Automated Code Review Tools for Security

Gary McGraw, Cigital

Static analysis identifies many common coding problems automatically, before a program is released.

Computer security has experienced important fundamental changes over the past decade. The most promising developments in security involve arming software developers and architects with the knowledge and tools they need to build more secure software. Among the many security tools available to software practitioners, static analysis tools for automated code review are the most effective. Here's how they work—and why all developers should use them.

THE RISE OF SOFTWARE SECURITY

Traditional approaches to computer security focused almost exclusively on the network; the idea was to keep malicious hackers away from vulnerable machines by placing a barrier between the two. Security vendors introduced the network firewall in the late 1980s as a way of creating such a barrier between a local area network and the Internet. Although firewalls certainly have their place in computer security and have since become ubiquitous, serious security problems persist.

Since the late 1990s, a new paradigm in computer security has evolved—software security (sometimes called application security). Software security is the engineering of software so that it continues to function correctly under malicious attack.

Although software security is relatively young as a discipline, much progress has been made on ways to integrate security best practices into the software development life cycle. Microsoft has helped to spearhead software security through its Trustworthy Computing Initiative and the resulting Secure Development Lifecycle (SDL). My company, Cigital, also has been instrumental in bringing software security to the wider market through thought leadership and professional services.

Numerous software security best practices have emerged in several methodologies, including OWASP (Open Web Application Security Project) CLASP (Comprehensive, Lightweight Application Security Project), Microsoft's SDL, and Cigital's Touchpoints. Figure 1 shows the Cigital Touchpoints as described in *Software Security: Building Security In* (McGraw, 2006). Without question, the top two best practices are code review (with a static analysis tool) and architectural risk analysis.

Among these two best practices, code review with a static analysis tool is the easiest and most straightforward to adopt. There are two reasons for this. First, every software project has code that can be reviewed (they should all have an architecture too, but that's a topic for another article). Second, code review has been partially automated with sophisticated tools.

WHY CODE REVIEW TOOLS?

Many security problems are caused by simple bugs that can be spotted in code. For example, a buffer overflow vulnerability is the common result of misusing various string functions, including strcpy() in C. Using a tool makes sense because code review is boring, difficult, and tedious. Analysts who practice code review often are well familiar with the "get done, go home" phenomenon described in *Building Secure Software: How to Avoid Security Problems the Right Way* (J. Viega and G. McGraw, 2001). It's all too easy to start a review with diligence and care, cross-referencing definitions and variable declarations, and end it by giving function definitions (and sometimes even entire pages) only a cursory glance.

Programmers make little mistakes all the time—a missing semicolon here, an extra parenthesis there. Most of the time, such gaffes are inconsequential; the compiler notes the error, the programmer fixes the code, and the development process continues. This quick cycle of feedback and response stands in sharp contrast to what happens with most security vulnerabilities, which can lie dormant, sometimes for years, before discovery. The longer a vulnerability lies dormant, the more expensive it can be to fix; and, adding insult to injury, the programming community has a long history of repeating the same security-related mistakes.

One problem is that security is not yet a standard part of the programming curriculum. You can't really blame programmers who introduce security problems into their software if nobody ever told them what to avoid or how to build secure software. Another problem is that most programming languages were not designed with security in mind. Unintentional (mis)use of various functions built into these languages leads to common and often exploited vulnerabilities.

Creating simple tools to help look for these problems is an obvious way forward. The promise of static analysis is to identify many common coding problems automatically, before a program is released.

Static analysis tools—also called source code analyzers—examine a program's text without attempting to execute it. Theoretically, these tools can examine either a program's source code or a compiled form of the program to equal benefit, although the problem of decoding the latter can be difficult. I focus on source code analysis here because that's where the most mature technology exists.

Manual auditing is a form of static analysis. This is very time-consuming, and human code auditors must first know what security vulnerabilities look like before they can rigorously examine the code. Static analysis tools compare favorably to manual audits because they're faster, which means they can evaluate programs much more frequently, and they encapsulate security knowledge in a way that doesn't require the tool operator to have the same level of security expertise as a human auditor. Just as a programmer can rely on a compiler to enforce the finer points of language syntax consistently, the operator of a good static analysis tool can successfully apply that tool without being aware of the finer points of security bugs.

Testing for security vulnerabilities is complicated because they often exist in hard-to-reach states or crop up in unusual circumstances. Static analysis tools can peer into more of a program's dark corners with less fuss than dynamic analysis, which requires actually running the code. Static analysis also has the potential to be applied before a program reaches a level of completion at which testing can be meaningfully performed. The earlier that security risks are identified and managed in the software life cycle, the better.

