Statically Scanning Java Code:

Finding Security Vulnerabilities

The source code scanning tool Jslint helps programmers automatically utilize existing security knowledge. The tool identifies insecure coding practices, by scanning for common problems, to prevent bugs familiar to the security community.

John Viega and Gary McGraw, *Cigital*
Tom Mutdosch, *Rochester Institute of Technology*
Edward W. Felten, *Princeton University*

Although there is a vast amount of literature on how to avoid and detect security vulnerabilities, there are still few practical solutions codified into tools. Both developers and users require some degree of assurance as to the presence or absence of known security vulnerabilities. To address these concerns, we have designed a prototype tool, Jslint, that statically scans Java source code looking for potentially insecure coding practices. Automated source code scanning tools can help programmers easily prevent some types of bugs.

In developing Jslint, we applied the software security assurance methodology we use while working on real-world systems,[1] the main purpose of which is to identify and correct potential security problems during the software development process as opposed to patching broken systems in the field. Like many useful technology applications, our approach involves a mix of art and engineering. The steps are straightforward:

1. Design a system with security in mind.
2. Analyze the system in light of known and anticipated risks.
3. Rank the risks according to their severity.
4. Test the risks.
5. Cycle broken systems back through the design process.

Our goal is to minimize the unfortunately pervasive "penetrate and patch" approach and avoid desperately trying to fix problems that attackers are actively exploiting. In simple economic terms, finding and removing bugs in a software system before its release is orders of magnitude cheaper and more effective than trying to fix systems after release.

Our methodology relies on security expertise during risk analysis and ranking (Steps 2 and 3). However, most software developers are unfamiliar with software security techniques and attacks. (This might explain why buffer overflow attacks[2]—a well-understood and preventable class of problems—accounted for over 50% of security alerts in 1998 reported by the CERT Coordination Center, a major reporting center for Internet security problems that Carnegie Mellon University operates.[3])

0740-7459/00/$10.00 © 2000 IEEE

Without extensive knowledge of known security problems and attack scenarios, applying Step 2 of our methodology will be ineffective. It is possible to ferret out and fix security problems in the code, but analyzing code by hand is tedious and error-prone (although studies have shown that inspection is effective in the field[4]). Our automated code-scanning tool addresses this tediousness and provides a repository for the expert knowledge once found only in a security practitioner's head. (See the "Related Work" sidebar for other work in this area.)

Securing Java by Scanning Code

We picked Java for our automated tool instead of C or C++ for two reasons. First, we developed 12 rules for writing more secure Java code,[5] which are mostly verifiable by static code scanning. Second, Java makes an interesting case study, because it was specifically designed with an eye toward security. Plenty of information on C and C++ vulnerabilities exist, and we have begun to build similar technologies for these languages.[6]

Java's approach to security includes modern memory encapsulation techniques and sophisticated access control mechanisms built on the *stack-inspection technique*.[7] Nevertheless, the Java platform is a complex system, and complex systems often fail in interesting ways. Practitioners have discovered several serious security problems in various versions of the Java platform, starting in 1996 with JDK 1.0 up through the present day.[5,8,9] For example, a flaw in Netscape's Java implementation that let an attacker remotely download files from the browser's machine through a Java applet was revealed in August 2000.

Developers must address two problems: secure the Java platform and write code that runs on the platform so that it does not introduce new holes. We compiled our 12 rules from security flaws found both in the Java platform (part of which is written in Java) and in the code that runs on it.

Implementations of the Java platform expose a user to three classes of risks:

- A JVM flaw might allow a malicious Java program to breach the JVM's security enforcement.
- The user might mistakenly grant powerful privileges to code that turns out to be malicious.
- The user might grant powerful privileges to code that is well intentioned but contains bugs that other malicious code can exploit.

Securing the Java platform best addresses the first risk, only the user's good judgment can prevent the second, and Jslint focuses on the third. Developers can use our tool on parts of the platform, Java applications, or applet code so that their software is less likely to contain security-critical bugs.

