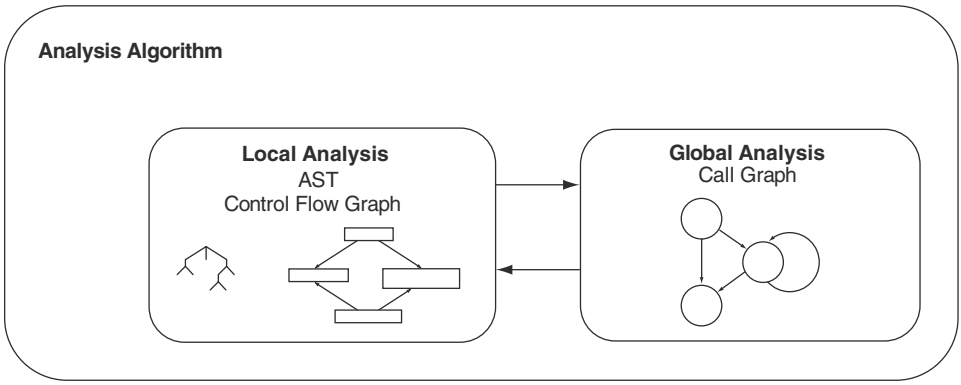

Figure 4.6 An analysis algorithm includes a local component and a global component.

Checking Assertions

Many security properties can be stated as assertions that must be true for the program to be secure. To check for a buffer overflow in the following line of code:

```
strcpy(dest, src);
```

imagine adding this assertion to the program just before the call to `strcpy()`:

```
assert(alloc_size(dest) > strlen(src));
```

If the program logic guarantees that this assertion will always succeed, no buffer overflow is possible.[1] If there are a set of conditions under which the assertion might fail, the analyzer should report a potential buffer overflow. This same assertion-based approach works equally well for defining the requirements for avoiding SQL injection, cross-site scripting, and most of the other vulnerability categories we discuss in this book.

For the remainder of this section, we treat static analysis as an assertion-checking problem. Choosing the set of assertions to make is the topic of Section 4.3, leaving this section to discuss how assertion checking can be performed. Drawing a distinction between the mechanics of performing the check and the particulars of what should be checked is valuable for more than just explicative purposes; it is also a good way to build a static analysis tool. By separating the checker from the set of things to be checked, the tool can quickly be adapted to find new kinds of problems or prevented from reporting issues that are not problems. From an engineering standpoint, designing a checker and deciding what to check are both major undertakings, and convoluting them would make for an implementation quagmire.

We typically see three varieties of assertions that arise from security properties:

- The most prevalent forms of security problems arise from programmers who trust input when they should not, so a tool needs to check assertions related to the level of trust afforded to data as they move through

1. For the purposes of this example, we have made up a function named `alloc_size()` that returns the number of allocated bytes that its argument points to. Note that the size of `dest` must be strictly greater than the string length of `src`. If the destination buffer is exactly the same size as the source string, `strcpy()` will write a null terminator outside the bounds of `dest`.

the program. These are the taint propagation problems. SQL injection and cross-site scripting are two vulnerability types that will cause a tool to make assertions about taint propagation. In the simplest scenario, a data value is either tainted (potentially controlled by an attacker) or untainted. Alternatively, a piece of data might carry one or more particular kinds of taint. An attacker might be able to control the contents of a buffer but not the size of the buffer, for example.

- Looking for exploitable buffer overflow vulnerabilities leads to assertions that are similar to the ones that arise from taint propagation, but determining whether a buffer can be overflowed requires tracking more than just whether tainted data are involved; the tool also needs to know the size of the buffer and the value used as an index. We term these *range analysis* problems because they require knowing the range of potential values a variable (or a buffer size) might have.
- In some cases, tools are less concerned with particular data values and more concerned with the state of an object as the program executes. This is called *type state*—variables can have a different type at each point in the code. For example, imagine a memory region as being in either the allocated state (after `malloc()` returns a pointer to it) or the freed state (entered when it is passed to the function `free()`). If a program gives up all references to the memory while it is in the allocated state, the memory is leaked. If a pointer to the memory is passed to `free()` when it is in the freed state, a double free vulnerability is present. Many such temporal safety properties can be expressed as small finite-state automata (state machines).

Naïve Local Analysis

With assertion checking in mind, we approach static analysis from a naïve perspective, demonstrate the difficulties that arise, and then discuss how static analysis tools overcome these difficulties. Our effort here is to provide an informal perspective on the kinds of issues that make static analysis challenging.

Consider a simple piece of code:

```
x = 1;
y = 1;
assert(x < y);
```

How could a static analysis tool evaluate the assertion? One could imagine keeping track of all the facts we know about the code before each statement is executed, as follows:

`x = 1;`	{} (no facts)
`y = 1;`	{ x = 1 }
`assert (x < y);`	{ x = 1, y = 1 }

When the static analysis tool reaches the assertion, it can evaluate the expression `a < b` in the context of the facts it has collected. By substituting the variable's values in the assertion, the expression becomes this:

```
1 < 1
```

This is always false, so the assertion will never hold and the tool should report a problem with the code. This same technique could also be applied even if the values of the variables are nonconstant:

```
x = v;
y = v;
assert(x < y);
```

Again tracking the set of facts known before each statement is executed, we have this:

`x = v;`	{} (no facts)
`y = v;`	{ x = v }
`assert (x < y);`	{ x = v, y = v }

Substituting the variable values in the assert statement yields this:

```
v < v
```

Again, this is never true, so the assertion will fail and our static analysis algorithm should again report a problem. This approach is a form of *symbolic simulation* because the analysis algorithm did not consider any concrete values for the variables. By allowing the variable v to be represented as a symbol, we evaluated the program for all possible values of v.

If we had to contend with only straight-line code, static analysis would be easy. Conditionals make matters worse. How should we evaluate the following code?

```
x = v;
if (x < y) {
  y = v;
}
assert (x < y);
```

We need to account for the fact that the conditional predicate `x < y` might or might not be true. One way to do this is to independently follow all the paths through the code and keep track of the facts we collect along each path. Because the code has only one branch, there are only two possible paths.

When the branch is taken (`x < y` is true):

`x = v;`	{} (no facts)
`if (x < y)`	{ x = v }
`y = v;`	{ x = v, x < y }
`assert (x < y)`	{ x = v, x < y, y = v }

When the branch is not taken (`x < y` is false):

`x = v;`	{} (no facts)
`if (x < y)`	{ x = v }
`assert (x < y)`	{ x = v, ¬(x < y) }

(See the sidebar "Notation" for an explanation of the symbols used here.)

