Andy Chou

Static Analysis, Security Holes, & Networking Code

Examining source code at compile time

Writing code that interacts with connections to the Internet and many users from the outside world is difficult. Developers writing for networking and telecommunications must contend with burgeoning code sizes, multiple platforms, and countless protocols that could present dangerous, unforeseeable security problems. In short, there are myriad challenges facing developers of any network-enabled application or device. I've seen a lot of telecommunications and networking code. Customers using the static-analysis technology I helped develop includes all but one wireless OS vendor (e.g., Symbian and Palm), the top networking companies (e.g., Juniper, Cisco, Avici, Ciena, and HP), and several telecommunications companies (e.g., France Telecom and AudioCodes). As important as it is to find the right solution to this problem, though, it is also important to understand why the available solutions could also be the wrong solutions. In this case, I am referring to the emergence of static analysis as a new tool for securing networking code. "Secure your software by fixing the code before it ships" is the latest mantra for a new breed of tools and companies that are jumping into this market.

Static Analysis

Static analyzers examine the source code at compile time rather than running the program and seeing what happens. A static analyzer never actually runs a program. Instead, the program source code is scanned in some way for potential violations of rule sets. For instance, a simple rule set would describe stylistic rules. While these stylistic rules are not indications of impending doom, they might point out unclear or misleading code that is worth changing if you have some time on a rainy day. Lint falls into this category of tools.

Andy Chou is cofounder and CTO at Coverity, where he is responsible for developing Coverity's source-code analysis technology. Dr. Chou received his Ph.D. in computer science from Stanford. He can be contacted at andy@coverity.com.

Moving up the sophistication ladder, the next class of static-analysis tools is the super-grep tools. These tools check for code patterns that can be identified with a minimal amount of context and that may (or may not) indicate that there is something wrong with the code.

At the top of the sophistication heap are those static-analysis tools that use either compiler-analysis techniques, theorem-proving techniques, or a combination of the two to focus on finding mistakes in the source code that will have disastrous effects at runtime—system crashes, back doors into the system, or memory corruption. At Coverity (where I work), I'm developing one such tool that uses a combination of compiler techniques and savvy software engineering to make static analysis scale to large programs. When discussing static analyses in this category, the most important indications of a sophisticated analysis are:

- Can it cross procedure boundaries (is it interprocedural or intraprocedural)?
- How much does it know about pointers and the shape of the heap (how sophisticated is its alias analysis)?
- Does it find all of the bugs of a certain type or might there be some bugs in the code even if the tool claims there are none (is the analysis sound)?
- What price do you pay for the results that are reported—are most of the reports real errors in the code or is there a lot of noise (what is the false-positive rate)?

Beware of Silver Bullets

Static analysis is no panacea for finding security holes in networking and telecommunications code. I don't say that because I don't like static analysis—I work for a static-analysis company after all—but because there are fundamental problems that have not been solved when applying static analysis to security properties. Many vendors claim to have static tools that find security holes out of the box, but are they really offering little more than developer aids—not tools that actually find and accurately describe live security holes? Here's what you should know.

Networking and telecommunications code inherently deals with protocols, and these protocols dictate a large part of the implementation of such systems. Understanding a protocol requires more than just the source code because protocols inherently deal with multiple participants communicating according to the protocol. The properties of the protocol are not explicitly stated in the source code; instead, they arise out of the interaction between multiple nodes participating in the protocol and the network they are using to communicate. Static analyzers have no notion of what a protocol is—there is simply too much missing information to analyze code directly without running it and still derive a coherent description of a protocol. Of course, there have been experiments with model checking. This technique requires either running the code (not really static—your code must be able to run!) or building a model of the code in a formal language (not C or C++, or anything even remotely like them). Model checkers have the luxury of being able to encode the actual state of a protocol as well as having a description of the possible transitions from every protocol state, such as received message types, timeouts, and errors. Encoding the possible transitions requires manual work—work that static analyzers shouldn't require. Static analyzers understand the source code and have a hard enough time

(continued from page 14)

navigating your interprocedural call graph, much less your protocol. So don't expect these analyzers to find deep protocol flaws having to do with security, authentication, or any other interesting emergent property.