12 Rules for More Secure Java Code

Combating poor coding practices is a hard problem, because there are many subtle ways in which a developer can accidentally introduce risks. We created 12 rules for developing more secure Java programs based on our experience with insecure Java code.[5,10] Together, the 12 rules can help a developer overcome many common problems found in Java programs. Our Java security scanner checks for

Related Work

Security experts have long proposed building simple source code scanners to look for simple patterns that attackers can exploit. To date, several limited prototypes of such systems for C and C++ exist, most notably the Race Condition Scanner[1] and the tool `slint`.

The Race Condition Scanner effectively detects some race condition vulnerabilities but nothing else. Its current implementation relies completely on simple pattern matching (using `grep`), instead of full-fledged language parsing. Unfortunately, the pattern matching is primitive enough that it does not find all the possible race conditions.[1] In addition, this system produces many false positives, because it relies on simple pattern matching.[1]

The hacker group formerly known as 10pht (www.10pht.com) is developing the general-purpose security scanner `slint`. It has the same limitations as the Race Condition Scanner, but it scans for over 100 distinct potential problems in C and C++ code, instead of the race conditions the Race Condition Scanner can check.

The Black Hat community—all hackers with malicious intent—has developed other, more limited tools that scan for buffer overflow attacks. These tools also result in an alarming number of false positives, although determined crackers might not mind. False positives are so frequent with these tools that they often flag commented remarks mentioning names of functions known to be susceptible to attack as potential vulnerabilities.

More recently, Cigital released ITS4, a scanner for C and C++, to the public.[2] It is similar in functionality to `slint` but is slightly more accurate. Researchers have also started looking at how to perform more accurate static analysis, particularly for the buffer overflow problem in the C language.[3]

References

1. M. Bishop and M. Dilger, *Checking for Race Conditions in Unix File Access*, Tech. Report 95-9, Dept. of Computer Science, Univ. of California at Davis, 1995.
2. J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code," to be published in *Proc. Ann. Computer Security Applications Conf. 2000*, Dec. 2000.
3. D. Wagner et al., "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Network and Distributed Systems Security Symp.* (NDSS 2000), 2000.

The goal is not to guarantee security but to eliminate certain kinds of security attacks of which the average programmer is likely unaware.

violations of these rules.

Following these rules certainly does not guarantee that the resulting code is secure. It is always possible to introduce security holes that no scanner could hope to detect. In fact, it is easy to write insecure code that follows these rules. The goal is not to guarantee security but to eliminate certain kinds of security attacks of which the average programmer is likely unaware. If a programmer follows these rules, those kinds of attacks will be impossible, but other kinds of attacks will still be possible.

1. Don't Depend on Initialization

Most Java developers think that there is no way to allocate an object without running a constructor, but there are several ways to allocate uninitialized objects.

2. Limit Access to Your Classes, Methods, and Variables

Every class, method, and variable that is not private provides a potential entry point for an attacker. By default, you should make everything private. Make something public only if there is a good reason and document that reason.

3. Make Everything Final by Default, Unless There Is a Good Reason Not to Do So

In Java, it is possible to make methods and classes final, meaning that they cannot be overridden. For example, a final class cannot be inherited. If a class or method isn't final, an attacker can extend it in a dangerous and unforeseen way. You should make something nonfinal only if there is a good reason and document that reason.

4. Don't Depend on Package Scope

A Java package is a collection of classes. Classes, methods, and variables that you do not explicitly label public, private, or protected are accessible within the same package. You should not rely on package-level access for security. Java packages are not closed (meaning that new elements can be added to them, even at program runtime). As a result, an attacker can potentially introduce a new class inside your package and use this new class to access the things you thought you hid. (A few packages, such as `java.lang`, are closed by default, and a few JVMs let you close your own packages, but you are better off assuming that packages are not closed.)

