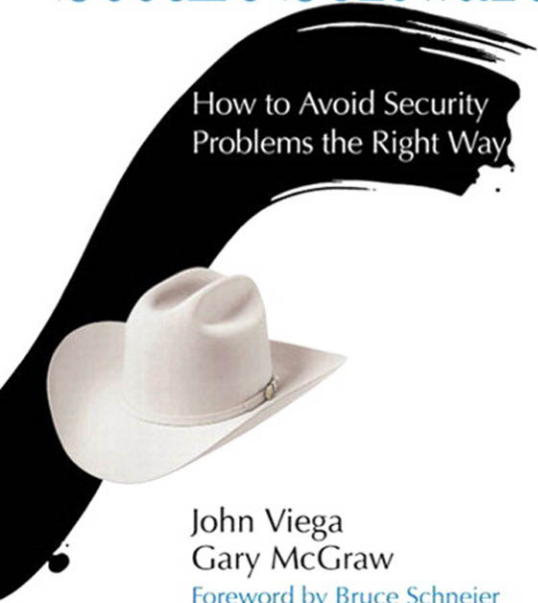


Building Secure Software



How to Avoid Security
Problems the Right Way



John Viega
Gary McGraw
Foreword by Bruce Schneier

Advance Praise for *Building Secure Software*

“John and Gary offer a refreshing perspective on computer security. Do it right the first time and you won’t have to fix it later. A radical concept in today’s shovelware world! In an industry where major software vendors confuse beta testing with product release, this book is a voice of sanity. A must-read!”

—*Marcus J. Ranum, Chief Technology Officer,
NFR Security, Inc. and author of Web Security Sourcebook*

“System developers: Defend thy systems by studying this book, and cyberspace will be a better place.”

—*Fred Schneider, Professor of Computer Science,
Cornell University and author of Trust in Cyberspace*

“Time and time again security problems that we encounter come from errors in the software. The more complex the system, the harder and more expensive it is to find the problem. Following the principles laid out in *Building Secure Software* will become more and more important as we aim to conduct secure and reliable transactions and continue to move from the world of physical identification to the world of digital identification. This book is well written and belongs on the shelf of anybody concerned with the development of secure software.”

—*Terry Stanley, Vice President, Chip Card Security,
MasterCard International*

“Others try to close the door after the intruder has gotten in, but Viega and McGraw begin where all discussions on computer security should start: how to build security into the system up front. In straightforward language, they tell us how to address basic security priorities.”

—*Charlie Babcock, Interactive Week*

“Application security problems are one of the most significant categories of security vulnerabilities hampering e-commerce today. This book tackles complex application security problems—such as buffer overflows, race conditions, and implementing cryptography—in a manner that is straightforward and easy to understand. This is a must-have book for any application developer or security professional.”

—*Paul Raines, Global Head of Information Risk Management,
Barclays Capital and Columnist, Software Magazine*

probably have plenty of ideas when it comes to the things you should be looking for. Enumerating them all is difficult. For example, in most languages, you can look for calls that are symptomatic of time-of-check/time-of-use (TOCTOU) race conditions (see Chapter 9). The names of these calls change from language to language, but such problems are universal. Much of what we look for consists of function calls to standard libraries that are frequently misused.

When we identify places of interest in the code, we must analyze things manually to determine whether there is a vulnerability. Doing this can be a challenge. (Sometimes it turns out to be better to rewrite any code that shows symptoms of being vulnerable, regardless of whether it is. This is true because it is rare to be able to determine with absolute certainty that a vulnerability exists just from looking at the source code, because validation generally takes quite a lot of work.)

Occasionally, highly suspicious locations turn out not to be problems. Often, the intricacies of code may end up preventing an attack, even if accidentally! This may sound weird, but we've seen it happen. In our own work we are only willing to state positively that we've found a vulnerability if we can directly show that it exists. Usually, it's not worth the time to go through the chore of actually building an exploit (something that is incredibly time-consuming). Instead, we say that we've found a "probable" vulnerability, then move on. The only time we're likely to build an exploit is if some skeptic refuses to change the code without absolute proof (which does happen).