Notation

We use the following logical operators, quantifiers, and set notation in this section:

- Conjunction (AND): ∧
- Disjunction (OR): ∨
- Negation (NOT): ¬
- Existential quantification (There exists): ∃
- Universal quantification (For all): ∀
- Contains (is in the set): ∈

For example, to say "x and not x is false," we write `x ∧ ¬x = false`.

When the branch is taken, the set of facts is the same as for the previous example. Because this statement is not true, the assertion is violated:

```
v < v
```

When the branch is not taken, checking the assertion requires us to evaluate the conjunction of the assertion predicate with all the facts we have gathered:

```
(x < y) ∧ (x = v) ∧ ¬(x < y)
```

The fact `x = v` is not useful, but the two inequalities are contradictory, so the assertion fails yet again. (See the sidebar "Prove It," at the end of this section, for more on how this determination might be made.) We have shown that, regardless of whether the branch is taken, the assertion will fail.

This approach to evaluating branches is problematic. The number of paths through the code grows exponentially with the number of conditionals, so explicitly gathering facts along each path would make for an unacceptably slow analyzer. This problem can be alleviated to some degree by allowing paths to share information about common subpaths and with techniques that allow for implicit enumeration of paths. *Program slicing* removes all the code that cannot affect the outcome of the assert predicate. The tool also needs a method for eliminating *false paths,* which are paths through the code that can never be executed because they are logically inconsistent.

The situation becomes much worse when the code being evaluated contains a loop. Our symbolic simulation would need to repeatedly evaluate the body of the loop until the loop is terminated or until the simulator could gain no new facts. (This is called establishing a *fixed point* for the loop.) If the loop can iterate a variable number of times, there is no simple way to know how many times we should simulate the loop body.

Approaches to Local Analysis

These problems related to branching and looping push static analysis tool towards methods for performing less precise but more dependable analysis. We now examine a few of the more popular avenues of research.

Abstract Interpretation

Abstract interpretation is a general technique formalized by Cousot and Cousot for abstracting away aspects of the program that are not relevant to the properties of interest and then performing an interpretation (potentially similar to the interpretation we have been doing in the previous examples) using the chosen program abstraction [Cousot, 1996].

The problems introduced by loops can be solved by performing a *flow-insensitive analysis,* in which the order the statements are executed is not taken into account. From a programmer's point of view, this might seem to be a rather extreme measure; the order in which statements appear in a program is important. But a flow-insensitive analysis guarantees that all orderings of statements are considered, which guarantees that all feasible statement execution orders are accounted for. This eliminates the need for sophisticated control flow analysis, but the price is that many impossible statement execution orders might be analyzed as well. Tools that are making an effort to rein in false positives usually try to be at least partially flow sensitive so that they don't report problems for statement orderings that could never possibly occur when the program runs.

Predicate Transformers

An alternative to simulation or interpretation is to derive the requirements that a function places upon its callers. Dijkstra introduced the notion of a *weakest precondition (WP)* derived from a program using *predicate transformers* [Dijkstra, 1976]. A weakest precondition for a program is the fewest (i.e., the weakest) set of requirements on the callers of a program that are necessary to arrive at a desired final state (a postcondition). The statements

in a program can be viewed as performing transformations on the postcondition. Working backward from the last statement in a program to the first, the program's desired postcondition is transformed into the precondition necessary for the program to succeed.

For example, to complete successfully, the statement

```
assert (x < y);
```

requires that the code leading up to it at least satisfy the requirement `(x < y)`. It is possible that the code leading up to the statement always satisfies a stronger requirement, such as this:

```
(x < 0 ∧ y > 0)
```

However, the assert statement does not require this. If we are interested in having a program reach a final state `r`, we can write a predicate transformer for deriving the weakest precondition for an assert statement as follows:

```
WP(assert(p), r) = p ∧ r
```

Predicate transformers are appealing because, by generating a precondition for a body of code, they abstract away the details of the program and create a summary of the requirements that the program imposes on the caller.

Model Checking

For temporal safety properties, such as "memory should be freed only once" and "only non-null pointers should be dereferenced," it is easy to represent the property being checked as a small finite-state automaton. Figure 4.7 shows a finite-state automaton for the property "memory should be freed only once." A *model checking* approach accepts such properties as specifications, transforms the program to be checked into an automaton (called the model), and then compares the specification to the model. For example, in Figure 4.7, if the model checker can find a variable and a path through the program that will cause the specification automaton to reach its error state, the model checker has identified the potential for a double free vulnerability.

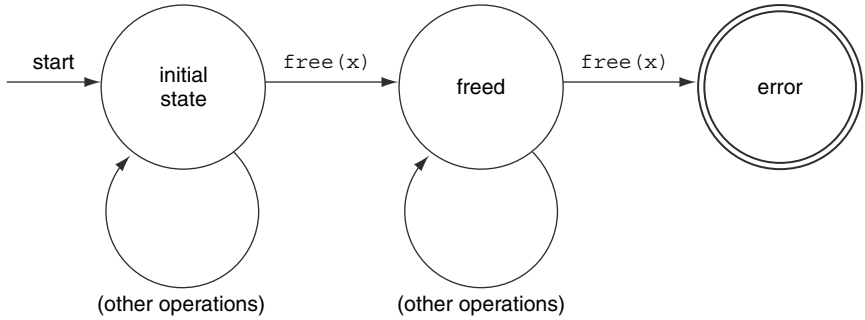

Figure 4.7 A finite-state automaton for the temporal safety property "memory should be freed only once."

Global Analysis

The simplest possible approach to global analysis is to ignore the issue, to assume that all problems will evidence themselves if the program is examined one function at a time. This is a particularly bad assumption for many security problems, especially those related to input validation and representation, because identifying these problems often requires looking across function boundaries.

Example 4.7 shows a program that contains a buffer overflow vulnerability. To identify the vulnerability, a tool needs to track that an unbounded amount of data from the environment (`argv[0]`) are being passed from `main()` to `setname()` and copied into a fixed-size buffer.

Just about all advanced security tools make an effort to identify bugs that involve more than one function.

Example 4.7 Accurately identifying this buffer overflow vulnerability requires looking across function boundaries.

```
static char progName[128];

void setname(char* newName) {
  strcpy(progName, newName);
}

int main(int argc, char* argv[]) {
  setname(argv[0]);
}
```

The most ambitious approach to global analysis is *whole-program analysis,* whose objective is to analyze every function with a complete understanding of the context of its calling functions. This is an extreme example of a *context-sensitive analysis,* whose objective is to take into account the context of the calling function when it determines the effects of a function call. A conceptually simple way to achieve whole-program analysis is *inlining,* replacing each function call in the program with the definition of the called function. (Recursion presents a challenge.) Other approaches are also possible, including the use a stack-based analysis model. Regardless of the technique, whole-program analysis can require a lot of time, a lot of memory, or both.

