Why is it we can't seem to produce
secure, high-quality code?

MARCUS J. RANUM, TRUSECURE

It doesn't seem that a day goes by without someone announcing a critical flaw in some crucial piece of software or other. Is software that bad? Are programmers so inept? What the heck is going on, and why is the problem getting worse instead of better?

One distressing aspect of software security is that we fundamentally don't seem to "get it." In the 15 years I've been working the security beat, I have lost track of the number of times I've seen (and taught) tutorials on "how to write secure code" or read books on that topic. It's clear to me that we're:

- Trying to teach programmers how to write more secure code
- Failing miserably at the task

We're stuck in an endless loop on the education concept. We've been trying to educate programmers about writing secure code for at least a decade and it flat-out hasn't worked. While I'm the first to agree that beating

one's head against the wall shows dedication, I am starting to wonder if we've chosen the wrong wall. What's Plan B?

Indeed, as I write this, I see that Microsoft, Intel, and AMD have jointly announced a new partnership to help prevent buffer overflows using hardware controls. In other words, the software quality problem has gotten so bad that the hardware guys are trying to solve it, too. Never mind that lots of processor memory-management units are capable of marking pages as nonexecutable; it just seems backward to me that we're trying to solve what is fundamentally a software problem using hardware. It's not even a generic software problem; it's a runtime environment issue that's specific to a particular programming language.

Normally, when someone mentions programming languages in an article about software quality, it's an invitation for everyone to jump in with useful observations

SECURITY

The root of the problem

SECURITY

The root of the problem

such as, “If we all programmed in [my favorite strongly hyped programming language], we wouldn’t have this problem!” That might be true in some cases, but it’s not reality.

We tried legislating a change of programming languages with Ada back in the 1990s. Remember Ada? That was an expensive disaster. Then we tried getting everyone to switch to a “sandboxed” environment with Java in the late 1990s, and it worked better—except that everyone complained about wanting to bypass the “sandbox” to get file-level access to the local host. In fact, Java worked so well, Microsoft responded with ActiveX, which bypasses security entirely by making it easy to blame the user for authorizing bad code to execute. Please, let’s not have any more alternative programming languages that will solve all our problems!

What’s Plan B? I think that Plan B is largely a matter of doing a lot more work on our compiler and runtime environments, with a focus on making them embed more support for code quality and error checking. We’ve got to put it “below the radar screen” of the programmer’s awareness, just as we did with compiler optimization, the creation of object code, and linking. We’ve done a great job building programming environments that produce fast executables without a lot of hand-holding from the programmer. In fact, most programmers today take optimization completely for granted—why not software security analysis and runtime security, too? For that matter, why are we still treating security as a separate problem from code quality? Insecure code is just buggy code!

IMPROVE SECURITY WITHOUT SWITCHING LANGUAGES

Anything we do to improve software security must work without the programmer having to switch languages. We need to be realistic in recognizing that we’re stuck with a set of languages and environments that are not susceptible to a massive change. So, we need to look at how to improve security within the context of what we already have. A tremendous amount of progress can be made with a few simple steps. Consider the following extremely small piece of bad code:

```
main()
{
    char  buf[512];
    printf("you said: %s %d\n", gets(buf));
}
```

The C library routine `gets()` reads a line from the terminal into a memory region that is provided for it to fill. `Gets()` used to be a huge problem for programmers because it doesn’t actually check the size of the memory region—a recipe for disaster. Back in 1994, I used to teach C programmers, “Don’t use `gets()`; it is bad!” But, of course, they kept doing it—a lot of textbooks use `gets()` because it is simple for writing example code. But, then, something changed: a few security-conscious programmers got sick of `gets()` and put a stake through its heart (see figure 1).

The compiler/linker generates a warning when someone tries to use that function. What a great (and simple) idea. In an earlier attempt to kill `gets()`, some of the programmers who maintain the Unix C library modified the `gets()` function itself to cause it to emit the warning at runtime. This worked well, until some programmers started posting patches to remove the warning message.¹ Why fix broken code if you can just shoot the message?

Of course, this has been tried before—most compilers generate various warnings when they encounter questionable code. Old-time Unix/C programmers will certainly recall `lint(1)`, a code-checker that did cross-file error checking and parameter type matching. These tools have existed for years but are not popular. Why? Because they generate a lot of warnings, and, as countless software engineers have pointed out, it’s time-consuming to sift through the spurious warnings looking for the ones that really matter. I’ve got news for them: there is no such thing as a warning that doesn’t matter. That’s why it warns you. Anyone who has worked with enough code will tell you that, generally, software that compiles without warnings crashes less often. As far as I’m concerned, warnings are for wimps. Tools such as `lint(1)` and `DevStudio` should not issue warnings: they should decide if they’ve found an error and stop the build process, or they should shut up and generate code.