5. Do Not Use Inner Classes

In Java, it is possible to define inner classes, which are classes nested inside other classes. Some Java language books say that inner classes can only be accessed by the outer classes that enclose them, but this is false. Java byte code has no concept of inner classes, only regular classes. Consequently, the compiler translates inner classes into ordinary classes that happen to be accessible to any code in the same package, and Rule 4 says you should not depend on package scope for protection.

Worse yet, an inner class can access private variables of the containing class. Because the Java protection mechanism does not let the programmer restrict access to a single class, it must grant access to the entire package. Fortunately, the only variables that are exposed in such a way are those actually used by an inner class (at least in most implementations). In addition, a distinction is made between variables that are read by an inner class and those that are written. If an inner class reads a variable, any class in the package can then read that variable. If an inner class writes to a variable, so can any other class in the package.

6. Avoid Signing Your Code

Code signing is a cryptography-based authentication mechanism. The developer digitally signs code using a secret cryptographic key, thereby vouching for its authenticity, and end users can validate that the signature is authentic—that is, the user can prove that the secret key associated with an identity was used to construct the signature. The user can then choose whether or not to run the code based on how much the signing identity is trusted. Code that is not signed will run without any special privileges. Code without special privileges is much less likely to do damage.

Of course, some of your code might have to acquire and use privileges to perform some dangerous operation. You should work hard to minimize the amount of privileged code and audit the privileged code more carefully than the rest.

7. If You Must Sign Your Code, Put it All in One Archive File

This rule prevents attackers from carry-

ing out a mix-and-match attack in which they construct a new applet or library that links some of your signed classes together with malicious classes or links together signed classes that you never meant to be used together. These attacks force trusted code to be used in untrusted ways. By signing a group of classes together, you make this harder.

Existing code-signing systems do an inadequate job of preventing mix-and-match attacks, so this rule cannot prevent such attacks completely. However, using a single archive can't hurt.

8. Make Your Classes Uncloneable

Java's object cloning mechanism lets the programmer make exact duplicates of objects that have already been instantiated in a running program. Because these objects are copies of objects that have already been initialized, constructors are generally not called in the cloned object. This mechanism can let an attacker manufacture new instances of classes you define, without executing any of your constructors. If your class is not cloneable, the attacker can define a subclass of your class and make the subclass implement `java.lang.Cloneable`. This lets the attacker make new instances of your class by copying the memory images of existing objects. This is often an unacceptable way to make a new object.

Rather than worry about this problem, you are better off making your objects uncloneable. If you want your class to be cloneable, and you've considered the consequences of that choice, then you can protect yourself by defining a clone method and making it final. If you're relying on a nonfinal clone method in one of your superclasses, then override the method to make it final or make your entire class final.

9. Make Your Classes Unserializeable

Java's serialization mechanism lets the programmer save entire objects to a storage mechanism such as a disc, database, or string. The mechanism also lets classes be restored from saved information later, perhaps from the same application after it has stopped and restarted or from another application. Saving an object's state in Java is *serialization*; restoring its state is *deserialization*. Serialization is dangerous because it lets adversaries access your objects' internal state. Adversaries can serialize one of your objects into a byte array that can be read, which lets them inspect your object's full internal state, including any fields you marked private and the internal state of any objects you reference.

10. Make Your Classes Undeserializeable

This rule is even more important than Rule 9. Even if your class is not serializeable, it might still be deserializeable. An adversary can create a sequence of bytes that happens to deserialize to an instance of your class. This is dangerous, because you do not have control over what state the deserialized object is in. You can think of deserialization as another kind of public constructor for your object. Unfortunately, it is a kind of constructor that is difficult for you to control.

11. Don't Compare Classes by Name

Sometimes you want to compare the classes of two objects to see whether they are the same or whether an object has a particular class. When you do this, you must be aware that there can be multiple classes with the same name in a JVM. A better way is to compare class objects for equality directly.