A more flexible approach to global analysis is to leverage a local analysis algorithm to create *function summaries*. With a function summary approach, when a local analysis algorithm encounters a function call, the function's summary is applied as a stand-in for the function. A function's summary can be very precise (and potentially very complex) or very imprecise (and presumably less complex), allowing the summary-generation and storage algorithm to make a trade-off between precision and scalability. A function summary might include both requirements that the calling context must meet (preconditions) and the effect that the function has had on the calling context when it returns (postconditions). Example 4.8 shows a summary for the C standard library function `memcpy()`. In English, the summary says: "`memcpy()` requires that its callers ensure that the size of the `dest` buffer and the size of the `src` buffer are both greater than or equal to the value of the `len` parameter. When it returns, `memcpy()` guarantees that the values in `dest` will be equal to the values in `src` in locations 0 through the value of `len` minus 1."

Example 4.8 A summary for the C function `memcpy()`.

```
memcpy(dest, src, len) [
  requires:
    ( alloc_size(dest) >= len ) ∧ ( alloc_size(src) >= len )
  ensures:
    ∀ i ∈ 0 .. len-1: dest[i]' == src[i]
]
```

Building and using function summaries often implies that global analysis is carried out by a *work-queue algorithm* that uses a local analysis subroutine

to find bugs and produce function summaries. Example 4.9 gives pseudocode for two procedures that together implement a work-queue algorithm for performing global analysis.

The `analyze_program()` procedure accepts two parameters: a program to be analyzed and a set of function summaries. The initial set of summaries might be characterizations for library functions. It first builds a call graph, then queues up all the functions in the program, and then pulls functions off the queue and analyzes each one until the queue is empty.

The `analyze_function()` procedure relies on a local analysis algorithm (not shown) that checks the function for vulnerabilities and also updates the function summary, if necessary. If a function's summary is updated, all the callers of the function need to be analyzed again so that they can use the new summary.

Depending on the specifics of the analysis, it might be possible to speed up `analyze_program()` by adjusting the order in which individual functions are analyzed.

Example 4.9 Pseudocode for a global analysis algorithm using function summaries.

```
analyze_program(p, summaries) {
  cg = build_callgraph(p)
  for each function f in p {
    add f to queue
  }
  while (queue is not empty) {
    f = first function in queue
    remove f from queue
    analyze_function(f, queue, cg, summaries);
  }
}

analyze_function(f, queue, cg, summaries) {
  old = get summary for f from summaries
  do_local_analysis(f, summaries);
  new = get summary for f from summaries
  if (old != new) {
    for each function g in cg that calls f {
      if (g is not in queue) {
        add g to queue
      }
    }
  }
}
```

Research Tools

The following is a brief overview of some of the tools that have come out of research labs and universities in the last few years:[2]

- ARCHER is a static analysis tool for checking array bounds. (The name stands for ARray CHeckER.) ARCHER uses a custom-built solver to perform a path-sensitive interprocedural analysis of C programs. It has been used to find more than a dozen security problems in Linux, and it has found hundreds of array bounds errors in OpenBSD, Sendmail, and PostgreSQL [Xie et al., 2003].
- The tool BOON applies integer range analysis to determine whether a C program is capable of indexing an array outside its bounds [Wagner et al., 2000]. Although it is capable of finding many errors that lexical analysis tools would miss, the checker is still imprecise: It ignores statement order, it can't model interprocedural dependencies, and it ignores pointer aliasing.
- Inspired by Perl's taint mode, CQual uses type qualifiers to perform a taint analysis to detect format string vulnerabilities in C programs [Foster et al., 2002]. CQual requires a programmer to annotate a small number of variables as either tainted or untainted, and then uses type-inference rules (along with preannotated system libraries) to propagate the qualifiers. After the qualifiers have been propagated, the system can detect format string vulnerabilities by type checking.
- The Eau Claire tool uses a theorem prover to create a general specification-checking framework for C programs [Chess, 2002]. It can be used to find such common security problems as buffer overflows, file access race conditions, and format string bugs. The system checks the use of standard library functions using prewritten specifications. Developers can also use specifications to ensure that function implementations behave as expected. Eau Claire is built on the same philosophical foundation as the extended static checking tool ESC/Java2 [Flanagan et al., 2002].
- LAPSE, short for Lightweight Analysis for Program Security in Eclipse, is an Eclipse plug-in targeted at detecting security vulnerabilities in J2EE applications. It performs taint propagation to connect sources of Web input with potentially sensitive operations. It detects vulnerabilities such as SQL injection, cross-site scripting, cookie poisoning, and parameter manipulation [Livshits and Lam, 2005].

2. Parts of this section originally appeared in *IEEE Security & Privacy Magazine* as part of an article coauthored with Gary McGraw [Chess and McGraw, 2004].

- MOPS (MOdel checking Programs for Security properties) takes a model checking approach to look for violations of temporal safety properties in C programs [Chen and Wagner, 2002]. Developers can model their own safety properties, and MOPS has been used to identify privilege management errors, incorrect construction of chroot jails, file access race conditions, and ill-conceived temporary file schemes in large-scale systems, such as the Red Hat Linux 9 distribution [Schwarz et al., 2005].
- SATURN applies Boolean satisfiability to detect violations of temporal safety properties. It uses a summary-based approach to interprocedural analysis. It has been used to find more than 100 memory leaks and locking problems in Linux [Xie and Aiken, 2005].
- Splint extends the lint concept into the security realm [Larochelle and Evans, 2001]. Without adding any annotations, developers can perform basic lint-like checks. By adding annotations, developers can enable Splint to find abstraction violations, unannounced modifications to global variables, and possible use-before-initialization errors. Splint can also reason about minimum and maximum array bounds accesses if it is provided with function preconditions and postconditions.
- Pixy detects cross-site scripting vulnerabilities in PHP programs [Jovanovic et al., 2006]. The authors claim that their interprocedural context and flow-sensitive analysis could easily be applied to other taint-style vulnerabilities such as SQL injection and command injection.
- The xg++ tool uses a template-driven compiler extension to attack the problem of finding kernel vulnerabilities in the Linux and OpenBSD operating systems [Ashcraft and Engler, 2002]. The tool looks for locations where the kernel uses data from an untrusted source without checking first, methods by which a user can cause the kernel to allocate memory and not free it, and situations in which a user could cause the kernel to deadlock. Similar techniques applied to general code quality problems such as null pointer dereferences led the creators of xg++ to form a company: Coverity.

A number of static analysis approaches hold promise but have yet to be directly applied to security. Some of the more noteworthy ones include ESP (a large-scale property verification approach) [Das et al., 2002] and model checkers such as SLAM [Ball et al., 2001] and BLAST [Henzinger et al., 2003].