FORGET PROGRAMMER COOPERATION

Anything we do to improve software security must not rely on cooperation from the programmer. I'm not implying that programmers (except in rare cases) will attempt to out-and-out bypass tools for software security, but it would be an awful lot easier if the tools were simply embedded in the development environment. Code already gets pushed through a preprocessor, compiler, then optimizer—why not add another step, or merge security features into the existing steps?

When I started programming in C in the early 1980s, some of us used to sneer at the programming languages that did garbage collection. Usually, the sneering was aimed at the performance impact of garbage collection. Today, garbage collection is a fairly well-understood problem, and its costs have been dramatically reduced. During the 1990s a lot of students wrote papers on memory allocators and garbage collectors, each of which pushed the state of the art another step forward. Now we can pretty

much treat garbage collection as “free” and recognize that a programmer's time is too valuable to spend worrying about memory allocation.

Of course, most of our programming is still being done in C/C++—languages that force programmers to manually mate up a `free()` with each `malloc()`. I'm amazed that someone hasn't come up with the idea yet of making `malloc()` and `free()` stub-calls into a generic garbage-collected memory allocator and doing away with C memory management altogether. Let the programmers who want to `malloc()` do so to their heart's content, but manage the memory using a completely different runtime from the current approach used by C. It would work without having to change a line of code, and memory leaks would be a thing of the past. We should be able to do the same kind of thing with security: put it below the “radar screen.”

What might work? I think we've got all the fundamental techniques we need for good software security,

we're just not using them. Obviously, we understand how to parse source code and generate an executable from it. We understand how to build runtime environments, and we even understand how programming languages use the stack. Otherwise, none of the code we write would work at all! Why do we have compilers that will accept questionable code by default, when they could just as easily reject it by default? Consider the example in figure two.

This is exactly the same code I used in the earlier example, which built without complaint using the default compiler options from `make`—but it complains bitterly when I turn on full compiler warnings with the `-Wall` flag. Some of those warnings are probably noise, but it caught a bug that I deliberately tried to slip past you in the first

Example of Default Compiler Options

```
mjr@lyra-> make foo
cc  foo.c -o foo
/tmp/ccpU9766.o: warning: gets() is very unsafe; consider using fgets()
mjr@lyra->
```

FIG 1

Example of Compilation with Full Warnings

```
mjr@lyra-> cc -Wall -o foo foo.c
foo.c:2: warning: return-type defaults to `int'
foo.c: In function `main':
foo.c:5: warning: implicit declaration of function `printf'
foo.c:5: warning: implicit declaration of function `gets'
foo.c:5: warning: format argument is not a pointer (arg 2)
foo.c:5: warning: too few arguments for format
foo.c:6: warning: control reaches end of non-void function
/tmp/ccx29876.o: warning: gets() is very unsafe; consider using fgets()
mjr@lyra->
```

FIG 2

SECURITY

The root of the problem

example: a spare parameter for `printf()` was missing. The real question that this example makes me ask is, why did the compiler generate any code at all, since this code is obviously broken? If I run it:

```
mjr@lyra-> foo
bar
you said: bar -541075068
mjr@lyra->
```

...it cheerfully prints back my input and then some value from someplace in the execution stack that my code has no business looking at. We should be able to do better than this. Note the “annoying” warnings that GCC (GNU Compiler Collection) emitted and which I ignored. Why on earth did the compiler generate code when it rightly “knew” it was buggy?

Right now, the state of the art in software security is to pass your code through some kind of static source-code analyzer such as ITS4 or Fortify that looks for dangerous practices and known bugs. That’s a great start, and, according to my friend Gary McGraw—chief technology officer of Cigital and author of several books on software security—who works with the stuff, it catches a significant number of potential security problems. But, as you can see, the compiler already knows a lot of what it needs to in order to make a good stab at determining what is being done wrong.

One really neat concept is embodied in the Perl programming language—tainting. The idea of *tainting* is that the interpreter tracks the source of data and turns off dangerous operations if they are called directly as a result of user input. For example, when you’re running a Perl script in taint mode, it turns on a lot of error checking before passing user-provided data to certain system calls. When you try to open a file for write using a filename that is tainted data, it checks to make sure the directory tree ownerships for the target directory are correct and that the filename doesn’t contain “../” path expansions. In other words, the runtime environment tracks not just the type and value of the data but also its origin. You can imagine how nice this capability can be for writing server-side code or captive applications.

Unfortunately, few programmers use tainting because

it imposes an extra burden on the programmer, and it’s sometimes difficult to figure out a secure way to get the job done. But what if we built tainting-type capabilities right into our runtime environments for C/C++? A simple high-value approach might be to modify I/O routines (read/write) to determine if they are connected to a socket from a remote system, and to do some basic checks on data coming across it, such as checking to see if the stack is altered across calls to certain functions following I/O.