12. Secrets Stored in Your Code Won't Protect You

Secrets you store in your code—for example, cryptographic keys in the code for your application or library—are completely accessible to anyone who runs your code. There is nothing to stop a malicious programmer or virtual machine from looking inside your code and learning its secrets.

Code obfuscation is another way to store a secret in your code—in obfuscation, the secret is simply the algorithm your code uses. Although there is not much harm in using an obfuscator, it won't provide much protection. There is no real evidence that you can obfuscate Java source code or byte code so that a dedicated adversary with good tools cannot reverse it.

The Jslint Security Scanner

Jslint is a static source code scanner that uncovers the problems in our 12 rules and can automatically repair any infringement of eight of the rules (Rules 1 through 4 and 8 through 11). (We do not check for Rule 12, because it is essentially impossible for

There is nothing to stop a malicious programmer or virtual machine from looking inside your code and learning its secrets.

Static Code Scanning

Complexity theory tells us that a code-scanning tool is necessarily incomplete, because it will not detect every possible security hole. Fortunately, this theoretical result does not keep us from detecting and correcting many classes of security problems often found in the field.

Several taxonomies of security vulnerabilities exist.[1,2] Examining known security problems, observing their common characteristics, and constructing categories creates these taxonomies. We can statically detect many such categories. One of the most common vulnerability categories is buffer overflow problems. Experience tells us that several calls in the standard C and Posix libraries are prime candidates for a buffer overflow exploit, including `strcpy` and `sprintf`. Consider the following code example that uses `sprintf`:

```
void main(int argc, char **argv) {
  if(argc <= 2) {
    char error_msg[1024];
    /* argv[0] usually denotes the name
     * of the program (enforced only by
     * convention)
     */
    sprintf(
        error_msg,
        "%s: not enough arguments!\n",
        argv[0]
    );
  }
}
```

This code might look harmless, but attackers can easily exploit it on many architectures by overflowing the `error_msg` string, which is stored on the stack. A malicious user can write a program to call this code so that the data copied into the `error_msg` buffer exceeds the 1,024 characters allocated to it. Done properly, this attack can compromise a machine.

We can easily detect instances of `sprintf` in source code with the Unix utility `grep`. Such a tool can report every instance of the string `sprintf` in a program, and someone can then inspect each occurrence and determine whether it is exploitable.

Because code scanning is necessarily incomplete, the scanner might often encounter situations in which it cannot definitively determine whether a construct is a security bug. Deciding what to do in such a situation is one of the most important engineering decisions in designing a code scanner. If the scanner flags a construct that is actually correct (a false positive), it wastes users' time and draws their attention away from real problems. If the scanner fails to flag a construct that is actually incorrect (a false negative), then its utility is diminished. Developers must engineer scanners to balance these two concerns. To address the problem of false positives, we believe that a scanner should group its results both by confidence in the result and by potential severity of any vulnerabilities it discovers.

Nonetheless, even with numerous false positives, simple sorts of scanning can save significant amounts of time, because they dramatically reduce the amount of code that developers must manually inspect. For example, SSH version 1.2.26 (a remote login server that uses cryptography to protect data as it traverses an insecure network) has at least one known buffer overflow in a `sprintf` call. If we did not know about this vulnerability and were looking for problems, we would have to look through 36,739 lines of code in just the base distribution. With even the simplest code scanners, we can pare this down to 134 lines of code, providing a much better starting point for a focused manual inspection.

References

1. C. Landwehr et al., "A Taxonomy of Computer Program Security Flaws," *Computing Surveys*, Vol. 26, No. 3, Sept. 1994, pp. 211–255.
2. P. Neumann, "Computer System Security Evaluation," *1978 National Computer Conf. Proc., AFIPS Conference Proc.*, 1978, pp. 1087–1095.

an automated tool to know which variables are meant to store secrets.)