Prove It

Throughout this discussion, we have quickly moved from an equation such as

$$(x < y) \wedge (x = v) \wedge \neg(x < y)$$

to the conclusion that an assertion will succeed or fail. For a static analysis tool to make this same conclusion, it needs to use a *constraint solver.* Some static analysis tools have their own specialized constraint solvers, while others use independently developed solvers. Writing a good solver is a hard problem all by itself, so if you create your own, be sure to create a well-defined interface between it and your constraint-generation code. Different solvers are good for different problems, so be sure your solver is well matched to the problems that need to be solved.

Popular approaches to constraint solving include the Nelson-Oppen architecture for cooperating decision procedures [Nelson, 1981] as implemented by Simplify [Detlefs et al., 1996]. Simplify is used by the static analysis tools Esc/Java [Flanagan et al., 2002] and Eau Claire [Chess, 2002]. In recent years, Boolean satisfiability solvers (SAT solvers) such as zChaff [Moskewicz et al., 2001] have become efficient enough to make them effective for static analysis purposes. The static analysis tool SATURN [Xie and Aiken, 2005] uses zChaff. Packages for manipulating binary decision diagrams (BDDs), such as BuDDy (http://sourceforge.net/projects/buddy/), are also seeing use in tools such as Microsoft SLAM [Ball et al., 2001].

Examples of static analysis tools that use custom solvers include the buffer overflow detectors ARCHER [Xie et al., 2003] and BOON [Wagner et al., 2000].

4.3 Rules

The rules that define what a security tool should report are just as important, if not more important, than the analysis algorithms and heuristics that the tool implements. The analysis algorithms do the heavy lifting, but the rules call the shots. Analysis algorithms sometimes get lucky and reach the right conclusions for the wrong reasons, but a tool can never report a problem outside its rule set.

Early security tools were sometimes compared simply by counting the number of rules that each tool came packaged with by default. More recent static analysis tools are harder to compare. Rules might work together to

detect an issue, and an individual rule might refer to abstract interfaces or match method names against a regular expression. Just as more code does not always make a better program, more rules do not always make a better static analysis tool.

Code quality tools sometimes infer rules from the code they are analyzing. If a program calls the same method in 100 different locations, and in 99 of those locations it pays attention to the method's return value, there is a decent chance that there is a bug at the single location that does not check the return value. This statistical approach to inferring rules does not work so well for identifying security problems. If a programmer did not understand that a particular construct represents a security risk, the code might uniformly apply the construct incorrectly throughout the program, which would result in a 100% false negative rate given only a statistical approach.

Rules are not just for defining security properties. They're also used to define any program behavior not explicitly included in the program text, such as the behavior of any system or third-party libraries that the program uses. For example, if a Java program uses the `java.util.Hashtable` class, the static analysis tool needs rules that define the behavior of a `Hashtable` object and all its methods. It's a big job to create and maintain a good set of modeling rules for system libraries and popular third-party libraries.

Rule Formats

Good static analysis tools externalize the rules they check so that rules can be added, subtracted, or altered without having to modify the tool itself. The best static analysis tools externalize all the rules they check. In addition to adjusting the out-of-the-box behavior of a tool, an external rules interface enables the end user to add checks for new kinds of defects or to extend existing checks in ways that are specific to the semantics of the program being analyzed.

Specialized Rule Files

Maintaining external files that use a specialized format for describing rules allows the rule format to be tailored to the capabilities of the analysis engine. Example 4.10 shows the RATS rule describing a command injection problem related to the system call `system()`. RATS will report a violation of the rule whenever it sees a call to `system()` where the first argument is not constant. It gives the function name, the argument number for the untrusted buffer (so that it can avoid reporting cases in which the argument is a constant), and the severity associated with a violation of the rule.

Example 4.10 A rule from RATS: calling `system()` is a risk if the first argument is not a string literal.

```
<Vulnerability>
  <Name>system</Name>
  <InputProblem>
    <Arg>1</Arg>
    <Severity>High</Severity>
  </InputProblem>
</Vulnerability>
```

Example 4.11 shows a rule from Fortify Source Code Analysis (SCA). The rule also detects command injection vulnerabilities related to calling `system()`, but this rule fires only if there is a path through the program through which an attacker could control the first argument and if that argument value has not been validated to prevent command injection.

The Fortify rule contains more metadata than the RATS example, including a unique rule identifier and kingdom, category, and subcategory fields. As in the RATS example, it contains a default severity associated with violating the rule. It also contains a link to a textual description of the problem addressed by the rule.

Example 4.11 A rule from Fortify Source Code Analysis. Calling `system()` is a risk if the first argument can be controlled by an attacker and has not been validated.

```
<DataflowSinkRule formatVersion="3.2" language="cpp">
  <MetaInfo><Group name="package">C Core</Group></MetaInfo>
  <RuleID>AA212456-92CD-48E0-A5D5-E74CC26A276F</RuleID>
  <VulnKingdom>Input Validation and Representation</VulnKingdom>
  <VulnCategory>Command Injection</VulnCategory>
  <DefaultSeverity>4.0</DefaultSeverity>
  <Description ref="desc.dataflow.cpp.command_injection"/>
  <Sink>
    <InArguments>0</InArguments>
    <Conditional>
      <Not>
          <TaintFlagSet taintFlag="VALIDATED_COMMAND_INJECTION"/>
      </Not>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
    <FunctionName><Value>system</Value></FunctionName>
  </FunctionIdentifier>
</DataflowSinkRule>
```

Annotations

In some cases, it is preferable to have rules appear directly in the text of the program, in the form of *annotations*. If special rules govern the use of a particular module, putting the rules directly in the module (or the header file for the module) is a good way to make sure that the rules are applied whenever the module is used. Annotations are often more concise than rules that appear in external files because they do not have to explain the context they apply to; an annotation's context is provided by the code around it. For example, instead of having to specify the name of a function, an annotation can simply appear just before the function declaration.

This tight binding to the source code has its disadvantages, too. For example, if the people performing the analysis are not the owners or maintainers of the code, they might not be allowed to add permanent annotations. One might be able to overcome this sort of limitation by creating special source files that contain annotations almost exclusively and using these source files only for the purpose of analysis.

Languages such as Java and C# have a special syntax for annotations. For languages that do not have an annotation syntax, annotations usually take the form of specially formatted comments. Example 4.12 shows an annotation written in the Java Modeling Language (JML). Although Sun has added syntax for annotations as of Java 1.5, annotations for earlier versions of Java must be written in comments. Annotations are useful for more than just static analysis. A number of dynamic analysis tools can also use JML annotations.