This is the extent of our general guidelines for implementation audits. It is worth noting that this strategy should be supplemented with thorough code reviews. Scrutinize the system to whatever degree you can afford. Our approach tends to do a very good job of finding common flaws, and doesn't take too long to carry out, but there are no guarantees of completeness.

Source-level Security Auditing Tools

One of the worst things about design analysis is that there are no tools available to automate the process or to encode some of the necessary expertise. Building large numbers of attack trees and organizing them in a knowledge base can help alleviate the problem in the long term. Fortunately, when auditing source code, there are tools that statically scan source code for function calls and constructs that are known to be "bad." That is, these tools search for language elements that are commonly involved in security-related implementation flaws, such as instances of the `strcpy` function, which is susceptible to buffer overflows (see Chapter 7).

Currently, there are three such tools available:

- RATS (Rough Auditing Tool for Security) is an open source tool that can locate potential vulnerabilities in C, C++, Python, PHP, and Perl programs. The RATS database currently has about 200 items in it. RATS is available from <http://www.securesw.com/rats/>.
- Flawfinder is an open source tool for scanning C and C++ code. Flawfinder is written in Python. At the time of this writing, the database has only 40 entries. It is available from <http://www.dwheeler.com/flawfinder/>.
- ITS4 (It's The Software, Stupid! [Security Scanner]) is a tool for scanning C and C++ programs, and is the original source auditing tool for security. It currently has 145 items in its database. ITS4 is available from <http://www.cigital.com/its4/>.

The goal of source-level security auditing tools is to focus the person doing an implementation analysis. Instead of having an analyst search through an entire program, these tools provide an analyst with a list of potential trouble spots on which to focus. Something similar can be done with `grep`. However, with `grep`, you need to remember what to look for every single time. The advantage of using a scanning tool is that it encodes a fair amount of knowledge about what to look for. RATS, for example, knows about more than 200 potential problems from multiple programming languages. Additionally, these tools perform some basic analysis to try to rule out conditions that are obviously not problems. For example, though `sprintf()` is a frequently misused function, if the format string is constant, and contains no “%s”, then it probably isn't worth examining. These tools know this, and discount such calls.

These tools not only point out potential problem spots, but also describe the problem, and potentially suggest remedies. They also provide a relative assessment of the potential severity of each problem, to help the auditor prioritize. Such a feature is necessary, because these tools tend to give a lot of output . . . more than most people would be willing to wade through.

One problem with all of these tools is that the databases currently are composed largely of UNIX vulnerabilities. In the future, we expect to see more Windows vulnerabilities added. In addition, it would be nice if these tools were a lot smarter. Currently, they point you at a function (or some other language construct), and it's the responsibility of the auditor to determine whether that function is used properly or not. It would be nice

if a tool could automate the real analysis of source that a programmer has to do. Such tools do exist, but currently only in the research lab.

We expect to see these tools evolve, and to see better tools than these in the near future. (In fact, we'll keep pointers to the latest and greatest tools on this book's Web site.) Even if these tools aren't perfect, though, they are all useful for auditing. Additionally, these tools can even be integrated easily with development environments such as Microsoft Visual Studio, so that developers can get feedback *while* they code, not after.

Using RATS in an Analysis

Any of these above tools can be used to streamline our approach to implementation analysis. In this section, we'll discuss RATS. Most of the features of RATS we discuss in this section exist in some form in other tools.

First, we can use RATS to help us find input points to a program. RATS has a special mode called input mode that picks out all input routines in the standard C libraries. To run RATS in input mode on a directory of C files, type

```
rats -i *.c
```

This mode doesn't know anything about inputs via a GUI, so we need to find such places via manual inspection.

