Static Analysis for Security

Brian Chess
Fortify Software

Gary McGraw
Cigital

All software projects are guaranteed to have one artifact in common—source code. Together with architectural risk analysis,[1] code review for security ranks very high on the list of software security best practices (see Figure 1).[2] Here, we'll look at how to automate source-code security analysis with static analysis tools.

Since ITS4's release in early 2000 (www.cigital.com/its4/), the idea of detecting security problems through source code has come of age. ITS4 is extremely simple—the tool basically scans through a file looking for syntactic matches based on several simple "rules" that might indicate possible security vulnerabilities (for example, use of `strcpy()` should be avoided). Much better approaches exist.

Catching implementation bugs early

Programmers make little mistakes all the time—a missing semicolon here, an extra parenthesis there. Most of the time, these 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. The promise of static analysis is to identify many common coding problems automatically before a program is released.

Static analysis tools examine the text of a program statically, without attempting to execute it. Theoretically, they 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. We'll focus on source code analysis here because that's where the most mature technology exists.

Manual auditing, a form of static analysis, is very time-consuming, and to do it effectively, 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 consistently enforce the finer points of language syntax, 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 by the fact that 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.

Aim for good, not perfect

Static analysis can't solve all your security problems. For starters, static analysis tools look for a fixed set of patterns, or rules, in the code. Although more advanced tools allow new rules to be added over time, if a rule hasn't been written yet to find a particular problem, the tool will never find that problem. When it comes to security, what you don't know is likely to hurt you, so beware of any tool that says something like, "zero defects found, your program is now secure." The appropriate output is, "sorry, couldn't find any more bugs."

A static analysis tool's output still requires human evaluation. There's no way for a tool to know exactly which problems are more or less important to you automatically, so there's no way to avoid trawling through the output and making a judgment call about which issues should be fixed and which ones represent an acceptable level of risk. Knowledgeable people still need to get a program's design right to avoid any flaws—although static analysis tools can find bugs in the nitty-gritty details, they can't critique design. Don't expect any tool to tell you, "I

see you're implementing a funds transfer application. You should tighten up the user password requirements."

Finally, there's Rice's theorem, which says (in essence) that any nontrivial question you care to ask about a program can be reduced to the halting problem. In other words, static analysis problems are undecidable in the worst case. The practical ramifications of Rice's theorem are that all static analysis tools are forced to make approximations and that these approximations lead to less-than-perfect output. A tool can also produce *false negatives* (the program contains bugs that the tool doesn't report) or *false positives* (the tool reports bugs that the program doesn't contain). False positives cause immediate grief to any analyst who has to sift through them, but false negatives are much more dangerous because they lead to a false sense of security. A tool is sound if, for a given set of assumptions, it produces no false negatives, but the down side to always erring on the side of caution is a potentially debilitating number of false positives. The static analysis crowd jokes that too high a percentage of false positives leads to 100 percent false negatives because that's what you get when people stop using a tool. A tool is *unsound* if it tries to reduce false positives at the cost of sometimes letting a false negative slip by.

Approaches to static analysis

Probably the simplest and most straightforward approach to static analysis is the Unix utility `grep`. Armed with a list of good search strings, `grep` can reveal quite a lot about a code base. The down side 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.

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 innocent and unrelated identifier

```
int begetsNextChild = 0;
```

Basic lexical analysis is the approach taken by early static analysis tools, including ITS4, FlawFinder (www.dwheeler.com/flawfinder/), and RATS (www.securesoftware.com), 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. Earlier, Matt Bishop and Mike Dilger built a special-purpose lexical analysis tool specifically for the purpose of identifying time-of-check to time-of-use (TOCTOU) flaws.[3]

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 can take into account the basic semantics of the program being evaluated.

Armed with ASTs, the next decision to make is 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.

Researchers have explored many methods for making sense of program semantics. Some are sound, some aren't; some are built to detect specific classes of bugs, while others are flexible enough to read definitions for what they're supposed to

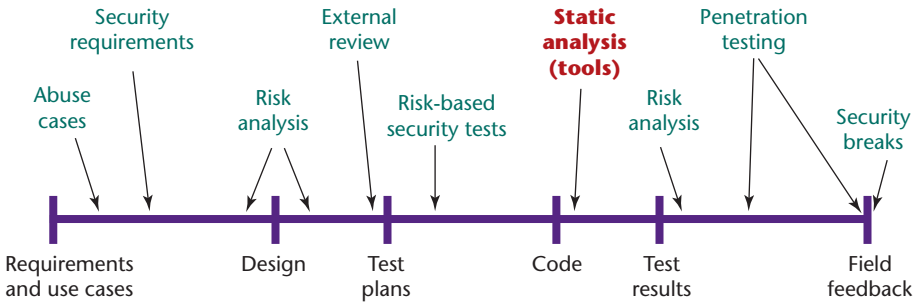

Figure 1. The software development life cycle. Throughout this series, we'll focus on specific parts of the cycle; here, we're examining static analysis.