Example 4.12 A specification for the `java.io.Reader` method `read()` written in JML. The specification requires the reader to be in a valid state when `read()` is called. It stipulates that a call to `read()` can change the state of the reader, and it ensures that the return value is in the range 1 to 65535.

```
/*@ public normal_behavior
  @   requires    valid;
  @   assignable  state;
  @   ensures     -1 <= \result && \result <= 65535;
  @*/
public int read();
```

Bill Pugh, a professor at the University of Maryland and one of the authors and maintainers of FindBugs, has proposed a set of standard Java 1.5 annotations such as `@NonNull` and `@CheckForNull` that would be useful

for static analysis tools [Pugh, 2006]. The proposal might grow to include annotations for taint propagation, concurrency, and internationalization.

Microsoft has its own version of source annotation: the Microsoft Standard Annotation Language (SAL). SAL works with the static analysis option built into Visual Studio 2005. You can use it to specify the ways a function uses and modifies its parameters, and the relationships that exist between parameters. SAL makes it particularly easy to state that the value of one parameter is used as the buffer size of another parameter, a common occurrence in C. Example 4.13 shows a function prototype annotated with SAL. Quite a few of the commonly used header files that ship with Visual Studio include SAL annotations.

Example 4.13 A function prototype annotated with Microsoft's SAL. The annotation (in bold) indicates that the function will write to the variable `buf` but not read from it, and that the parameter `sz` gives the number of elements in `buf`.

```
int fillBuffer(
  __out_ecount(sz) char* buf,
  size_t sz
);
```

Other Rule Formats

Another approach to rule writing is to expose the static analysis engine's data structures and algorithms programmatically. FindBugs allows programmers to create native plug-ins that the analysis engine loads at runtime. To add a new bug pattern, a programmer writes a new visitor class and drops it in the plug-ins directory. FindBugs instantiates the class and passes it a handle to each class in the program being analyzed. Although a plug-in approach to rule writing is quite flexible, it sets a high bar for authors: A rule writer must understand both the kind of defect he or she wants to detect and the static analysis techniques necessary to detect it.

One of the first static analysis tools we wrote was a checker that looked for testability problems in hardware designs written in Verilog. (Brian wrote it back when he worked at Hewlett-Packard.) It used a scripting language to expose its analysis capabilities. Users could write TCL scripts and call into a set of functions for exploring and manipulating the program representation. This approach requires less expertise on the part of rule writers, but user feedback was largely negative. Users made alterations to the default rule

scripts, and then there was no easy way to update the default rule set. Users wrote scripts that took a long time to execute because they did not understand the computational complexity of the underlying operations they were invoking, and they were not particularly happy with the interface because they were being asked not only to specify the results they wanted to see, but also to formulate the best strategy for achieving them. Just as a database exposes the information it holds through a query language instead of directly exposing its data structures to the user, we believe that a static analysis tool should provide a good abstraction of its capabilities instead of forcing the user to understand how to solve static analysis problems.

The most innovative approach to rule writing that we have seen in recent years is *Program Query Language (PQL)* [Martin et al., 2005]. PQL enables users to describe the sequence of events they want to check for using the syntax of the source language. Example 4.14 gives a PQL query for identifying a simple flavor of SQL injection.

Example 4.14 A PQL query for identifying a simple variety of SQL injection: When a request parameter is used directly as a database query.

```
query simpleSQLInjection()
uses
  object HttpServletRequest r;
  object Connection c;
  object String p;
matches { p = r.getParameter(_); }
replaces c.execute(p)
with Util.CheckedSQL(c, p);
```

Rules for Taint Propagation

Solving taint propagation problems with static analysis requires a variety of different rule types. Because so many security problems can be represented as taint propagation problems, we outline the various taint propagation rule types here:

- **Source** rules define program locations where tainted data enter the system. Functions named `read()` often introduce taint in an obvious manner, but many other functions also introduce taint, including `getenv()`, `getpass()`, and even `gets()`.
- **Sink** rules define program locations that should not receive tainted data. For SQL injection in Java, `Statement.executeQuery()` is a sink. For

buffer overflow in C, assigning to an array is a sink, as is the function `strcpy()`.

- **Pass-through** rules define the way a function manipulates tainted data. For example, a pass-through rule for the `java.lang.String` method `trim()` might explain "if a String `s` is tainted, the return value from calling `s.trim()` is similarly tainted."
- A **cleanse** rule is a form of pass-through rule that removes taint from a variable. Cleanse rules are used to represent input validation functions.
- **Entry-point** rules are similar to source rules, in that they introduce taint into the program, but instead of introducing taint at points in the program where the function is invoked, entry-point functions are invoked by an attacker. The C function `main()` is an entry point, as is any Java method named `doPost()` in an `HttpServlet` object.

To see how the rule types work together to detect a vulnerability, consider Figure 4.8. It shows a source rule, a pass-through rule, and a sink rule working together to detect a command injection vulnerability. A source rule carries the knowledge that `fgets()` taints its first argument (`buf`). Dataflow analysis connects one use of `buf` to the next, at which point a pass-through rule allows the analyzer to move the taint through the call to `strcpy()` and taint `othr`. Dataflow analysis connects one use of `othr` to the next, and finally a sink rule for `system()` reports a command injection vulnerability because `othr` is tainted.

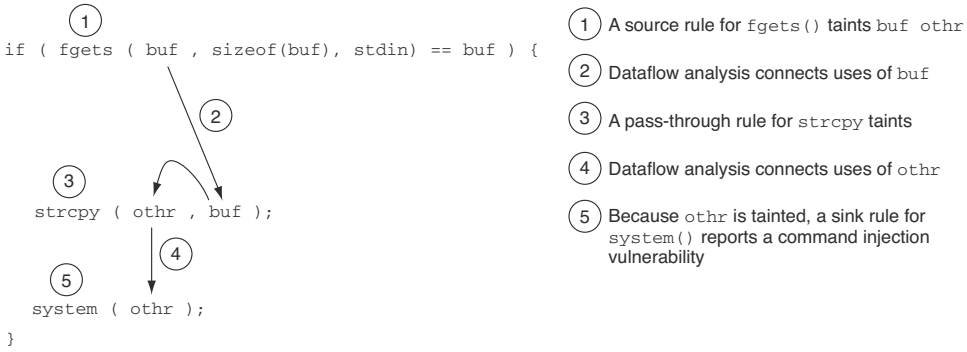

Figure 4.8 Three dataflow rules work together to detect a command injection vulnerability.