When we determine the internal input API of a program, we configure RATS to scan for this API as well. We can either add database entries of our own to our local copy of the RATS database or we can pass in function names at the command line during future scans. Let's say we found a function `read_line_from_socket` and `read_line_from_user`. In a subsequent scan we can add these functions to our search list as follows:

```
rats -a read_line_from_socket -a read_line_from_user *.c
```

Running RATS in this mode (the default mode) should produce a lot of output. By default, the output is ordered by severity, then function name. The following shows a small piece of code:

```
void main(int argc, char **argv) {  
    char buf[1024];  
    char fmt = "%d:%s\n";  
    int i;  
    FILE *f;
```

```
buf[0] = 0;
for(i=2;i<argc;i++) {
    strcat(buf, argv[i]);
    if(getenv("PROGRAM_DEBUG")) {
        fprintf(stderr, fmt, i, argv[i]);
    }
}
if(argc > 2) {
    f = fopen(argv[1], "a+");
    start_using_file(f);
} else {
    start_without_file();
}
```

Here is the output from RATS:

test.c:2: High: fixed size local buffer

Extra care should be taken to ensure that character arrays that are allocated on the stack are used safely. They are prime targets for buffer overflow attacks.

test.c:9: High: strcat

Check to be sure that argument 2 passed to this function call will not pass in more data than can be handled, resulting in a buffer overflow.

test.c:10: High: getenv

Environment variables are highly untrustable input. They may be of any length, and contain any data. Do not make any assumptions regarding content or length. If at all possible avoid using them, and if it is necessary, sanitize them and truncate them to a reasonable length.

test.c:11: High: fprintf

Check to be sure that the non-constant format string passed as argument 2 to this function call does not come from an untrusted source that could have added formatting characters that the code is not prepared to handle.

In the output above, we first see that the code contains a stack-allocated variable, which means that any buffer overflow that happens to be in the code is likely to be exploitable (see Chapter 7).

Next in the output, we see two calls commonly associated with buffer overflows: `strcat` and `getenv`. Here, `strcat` is used improperly, but the call to `getenv` is not exploitable.

Finally, we see an `stdio` call where format strings are variables and are not string constants. The general construct is at risk for format string attacks (see Chapter 12). In this instance, though, the format string is completely uninfluenced by user input, and thus is not a problem. RATS doesn't know any of this; all it knows is that an `stdio` function has been used where a variable was passed as the format parameter.

The default mode only shows reasonably likely vulnerabilities. If we pass the “-w 3” flag to RATS to show all warnings, we will see a call to `fopen` that may be involved in a race condition. However, because this is not reported in the default output, RATS couldn't find a file check that might match up to it, so it decides that there is a significant chance of this call is not actually a problem (it's hard to determine; there may need to be checks around the `fopen` that aren't performed. It depends on the context of the program).

On a real program, RATS may produce hundreds of lines of output. We can't emphasize enough how important it is not to skim the output too lightly, even when it seems like you're only getting false hits. False hits are common, but real vulnerabilities often lie buried in the list. Go as far through the output as you have the time to go.

Also, if you don't understand the issue a tool is raising, be sure to learn as much as you can about the problem before you dismiss it. This book should be a good resource for that.

The Effectiveness of Security Scanning of Software

Here are some initial conclusions based on our experiences using software security scanners:

- **They still require a significant level of expert knowledge.** Although security scanners encode a fair amount of knowledge on vulnerabilities that no longer must be kept in the analyst's head, we have found that an expert still does a much better job than a novice at taking a potential vulnerability location and manually performing the static analysis necessary to determine whether an exploit is possible. Experts tend to be far more efficient and far more accurate at this process.
- **Even for experts, analysis is still time-consuming.** In general, this type of scanner only eliminates one quarter to one third of the time it takes to

perform a source code analysis because the manual analysis still required is so time-consuming. Code inspection takes a long time.

- **Every little bit helps.** These tools help significantly with fighting the “get-done go-home” effect. In the case where a tool prioritizes one instance of a function call over another, we tend to be more careful about analysis of the more severe problem.
- **They can help find real bugs.** These tools have been used to find security problems in real applications. It is often possible to find problems within the first 10 minutes of analysis that would not otherwise have been found as quickly [Viega, 2000].

Conclusion

Performing a security audit is an essential part of any software security solution. Simply put, you can’t build secure software without thinking hard about security risks. Our expertise-driven approach has been used successfully in the field for years and has resulted in much more secure software for our clients. By leveraging source-code scanning tools, an architectural analysis can be enhanced with an in-depth scan of the code. As the field of software security matures, we expect to see even better tools become available.