detect. Let's review some of the most recent tools:

- *BOON* applies integer range analysis to determine whether a C program can index an array outside its bounds.[4] While 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, which detects format string vulnerabilities in C programs.[5] CQual requires a programmer to annotate a few variables as either tainted or untainted and then uses type inference rules (along with pre-annotated system libraries) to propagate the qualifiers. Once the qualifiers are propagated, the system can detect format string vulnerabilities by type checking.
- The *xg++* tool uses a template-driven compiler extension to attack the problem of finding kernel vulnerabilities in the Linux and OpenBSD.[6] It looks for locations where the kernel uses data from an untrusted source without checking it 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.
- The *Eau Claire* tool uses a theorem prover to create a general specification-checking framework for C programs.[7] It can help find common security problems like buffer overflows, file access race conditions, and format string bugs. Developers can use specifications to ensure that function implementations behave as expected.
- *MOPS* takes a model-checking approach to look for violations of temporal safety properties.[8] Developers can model their own safety properties, and some have used the tool to check for privilege management errors, incorrect construction of chroot jails, file access race conditions, and ill-conceived temporary file schemes.
- *Splint* extends the lint concept into the security realm.[9] By adding annotations, developers can enable the tool 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 pre- and postconditions.

Good static checkers can help spot and eradicate common security bugs.

Many 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),[10] model checkers such as SLAM and BLAST (which use predicate abstraction to examine program safety properties),[11,12] and FindBugs (a lightweight checker with a good reputation for unearthing common errors in Java programs).[13]

Several commercial tool vendors are starting to address the need for static analysis, moving some of the approaches touched on here into the mainstream.

Good static analysis tools must be easy to use, even for non-security people. This means that their results must be understandable to normal developers who might not know much about security and that they educate their users about good programming practice. Another critical feature is the kind of knowledge (the rule set) the tool enforces. The importance of a good rule set can't be overestimated.

In the end, good static checkers can help spot and eradicate common security bugs. This is especially important for languages such as C, for which a very large corpus of rules already exists. Static analysis for security should be applied regularly as part of any modern development process. □

References

1. D. Verndon and G. McGraw. "Risk Analysis in Software Design," *IEEE Security & Privacy,* vol. 2, no. 5, 2004, pp. 79–84.
2. G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80–83.
3. M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, no. 2, 1996, pp. 131–152.
4. D. Wagner et al., "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. 7th Network and Distributed System Security Symp.* (NDSS2000), Internet Soc., 2000, pp. 3–17.
5. J. Foster, T. Terauchi, and A. Aiken, "Flow-Sensitive Type Qualifiers," *Proc. ACM Conf. Programming Language Design and Implementation* (PLDI2002), ACM Press, 2002, pp. 1–12.
6. K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 2002, pp. 131–147.
7. B. Chess, "Improving Computer Security using Extended Static Checking," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 2002, pp. 118–130.
8. H. Chen and D. Wagner, "MOPS: An Infrastructure for Examining Security Properties of Software,"

Proc. 9th ACM Conf. Computer and Communications Security (CCS2002), ACM Press, 2002, pp. 235–244.
9. D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," *Proc. 10th Usenix Security Symp.* (USENIX'01), Usenix Assoc., 2001, pp. 177–189.
10. M. Das, S. Lerner, and M. Seigle, "ESP: Path-Sensitive Program Verification in Polynomial Time," *Proc. ACM Conf. Programming Language Design and Implementation* (PLDI2002), ACM Press, 2002, pp. 57–68.
11. T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proc. 8th Int'l SPIN Workshop on Model Checking of Software*, LNCS 2057, Springer-Verlag, 2001, pp. 103–122.
12. T.A. Henzinger et al., "Software Verification with Blast," *Proc. 10th Int'l Workshop Model Checking of Software*, LNCS 2648, Springer-Verlag, 2003, pp. 235–239.
13. D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," to appear in *Companion of the 19th Ann. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, 2004.

Brian Chess *is chief scientist at Fortify Software. His technical interests include static analysis, defect modeling, and Boolean satisfiability. He received a PhD in computer engineering from the University of California, Santa Cruz. Contact him at brian@fortifysoftware.com.*

Gary McGraw *is chief technology officer of Cigital. His real-world experience is grounded in years of consulting with major corporations and software producers. McGraw is the coauthor of* Exploiting Software *(Addison-Wesley, 2004),* Building Secure Software *(Addison-Wesley, 2001),* Java Security *(John Wiley & Sons, 1996), and four other books. He has a BA in philosophy from the University of Virginia and a dual PhD in computer science and cognitive science from Indiana University. Contact him at gem@cigital.com.*