For example, back in the 1980s I used a tool on MIPS Ultrix² called Pixie that would take a compiled executable, insert performance-collecting routines at function points, and write a new executable that had been instrumented for performance measurements. The MIPS instruction set that Pixie worked on was pretty complicated RISC-y stuff, so I assume that Pixie was a pretty neat piece of work under the covers. But what’s important is that, in all the times I used Pixie on an executable, it never created an executable that failed to work properly or that was dramatically slower than the original optimized code. So here’s one possibility: What if someone writes a “security post-processor” that takes an ordinary executable and emits a “hardened” version? It might be feasible to emit a hardened version of an application that does additional checks on memory usage, socket usage, calls to the operating system, or file I/O. Tools such as Purify and BoundsChecker already do a form of this but primarily as a means of looking for memory leaks or dangling pointers. How about a post-processor that adds Perl-style tainting into the runtime without requiring any action on the part of the programmer?

I use a software development environment for C programming called CodeCenter (formerly Saber-C) that is a C language interpreter. It’s a wonderful debugging environment because one of the things it does is emulate a runtime environment in its interpreter. So, when you `malloc()` a chunk of memory and stick a pointer to an integer in it, it stops execution and generates an error if you then change the pointer to point to a character array or unallocated memory. Part of what it does is to bypass the normal C library routines for memory allocation and replace them with versions that do exhaustive error

checking—a good idea.

For compiled applications we could build “smarts” into the compiler so it passes more information about the type and size of statically allocated memory, which the linker could take advantage of for adding runtime checks. Sure, it might slow down our compile-link cycle a tiny bit or make our object code a bit bigger, but I’d rather have software that runs correctly than software that is tiny. Besides, these days, it seems most software is neither tiny nor correct. If it becomes an issue, then give me a compiler flag that lets me mark certain modules as “safe for handling data accepted from a user,” then turn the checking off after I’ve tested the modules thoroughly.

DECENT ENVIRONMENTS

I think we need to recognize that we’re building software for the 21st century using 20th-century tools. C is almost certainly not the best language for a lot of the applications it is used for, but that’s unlikely to change. Therefore, let’s try to upgrade our environments a bit. As a Unix systems developer for a long time, I have been dismayed at the qualitative differences in the development tools between the Unix and Windows environments. The tools that are available to Windows developers are fantastic, compared with what most Unix programmers use: GCC, GDB (GNU debugger), and Make.

This is one area where I believe the open source movement has hurt us more than it has helped us: the availability of free, adequate tools for Unix has gutted the potential market for commercial high-quality tools. Very few programmers are willing to pay thousands of dollars for a better programming environment, when the customer can’t tell the difference by how the resulting software runs. The Windows programmer has access to fully integrated environments that manage dependencies, debuggers that render execution with amazing detail, and visual development engines that take most of the work (and all of the errors) out of user-interface code. We need to put security on the to-do list for the companies that are building those environments, so it’s just another embedded feature. When you’re operating in an integrated environment, it’s not hard to add the kind of extra analysis and cross-checking that could make all the difference.

WHERE DO WE GO FROM HERE?

If all the industry analysts are right, software is going to be one of the crucial building blocks of the global information economy. We’re already seeing that software development costs sometimes dominate product development budgets for some manufactured goods. This all

points to a future where software development is driven faster and faster, and potential failures become increasingly expensive. When you consider the rate at which development is being outsourced, it’s clear to me that the winners in the software game will be the ones who can write more, better code faster. They won’t be purists and they won’t care if they lose a few CPU ticks on garbage collection and runtime security if it means they win the development contract and deliver on time.

I realize I sound negative about the state of software security, but I think that what we’re seeing is only a temporary crisis. In the software world of the future, only the strong will survive, and I think that security is increasingly being seen as a weakness. This could fuel the development of a whole new set of capabilities in our development environments. Do you remember writing Windows user-interface code in the days before visual development environments? If you compare the tools today with the tools that were available back then, I think you can get a good idea how far we’ve come and how far we’re likely to go.

Whenever there’s a major shift in programming environments, there’s always a backlash of old-school programmers who insist that the new approach is too bloated, too slow, or too complicated. There’s always the problem of legacy code, and people complain about the cost of upgrading it. But the truth is: old-school programmers retire or become managers, and the legacy code gets replaced. To me, the real question is: Will we be able to get security embedded into the tools soon enough? If we can, perhaps time is on our side. Q

REFERENCES

1. For information on programmers who started posting patches to remove the warning message, visit the Grass GIS (Geographic Resources Analysis Support System) Web site: <http://grass.itc.it/pipermail/grassuser/2000-May/003586.html>.
2. Digital Equipment Corporation’s version of Unix.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MARCUS J. RANUM, senior scientist at TruSecure, is an expert on security system design and implementation. He has designed a number of security products and has been involved in every level of operations in the security product business, from developer to founder and chief executive officer.

© 2004 ACM 1542-7730/04/0600 \$5.00