In its simplest form, taint is a binary attribute of a piece of data—the value is either tainted or untainted. In reality, input validation problems are not nearly so clear cut. Input can be trusted for some purposes, but not for others. For example, the argument parameters passed to a C program's `main()` function are not trustworthy, but most operating systems guarantee that the strings in the `argv` array will be null-terminated, so it is safe to treat them as strings. To represent the fact that data can be trusted for some purposes but not for others, different varieties of tainted data can be modeled as carriers of different *taint flags*. Taint flags can be applied in a number of different ways.

First, different source rules can introduce data with different taint flags. Data from the network could be marked `FROM_NETWORK`, and data from a configuration file might be marked `FROM_CONFIGURATION`. If these taint flags are carried over into the static analysis output, they allow an auditor to prioritize output based on the source of the untrusted data.

Second, sink functions might be dangerous only when reached by data carrying a certain type of taint. A cross-site scripting sink is vulnerable when it receives arbitrary user-controlled data, but not when it receives only numeric data.

Source, sink, and pass-through rules can manipulate taint in either an additive or a subtractive manner. We have seen successful implementations of both approaches. In the subtractive case, source rules introduce data carrying all the taint flags that might possibly be of concern. Input validation functions are modeled with pass-through rules that strip the appropriate taint flags, given the type of validation they perform. Sink rules check for dangerous operations on tainted data or for tainted data escaping from the application tier (such as passing from business logic to back-end code) and trigger if any of the offending taint flags are still present. In the additive case, source rules introduce data tainted in a generic fashion, and input-validation functions add taint flags based on the kind of validation they perform, such as `VALIDATED_XSS` for a function that validates against cross-site scripting attacks. Sinks fill the same role as in the subtractive case, firing either when a dangerous operation is performed on an argument that does not hold an appropriate set of taint flags or when data leave the application tier without all the necessary taint flags.

Rules in Print

Throughout Part II, "Pervasive Problems," and Part III, "Features and Flavors," we discuss techniques for using static analysis to identify specific security problems in source code. These discussions take the form of

specially formatted callouts labeled "Static Analysis." Many of these sections include a discussion of specific static analysis rules that you can use to solve the problem at hand. For the most part, formats that are easy for a computer to understand, such as the XML rule definition that appears earlier in this chapter, are not ideal for human consumption. For this reason, we introduce a special syntax here for defining rules. This is the rule syntax we use for the remainder of the book.

Configuration Rules

We specify configuration rules for XML documents with XPath expressions. The rule definitions also include a file pattern to control which files the static analysis tool applies the XPath expression to, such as `web.xml` or `*.xml`.

Model Checking Rules

Instead of giving their definitions textually, we present model checking rules using state machine diagrams similar to the one found earlier in this chapter. Each model checking diagram includes an edge labeled "start" that indicates the initial state the rule takes on, and has any number of transitions leading to other states that the analysis algorithm will follow whenever it encounters the code construct associated with the transition.

Structural Rules

We describe structural rules using the special language introduced in the sidebar earlier this chapter. Properties in the language correspond to common properties in source code, and most rules are straightforward to understand without any existing knowledge of the language.

Taint Propagation Rules

Taint propagation rules in the book include a combination of the following elements:

- **Method or function**—Defines the method or function that the rule will match. All aspects of the rule are applied only to code constructs that match this element, which can include special characters, such as the wildcard (`*`) or the logical or operator (`|`).
- **Precondition**—Defines conditions on the taint propagation algorithm's state that must be met for the rule to trigger. Precondition statements typically specify which arguments to a function must not be tainted or which taint flags must or must not be present, so preconditions stand for

sink rules. If the precondition is not met, the rule will trigger. In the case of sink rules, a violation of the precondition results in the static analysis tool reporting an instance of the vulnerability the rule represents.

- **Postcondition**—Describes changes to the taint propagation algorithm's state that occur when a method or function the rule matches is encountered. Postcondition statements typically taint or cleanse certain variables, such as the return value from the function or any of its arguments, and can also include assignment of taint flags to these variables. Postconditions represent source or passthrough information.
- **Severity**—Allows the rule definition to specify the severity of the issues the taint propagation algorithm produces when a sink rule is triggered. In some cases, it is important to be able to differentiate multiple similar results that correspond to the same type of vulnerability.

4.4 Reporting Results

Most of the academic research effort invested in static analysis tools is spent devising clever new approaches to identifying defects. But when the time comes for a tool to be put to work, the way the tool reports results has a major impact on the value the tool provides. Unless you have a lab full of Ph.D. candidates ready to interpret raw analyzer output, the results need to be presented in such a way that the user can make a decision about the correctness and importance of the result, and can take an appropriate corrective action. That action might be a code change, but it might also be an adjustment of the tool.

Tool users tend to use the term *false positive* to refer to anything that might come under the heading "unwanted result." Although that's not the definition we use, we certainly understand the sentiment. From the user's perspective, it doesn't matter how fancy the underlying analysis algorithms are. If you can't make sense of what the tool is telling you, the result is useless. In that sense, bad results can just as easily stem from bad presentation as they can from an analysis mistake.

It is part of the tool's job to present results in such a way that users can divine their potential impact. Simple code navigation features such as jump-to-definition are important. If a static analysis tool can be run as a plug-in inside a programmer's integrated development environment (IDE), everyone wins: The programmer gets a familiar code navigation setup, and the static analysis tool developers don't have to reinvent code browsing.

Auditors need at least three features for managing tool output:

- Grouping and sorting results
- Eliminating unwanted results
- Explaining the significance of results

We use the Fortify audit interface (Audit Workbench) to illustrate these features. Figure 4.9 shows the Audit Workbench main view.

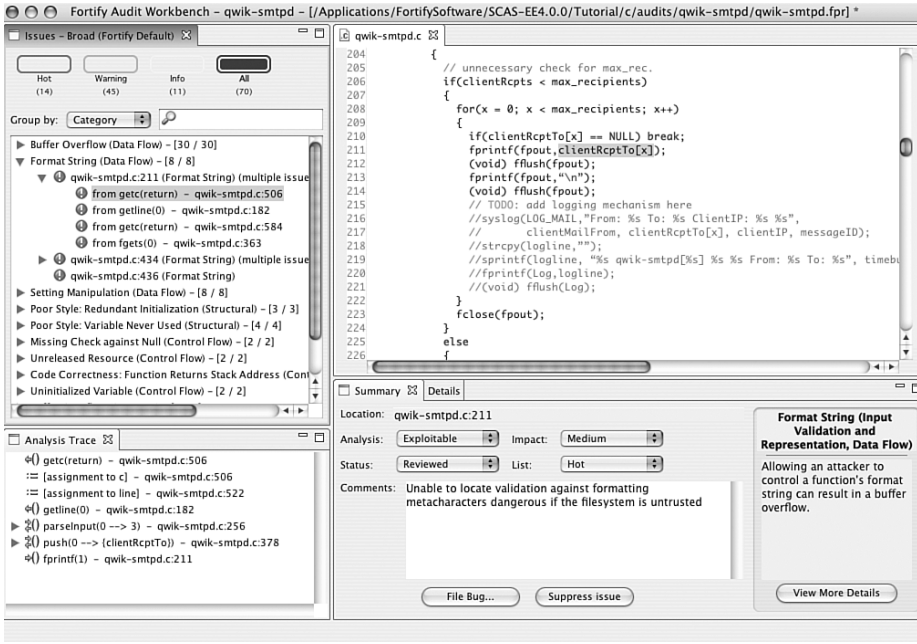

Figure 4.9 The Audit Workbench interface.