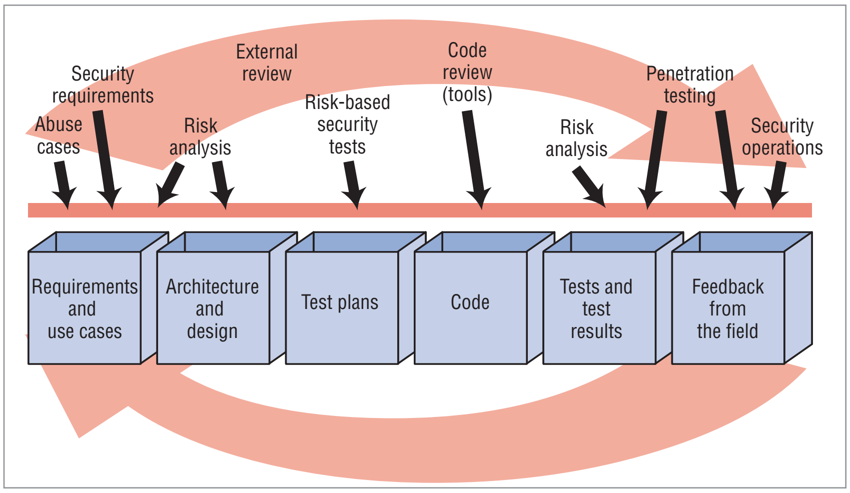


Figure 1. The Cigital Touchpoints methodology. Software security best practices (arrows) applied to various software artifacts (boxes).

A BRIEF HISTORY OF CODE REVIEW TOOLS

The first scanner built to look for security problems in code was Cigital's ITS4 (www.cigital.com/its4). (ITS4 is an acronym for "It's the Software Stupid Security Scanner," a name we invented much to the dismay of our marketing people. That was back in the day when Cigital was called Reliable Software Technologies.) Since ITS4's release in early 2000, the idea of detecting security problems by looking over source code with a tool has come of age. Much better approaches exist and are being rapidly commercialized.

ITS4 and its counterparts RATS (no longer available) and Flawfinder (www.dwheeler.com/flawfinder) are extremely simple—the tools scan through a file (lexically) looking for syntactic matches based on several simple "rules" that might indicate possible security vulnerabilities. One such rule might be "use of strcpy() should be avoided," which can be applied by looking through the software for the pattern "strcpy" and alerting the user when and where it is found. This is obviously a simple-minded approach, often referred to with the derogatory label "glorified grep."

The best thing about ITS4 and similar tools was that creating them involved gathering and publishing a preliminary set of software security rules all in one place. When we released ITS4 as an open source tool, our hope was that the world would participate in helping to gather and improve the rule set. Although more than 15,000 people downloaded ITS4 in its first year, we never received even one rule to add to its knowledge base. The world did not end, however, and several prominent commercial efforts to build up and evolve rule sets were undertaken.

ITS4 and its counterparts were never intended to be "push the button, see the bug" kinds of tools. The basic idea was instead to turn an impossible problem (remembering all those rules while doing manual code review) into a really hard one (figuring out whether the things the tool flagged matter or not). Simple tools like ITS4 help carry out a

References for Secure Code Review

There are numerous good resources for readers interested in learning more about automated code review and other software security technologies. Here are the top five.

- R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed., John Wiley & Sons, 2008; www.cl.cam.ac.uk/~rja14/book.html.
- B. Chess and J. West, *Secure Programming with Static Analysis*, Addison-Wesley, 2007; http://buildingsecurityin.com.
- M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed., Microsoft Press, 2003; http://blogs.msdn.com/michael_howard.
- G. McGraw, *Software Security: Building Security In*, Addison-Wesley, 2006; www.swsec.com.
- J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001; www.buildingsecuresoftware.com.

source code security review, but they certainly don't do it for you. The same can be said for modern tools, although they definitely make things much easier than the first-generation tools did.

Probably the simplest and most straightforward approach to static analysis is the Unix utility grep—the same functionality implemented in the earliest tools such as ITS4. Armed with a list of good search strings, grep can reveal quite a lot about a code base. The downside is that grep is rather lo-fi because it doesn't understand anything about the files it scans. Comments, string literals, declarations, and function calls are all just part of a stream of characters to be matched against.

You might be amused to note that a grep through code for words like "bug," "XXX," "Fix," "Here," and best of all, "Assume" often reveals interesting and relevant tidbits. Any good security source code review should start with that.