It works by looking at and analyzing the source code, as opposed to a dynamic vulnerability detector, which would try to find problems at runtime. Static scanning can be quite effective in preventing common mistakes when developers use it as an aid. However, this technique is problematic when developers apply it to the problem of protecting against hostile code. It is too easy for attackers to avoid particular code structures that they know a scanner can detect. The upshot is that we can best apply scanning technologies during code development or certification (that is, by people concerned about avoiding vulnerabilities and coding defensively) and not as a line of defense on the client side (by people worried about externally developed code). Scanning tools necessarily use best-effort methods, and a hostile programmer can evade a scanner with clever coding. When the developer is trying to cooperate, scanning is much more useful. (See the "Static Code Scanning" sidebar for more information.)

Figure 1 shows Jslint's top-level user interface. The user chooses a file, directory, or archive to scan. We provided check boxes that let the user specify precisely which rules the tool should check. This feature lets a user ignore certain rules and avoid some potential false positives.

When the user has specified which code to scan and hits the scan button, a results browser appears, showing all trouble spots the scanner detects and sorting the results by rule, file, and line. We integrated Jslint with GNU Emacs. Clicking on a particular error in the results browser jumps the user

to the offending line of code in Emacs. An online help system explains why each rule is a potential hazard and suggests possible improvements.

Jslint can also run on byte code. In its current implementation, the scanner uses `jed`—a freely available Java decompiler—to report problems based on scans of decompiled output.

The Jslint implementation is straightforward. First, it parses the source file into a syntax tree that it can traverse using the Visitor design pattern.[11] This pattern encapsulates each operation on a syntax tree into a single object called a Visitor, allowing users to define new operations on a tree without changing the tree elements. It encodes each rule in a single Visitor that traverses the parse tree looking for instances of the violation in question.

Each scanning criterion is a single class that is dynamically loaded at runtime. The base scanner simply runs through each of the scanning guideline classes and collects the results (potential security vulnerabilities) that each one finds in the source file. This loose coupling provides for great extensibility, so users can add new scanning criteria.

Users can easily port our tool to scan other languages by following three simple steps:

1. replace the Java grammar as appropriate,
2. write a new set of Visitor classes to detect each problem area in the new language, and
3. add a new user interface.

One drawback of this tool is that it is completely static. Unfortunately, Java is a highly dynamic language that fetches classes as they are required (possibly across the Web). Because we do not execute our tool at runtime, code that we have never scanned and that possibly includes flaws might later be dynamically loaded. We do not view this as a large problem, because our scanner is meant for software developers or certifiers. We presume that the developer is interested in identifying and scanning all the code. However, any use of Jslint as a certification technology must take its currently static nature into account.

A Simple Example

Consider the code in Figure 2. This is a straightforward fragment that should look normal to the average programmer with no knowledge of our 12 rules. However, Jslint finds several stylistic weaknesses that attackers could perhaps leverage, given the right context (see Figure 3):

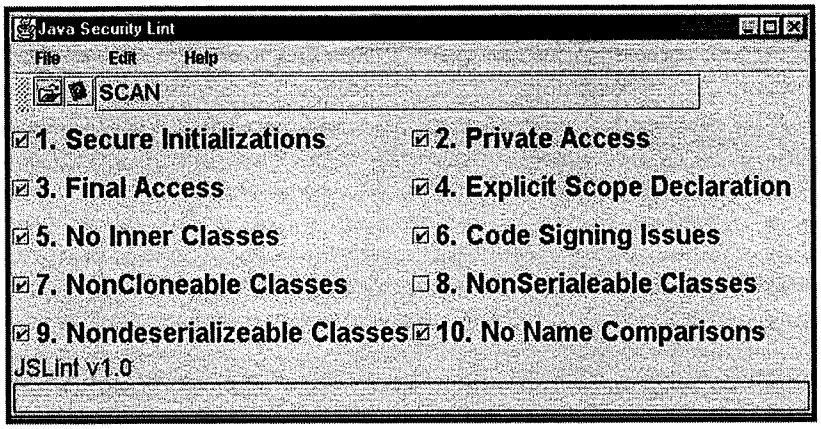


Figure 1. The Jslint user interface. (Note that we combined Rules 6 and 7 into item six.)