Grouping and Sorting Results

If users can group and sort issues in a flexible manner, they can often eliminate large numbers of unwanted results without having to review every issue individually. For example, if the program being analyzed takes some of its input from a trusted file, a user reviewing results will benefit greatly from a means by which to eliminate all results that were generated under the assumption that the file was not trusted.

Because static analysis tools can generate a large number of results, users appreciate having results presented in a ranked order so that the most important results will most likely appear early in the review. Static analysis tools have two dimensions along which they can rank results. *Severity* gives the gravity of the finding, under the assumption that the tool has not made any mistakes. For example, a buffer overflow is usually a more severe security problem than a null pointer dereference. *Confidence* gives an estimate of the likelihood that the finding is correct. A tool that flags every call to `strcpy()` as a potential buffer overflow produces low confidence results. A tool that can postulate a method by which a call to `strcpy()` might be exploited is capable of producing higher confidence results. In general, the more assumptions a tool has to make to arrive at a result, the lower the confidence in the result. To create a ranking, a tool must combine severity and confidence scores for each result. Typically, severity and confidence are collapsed into a simple discrete scale of importance, such as Critical (C), High (H), Medium (M), and Low (L), as shown in Figure 4.10. This gives auditors an easy way to prioritize their work.

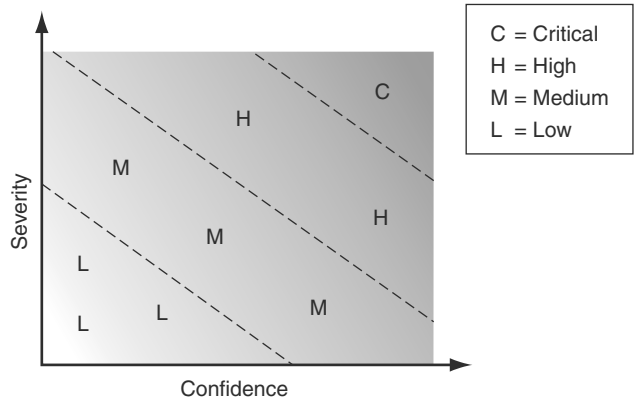

Figure 4.10 Severity and confidence scores are usually collapsed into a simple discrete scale of importance, such as Critical (C), High (H), Medium (M), and Low (L).

Audit Workbench groups results into folders based on a three-tier scale. It calls the folders Hot, Warning, and Info. A fourth folder displays all issues. Figure 4.11 shows the Audit Workbench folder view.

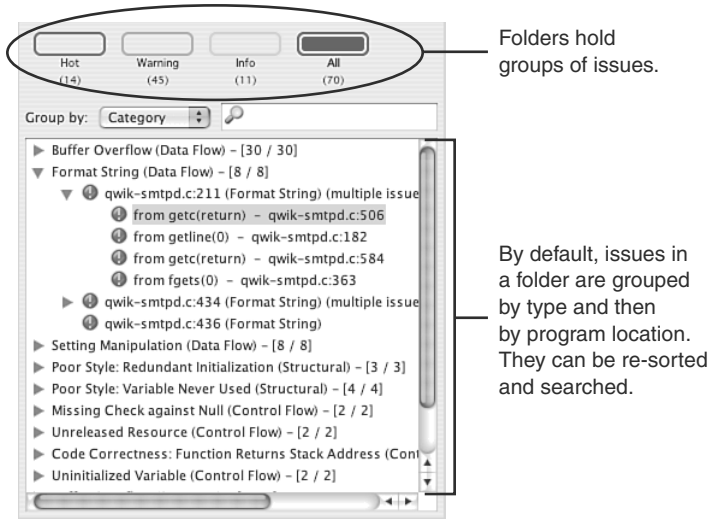

Figure 4.11 Sorting and searching results in Audit Workbench.

Eliminating Unwanted Results

Reviewing unwanted results is no fun, but reviewing the same unwanted results more than once is maddening. All advanced static analysis tools provide mechanisms for suppressing results so that they will not be reported in subsequent analysis runs. If the system is good, suppression information will carry forward to future builds of the same codebase. Similarly, auditors should be able to share and merge suppression information so that multiple people don't need to audit the same issues.

Users should be able to turn off entire categories of warnings, but they also need to be able to eliminate individual errors. Many tools allow results to be suppressed using pragmas or code annotations, but if the person performing the code review does not have permission to modify the code, there needs to be a way to store suppression information outside the code. One possibility is to simply store the filename, line number, and issue type. The problem is that even a small change to the file can cause all the line numbers to shift, thereby invalidating the suppression information. This problem can be lessened by storing a line number as an offset from the beginning of the function it resides in or as an offset from the nearest labeled statement. Another approach, which is especially useful if a result includes a trace

through the program instead of just a single line, is to generate an identifier for the result based on the program constructs that comprise the trace. Good input for generating the identifier includes the names of functions and variables, relevant pieces of the control flow graph, and identifiers for any rules involved in determining the result.

Explaining the Significance of the Results

Good bug reports from human testers include a description of the problem, an explanation of who the problem affects or why it is important, and the steps necessary to reproduce the problem. But even good bug reports are occasionally sent back marked "could not reproduce" or "not an issue." When that happens, the human tester gets a second try at explaining the situation. Static analysis tools don't get a second try, so they have to make an effective argument the first time around. This is particularly difficult because a programmer might not immediately understand the security ramifications of a finding. A scenario that might seem far-fetched to an untrained eye could, in fact, be easy pickings for an attacker. The tool must explain the risk it has identified and the potential impact of an exploit.

Audit Workbench makes its case in two ways. First, if the result is based on tracking tainted data through the program, it presents a dataflow trace that gives the path through the program that an exploit could take. Second, it provides a textual description of the problem in both a short form and a detailed form. Figure 4.12 shows a dataflow trace and a short explanation.