Better fidelity requires taking into account the lexical rules that govern the programming language being analyzed. By doing this, a tool can distinguish between a vulnerable function call

```
gets(&buf);
```

a comment

```
/* never ever call gets */
```

and an unrelated identifier

```
int begetsNextChild = 0;
```

Basic lexical analysis is the approach taken by early static analysis tools, including ITS4, Flawfinder, and RATS, all of which preprocess and tokenize source files (the same first steps a compiler would take) and then match the resulting token stream against a library of vulnerable constructs.

While lexical analysis tools are certainly a step up from grep, they produce a hefty number of false positives because they make no effort to account for the target code's semantics. A stream of tokens is better than a stream of characters, but it's still a long way from understanding how a program will behave when it executes. Although some security defect signatures are so strong that they don't require semantic interpretation to be identified accurately, most are not so straightforward.

To increase precision, a static analysis tool must leverage more compiler technology. By building an abstract syntax tree (AST) from source code, such a tool could take into account the basic semantics of the program being evaluated.

Armed with an AST, the next decision to make involves the scope of the analysis. Local analysis examines the program one function at a time and doesn't consider relationships between functions. Module-level analysis considers one class or compilation unit at a time, so it takes into account relationships between functions in the same module and considers properties that apply to classes, but it doesn't analyze calls between modules. Global analysis involves analyzing the entire program, so it takes into account all relationships between functions.

The scope of the analysis also determines the amount of context the tool considers. More context is better when it comes to reducing false positives, but it can lead to a huge amount of computation to perform.

MODERN CODE REVIEW TOOLS

In 2004 and 2005, several start-ups were formed to address the software security space. Many of these vendors have built and are selling basic source code analysis tools. Major vendors in the space include

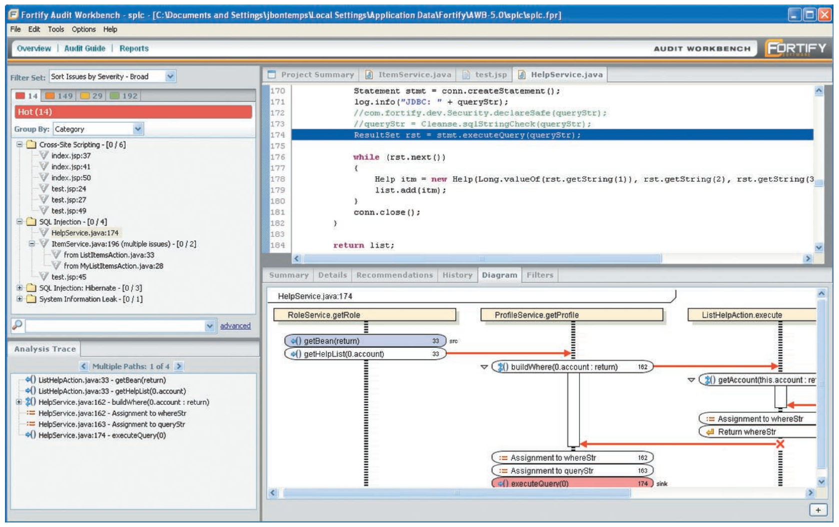
- Coverity (www.coverity.com),
- Fortify (www.fortify.com), and
- Ounce Labs (www.ouncelabs.com).

These vendors take a similar technological approach, but some are more academically inclined than others. By basing their tools on compiler technology, these vendors have upped the level of sophistication far beyond the early almost unusable tools like ITS4.

A critical feature that currently serves as an important differentiator in the static analysis tools market is the kind of knowledge (the rule set) that a tool enforces. The importance of a good rule set can't be overestimated.

One reason to use a source code analysis tool is that manual review is costly and time consuming. Manual review is such a pain that reviewers regularly suffer from the "get done, go home" phenomenon—starting strong and ending with a sputter. An automated tool can begin to check every line of code whenever a build is complete, allowing development shops to get on with the business of building software.

Integrating a source code analyzer into a development life cycle can be painless and easy. As long as your code builds, you should be able to run a modern analysis. Working through the results remains a challenge, but it is nowhere near as much trouble as painstakingly checking every line of code by hand. Figure 2 shows a screenshot from Fortify's SCA product, demonstrating how results are presented in a commercial tool.