- The method `doSomething()` should check to see if the class is initialized before running (Rule 1).
- If possible, the `doSomething()` method should be private (Rule 2).
- If possible, the class and the method `doSomething()` should be final (Rule 3).
- If possible, all fields should be private (Rule 4).
- The class should not be cloneable, serializable, or deserializeable (Rules 8, 9, and 10).
- Using `getClass().getName()` on an object and comparing the result to a `String` doesn't always give the desired results. If a malicious user can substitute

```
public class TestClass
{
     String field;
     boolean inherited;

     public TestClass(String field){
       this.field = field;

       if(this.getClass().getName().
                        equals("TestClass")){
            inherited = false;
       }
       else {
            inherited = true;
       }
     }

     public void doSomething(){
       if(!inherited){
            System.out.println("Hello,"
                             + field + "!");
       }
       else{
            System.out.println("Hi," +
                             + field + "!");
       }
     }
}
```

Figure 2. A simple code example.

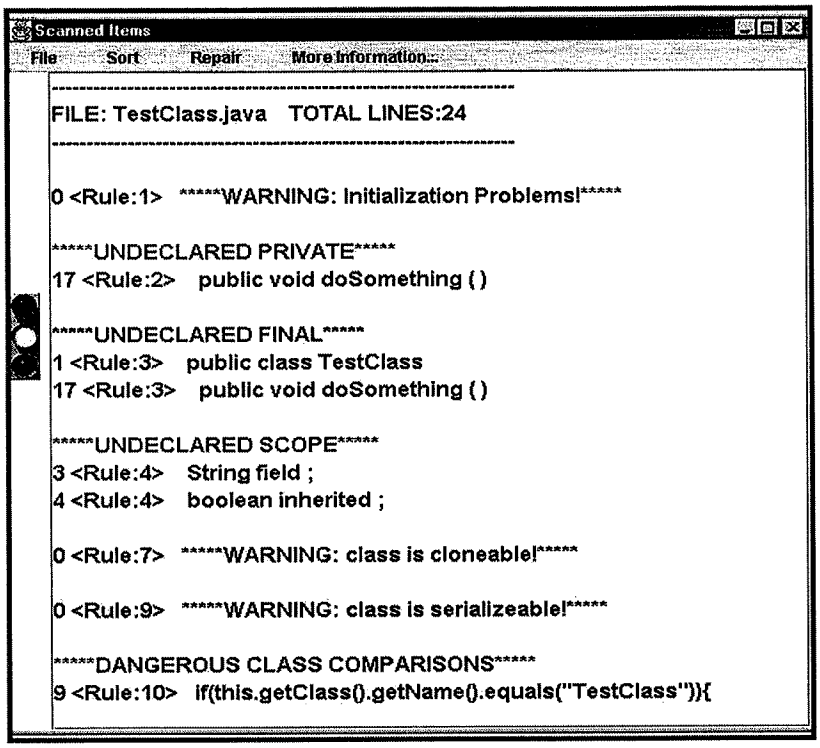


Figure 3. The Jslint results window.

an object instantiated from a different class in a different namespace with the same name, the test will not yield the intended result.

In the future, we plan to continue improving our scanner by developing new rules for secure Java code and incorporating them into the scanner. As time passes and the industry discovers new vulnerabilities in Java programs, the research community will likely gain more insight into what specific types of programmer errors lead to security vulnerabilities. We can and should incorporate any such information into our scanning tool. We also plan to add a dataflow analysis component to help reduce false positives. Such a component should be able to determine whether there is possible dataflow from an arbitrary input to a line in question. Essentially, the component would calculate slicing information and automatically analyze that information. Lastly, we hope to apply the scanning framework in our prototype to other languages, possibly addressing C and C++ vulnerabilities.