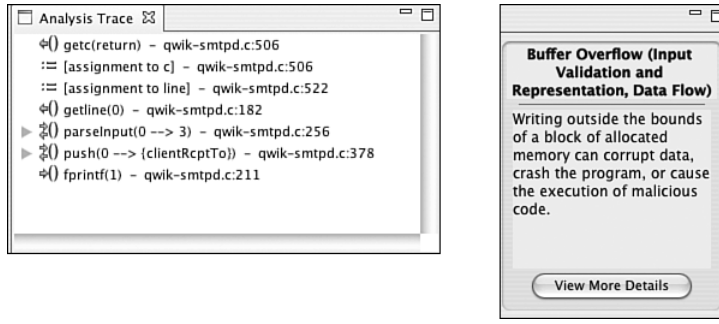

Figure 4.12 Audit Workbench explains a result with a dataflow trace (when available) and a brief explanation.

The detailed explanation is divided into five parts:

- The abstract, a one sentence explanation of the problem
- A description that explains the specific issue in detail and references the specifics of the issue at hand (with code examples)
- Recommendations for how the issue should be fixed (with a different recommendation given depending on the specifics of the issue at hand)
- Auditing tips that explain what a reviewer should do to verify that there is indeed a problem
- References that give motivated reviewers a place to go to read more if they are so inclined

For these two lines of code

```
36 fread(buf, sizeof(buf), FILE);
37 strcpy(ret, buf);
```

Audit Workbench would display the following detailed explanation:

String Termination Error

ABSTRACT

Relying on proper string termination may result in a buffer overflow.

DESCRIPTION

String termination errors occur when:

1. Data enter a program via a function that does not null terminate its output.

   In this case, the data enter at `fread` in `reader.c` at line 36.

2. The data are passed to a function that requires its input to be null terminated.

   In this case, the data are passed to `strcpy` in `reader.c` at line *37*.

Example 1: The following code reads from `cfgfile` and copies the input into `inputbuf` using `strcpy()`. The code mistakenly assumes that `inputbuf` will always contain a null terminator.

```
#define MAXLEN 1024
...
char pathbuf[MAXLEN];
...
read(cfgfile,inputbuf,MAXLEN); //does not null terminate
strcpy(pathbuf,inputbuf); //requires null terminated input
...
```

The code in Example 1 will behave correctly if the data read from `cfgfile` are null terminated on disk as expected. But if an attacker is able to modify this input so that it does not contain the expected null character, the call to `strcpy()` will continue copying from memory until it encounters an arbitrary null character. This will likely overflow the destination buffer and, if the attacker can control the contents of memory immediately following `inputbuf`, can leave the application susceptible to a buffer overflow attack.

Example 2: In the following code, `readlink()` expands the name of a symbolic link stored in the buffer `path` so that the buffer `buf` contains the absolute path of the file referenced by the symbolic link. The length of the resulting value is then calculated using `strlen()`.

```
...
char buf[MAXPATH];
...
readlink(path, buf, MAXPATH);
int length = strlen(buf);
...
```

The code in Example 2 will not behave correctly because the value read into `buf` by `readlink()` will not be null-terminated. In testing, vulnerabilities such as this one might not be caught because the unused contents of `buf` and the memory immediately following it might be null, thereby causing `strlen()` to appear as if it is behaving correctly. However, in the wild, `strlen()` will continue traversing memory until it encounters an arbitrary null character on the stack, which results in a value of `length` that is much larger than the size of `buf` and could cause a buffer overflow in subsequent uses of this value.

Traditionally, strings are represented as a region of memory containing data terminated with a null character. Older string handling methods

frequently rely on this null character to determine the length of the string. If a buffer that does not contain a null terminator is passed to one of these functions, the function will read past the end of the buffer.

Malicious users typically exploit this type of vulnerability by injecting data with unexpected size or content into the application. They might provide the malicious input either directly as input to the program or indirectly by modifying application resources, such as configuration files. If an attacker causes the application to read beyond the bounds of a buffer, the attacker might be able to use a resulting buffer overflow to inject and execute arbitrary code on the system.

RECOMMENDATIONS

As a convention, replace all calls to `strcpy()` and similar functions with their bounded counterparts, such as `strncpy()`. On Windows platforms, consider using functions defined in `strsafe.h`, such as `StringCbCopy()`, which takes a buffer size in bytes, or `StringCchCopy()`, which takes a buffer size in characters. On BSD UNIX systems, `strlcpy()` can be used safely because it behaves the same as `strncpy()`, except that it always null-terminates its destination buffer. On other systems, always replace instances of `strcpy(d, s)` with `strncpy(d, s, SIZE_D)` to check bounds properly and prevent `strncpy()` from overflowing the destination buffer. For example, if `d` is a stack-allocated buffer, `SIZE_D` can be calculated using `sizeof(d)`.

If your security policy forbids the use of `strcpy()`, you can enforce the policy by writing a custom rule to unconditionally flag this function during a source analysis of an application.

Another mechanism for enforcing a security policy that disallows the use of `strcpy()` within a given code base is to include a macro that will cause any use of `strcpy` to generate a compile error:

```
#define strcpy unsafe_strcpy
```

AUDITING TIPS

At first glance, the following code might appear to correctly handle the fact that `readlink()` does not null-terminate its output. But read the code carefully; this is an off-by-one error.

```
...
char buf[MAXPATH];
int size = readlink(path, buf, MAXPATH);
if (size != -1){
    buf[size] = '\0';
    strncpy(filename, buf, MAXPATH);
    length = strlen(filename);
}
...
```

By calling `strlen()`, the programmer relies on a string terminator. The programmer has attempted to explicitly null-terminate the buffer to guarantee that this dependency is always satisfied. The problem with this approach is that it is error prone. In this example, if `readlink()` returns `MAXPATH`, then `buf[size]` will refer to a location outside the buffer; `strncpy()` will fail to null-terminate `filename`, and `strlen()` will return an incorrect (and potentially huge) value.

REFERENCES

[1] M. Howard and D. LeBlanc. *Writing Secure Code, Second Edition.* Microsoft Press, 2003. (Discussion of Microsoft string-manipulation APIs.)

Summary

Major challenges for a static analysis tool include choosing an appropriate representation for the program being analyzed (building a model) that makes good trade-offs between precision and scalability, and choosing algorithms capable of finding the target set of defects. Essential static analysis problems often involve some of the same techniques as compiler optimization problems, including tracking dataflow and control flow. Tracking tainted data through a program is particularly relevant to identifying security defects because so many security problems are a result of trusting untrustworthy input.

Good static analysis tools are rule driven. Rules tell the analysis engine how to model the environment and the effects of library and system calls.

Rules also define the security properties that the tool will check against. Good static analysis tools are extensible—they allow the user to add rules to model new libraries or environments and to check for new security properties.

Ease of use is an often-overlooked but critical component of a static analysis tool. Users need help understanding the results the tool finds and why they are important.