Fortify Audit Workbench - splc - [C:\Documents and Settings\jbontemps\Local Settings\Application Data\Fortify\AWB-5.0\splc\splc.fpr]
File Edit Tools Options Help
Overview | Audit Guide | Reports
AUDIT WORKBENCH
FORTIFY
Project Summary
ItemService.java
test.jsp
HelpService.java
Summary
Details
Recommendations
History
Diagram
Filters
Analysis Trace
advanced
HelpService.java:174
RoleService.getRole
ProfileService.getProfile
ListHelpAction.execute
getBean(return) 33 src
getHelpList(0.account) 33
buildWhere(0.account : return) 162
getAccount(this.account : re
Assignment to whereStr
Return whereStr
Assignment to whereStr 162
Assignment to queryStr 163
executeQuery(0) 174 sink

Filter Set: Sort Issues by Severity - Broad

🟥 14 🟧 149 🟨 29 🟩 192

Hot (14)

Group By: Category

- Cross-Site Scripting - [0 / 6]
  - index.jsp:37
  - index.jsp:41
  - index.jsp:50
  - test.jsp:24
  - test.jsp:27
  - test.jsp:49
- SQL Injection - [0 / 4]
  - HelpService.java:174
  - ItemService.java:196 (multiple issues) - [0 / 2]
    - from ListItemsAction.java:33
    - from MyListItemsAction.java:28
  - test.jsp:45
- SQL Injection: Hibernate - [0 / 3]
- System Information Leak - [0 / 1]

Multiple Paths: 1 of 4

```
ListHelpAction.java:33 - getBean(return)
ListHelpAction.java:33 - getHelpList(0.account)
HelpService.java:162 - buildWhere(0.account : return)
HelpService.java:162 - Assignment to whereStr
HelpService.java:163 - Assignment to queryStr
HelpService.java:174 - executeQuery(0)
```

```
            Statement stmt = conn.createStatement();
            log.info("JDBC: " + queryStr);
            //com.fortify.dev.Security.declareSafe(queryStr);
            //queryStr = Cleanse.sqlStringCheck(queryStr);
            ResultSet rst = stmt.executeQuery(queryStr);

            while (rst.next())
            {
                Help itm = new Help(Long.valueOf(rst.getString(1)), rst.getString(2), rst.getString(3
                list.add(itm);
            }
            conn.close();
        }

        return list;
```

Figure 2. Screenshot from Fortify SCA. Browsable results are displayed in commercial source code analysis tools.

Modern approaches to static analysis can now process on the order of millions of lines of code quickly and efficiently. Although a complete review certainly requires an analyst with a clue, the process of looking through the results of a tool and thinking through potential vulnerabilities beats looking through everything. A time savings of several hundred percent is feasible.

Modern tools have several built-in time-saving mechanisms. The first is the knowledge encapsulated in a tool. Keeping a burgeoning list of all known security problems found in a language like C (several hundred) in your head while attempting by hand to trace control flow, data flow, and an explosion of states is extremely difficult. Having a tool that remembers security problems (and can easily be expanded to cover new problems) is a huge help.

The second time-saving mechanism involves automatically tracking control flow, call chains, and data flow. Although commercial tools make tradeoffs when it comes to soundness, they certainly make the laborious process of control and data-flow analysis much easier. For example, a decent tool can locate a potential strcpy() vulnerability on a given line, present the result in a results browser, and arm the user with an easy and automated way of determining (through control flow call chains and data-flow structures) whether the possible vulnerability is real. Although tools are getting better at determining this kind of thing, they are not perfect.

The root cause of many security problems can be found in the source code and configuration files of common software applications—especially custom apps that you write yourself. Problems are seeded when vulnerable code is written into the system—undeniably the most efficient and effective time to remove them.

The way forward is to use automated tools and processes that systematically and comprehensively target the root cause of security issues in source code. Instead of sorting through millions of lines of code looking for vulnerabilities, a developer using an advanced software security tool that returns a small set of potential vulnerabilities can pinpoint actual vulnerabilities in seconds—precisely the same vulnerabilities that would take a malicious hacker or manual code reviewer weeks or even months to find. Of course, most bad guys know this and will use these kinds of tools themselves!

ARM THE DEVELOPERS

Good static analysis tools must be easy to use, even for non-security people. This means that the results from these tools must be understandable to normal developers who might not know much about security. In the end, source code analysis tools serve to educate their users about good programming practice. Good static checkers can help their users spot and eradicate common security bugs. This is especially important for languages such as C or C++, for which a very large corpus of rules already exists.

Static analysis for security should be applied regularly as part of any modern software development process. Automated code review for security uses straightforward technology to help solve an important hard problem—identifying security bugs in source code. As one of the top two best practices for software security, static code review is moving into widespread use.

Acknowledgment

Some portions of this article reprinted by permission from *Software Security: Building Security In*, Addison-Wesley, 2006.

Gary McGraw is the CTO of Cigital, a software security firm. Contact him through www.cigital.com/~gem.