As we discussed earlier, our approach makes the most sense when developers and organizations apply it with a desire to create secure code. Therefore, investigating the use of static scanning technologies in certification and certifying some properties of code in these situations might be possible.

Acknowledgments

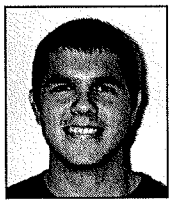
The Advanced Technology Program, administered by the National Institute for Standards and Technology Cooperative Agreement 70 NANB7H3049, supported this work.

References

1. G. McGraw, "Software Assurance for Security," *Computer*, Vol. 32, No. 4, Apr. 1999, pp. 103–105.
2. C. Cowan et al., "Stackgard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attack," *Proc. Seventh Usenix Security Symp.*, Usenix Assoc., San Diego, Calif., 1998.
3. R. Pethia, "CERT/CC 10th Anniversary Retrospective and Intruder Outlook for the Next 10 Years," *Information Survivability Workshop*, 1998.
4. T. Gilb and D. Graham, *Software Inspections*, Addison-Wesley, Reading, Mass., 1993.
5. G. McGraw and E. Felten, *Securing Java: Getting Down to Business with Mobile Code*, John Wiley & Sons, New York, 1999.
6. J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code," to be published in *Proc. Ann. Computer Security Applications Conf. 2000*, Dec. 2000.
7. D.S. Wallach and E.W. Felten, "Understanding Java Stack Inspection," *Proc. 1998 IEEE Symp. Security and Privacy*, IEEE Computer Soc. Press, Los Alamitos, 1998.
8. D. Dean, E.W. Felten, and D.S. Wallach, "Java Security: From HotJava to Netscape and Beyond," *Proc. 1996 IEEE Symp. Security and Privacy*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 190–200.
9. D. Dean et al., "Java Security: Web Browsers and Beyond," *Internet Besieged: Countering Cyberspace Scofflaws*, D.E. Denning and P.J. Denning, eds., ACM Press, New York, 1997.
10. G. McGraw and E. Felten, "Twelve Rules for Developing More Secure Java," *JavaWorld*, Dec. 1998, www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html (current Aug. 2000).
11. E. Gamma et al., *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, Reading, Mass., 1995.

About the Authors

John Viega is a senior research associate and senior consultant at Cigital (formerly Reliable Software Technologies). He is currently the principal investigator on a DARPA grant on adding security features to programming languages using the aspect-oriented programming paradigm. He is also writing two books, *Building Secure Software* (Addison-Wesley, 2001) with Gary McGraw and *Java Enterprise Development* (O'Reilly, 2001) with George Reese. Contact him at Cigital, 21351 Ridgetop Cir., Ste. 400, Dulles, VA 20166; viega@cigital.com.



Tom Mutdosch is a fourth year software engineering student at Rochester Institute of Technology. He is currently a coop at Intel's Sacramento campus. Previously, he cooped at the CIA and Cigital (formerly Reliable Software Technologies). Contact him at the Dept. of Computer Science, Rochester Inst. of Technology, 102 Lomb Memorial Dr., Rochester, NY 14623; tommut@csh.rit.edu.

Edward W. Felten is an associate professor of computer science at Princeton University. He also leads the Secure Internet Programming Group at Princeton, which is responsible for finding several security flaws in Java implementations. He has received a National Young Investigator award from the National Science Foundation and an Alfred P. Sloan Fellowship. He testified twice as an expert witness for the government in the US v. Microsoft antitrust case. He wrote both *Java Security* (John Wiley & Sons, 1996) and *Securing Java* (John Wiley & Sons, 1999) with Gary McGraw. Contact him at the Dept. of Computer Science, Princeton Univ., 35 Olden St., Princeton, NJ 08544; felten@cs.princeton.edu.

Gary McGraw's biography appears on page 40.