You need alias analysis—and you don't have it. Networking code deals with packets from the outside world—packets with arbitrarily scary contents. Where do the "tainted" contents of network packets end up? Which pieces of code deal with the stuff you pulled straight out of a network packet, jammed into a structure, and yanked out millions of instructions later, when it seemed like the "right time" to deal with that data? To really find security holes, you need to be able to tell which pointers could point to that poisonous data, and which ones ultimately end up being consumed in a trusted manner. If a database consumes that data, you might get a SQL-injection attack. A printing or logging function? A format-string attack. An allocation function? A buffer-overflow attack. A loop bound? A denial-of-service attack. An arithmetic expression? An integer overflow. There are many types of security holes all caused by trusting some form of input that shouldn't be trusted—and you need alias analysis to scratch the surface.

Why don't we have alias analysis? Because it's hard—really hard. It's so hard that some of the brightest people in computer science have tackled it for decades and only come up with algorithms that are precise but don't scale, or scale but aren't precise, or scale and are precise but only for well-chosen input programs. It's considered a victory if an alias analysis can handle 1 million lines of code while forgetting about fields, ignoring the ordering of all statements in the entire program (meaning they can't tell the difference between $p = q; q = r;$ and $q = r; p = q;$), and generally losing precision left and right. It's bad enough that these analyses lose precision when scaling to what would be considered a moderately sized code base in networking/telecommunications systems, but it's also a nightmare to interpret the results of an analysis that has such imprecision. For example, say you get an error at line 16 stating, "Potential buffer overflow here because the variable p points to a buffer of size 10." Okay, so why does p point to a buffer of size 10? The answer is likely to be, "Well, there was an assignment from X to p in this totally unrelated place (and I can't tell you how we got from there to here), and some other assignment of Y to X in yet another unrelated place very far away in the code, and, oh yeah, there's some buffer allocated here, and it looks like it might be pointed to by Y..." Not exactly an easy diagnosis for the person looking at the results.

Even if you had a context-sensitive, flow-sensitive, path-sensitive, field-preserving, full-blown alias analysis, it wouldn't be enough for many code bases. Many large systems are not all code, you see. Parts of many systems are inherently dynamic, and no static analyzer can get at these pieces of your code without lots of help. For example, there may be a dynamic library loaded that you know is that library a colleague down the hall wrote last week, and the function `foo()` he wrote does X, Y, and Z. But the dynamic symbol is found by passing a string "`foo`" to `dlsym()`, which your static analyzer doesn't understand (because it can't know which dynamic library is being loaded without your help). Using `dlsym()` is the easiest of such scenarios. For various reasons, developers of networking devices often reimplement essentially the same functionality of DLLs using a custom mechanism; for example, by using a bunch of text files describing linkages along with macros to mangle function names in just the right way, and a function symbol lookup that amounts to a big switch statement (also generated through macro hackery) that compares strings passed into the lookup function with the strings in the text file. No static analysis is going to be able to figure that out without tens of thousands of dollars of professional services. More than likely, the analysis will think that all of the functions in that big switch statement could be called every time the dispatch function is called, leaving you with a hornet's nest of false positives. If you want your analysis to understand such things more accurately, you'll probably be asked to pay for expensive customization—or add annotations to the source code yourself. What's so bad about annotations? Think of the last time you tried to make a method `const` in your C++ code, then multiply that pain by a thousand (annotations for serious static analysis can easily be larger than the code being annotated!).

Alias analysis and annotations are just two pieces of what's usually necessary for a static analysis to be sound. A sound analysis for security properties is what you really want; it's

a guarantee that if the analyzer says "no vulnerabilities of this form found in your code," then there are none. Some static analyzers will even give you a proof that the program is safe from certain kinds of harm. But soundness comes at a high cost: Only a limited number of properties can be checked in this way, with poor scalability and huge numbers of false positives. Currently, the best sound analyses only scale to hundreds of thousands of lines of code—with a few academic projects claiming to break the 1 million lines of code mark. To the best of my knowledge, the largest programs that have been verified by some form of sound analysis are programs owned by Microsoft—and you won't be seeing those analyzers any time soon. For one thing, it probably takes a small army of very expensive researchers to get anything to go through such an analyzer.

Another difficult piece for static analyzers to help with is configuration. Configuration can come in many forms, from entries in a default shipped database to XML files, to flat files with wacky formatting conventions, to entries in the Windows registry. Configuration is also the source of innumerable security holes—some application features are inherently insecure; only an obscure string in some text file protects your system from the hacking hordes. Given the wide variety of formats and semantics for configuration, it's very likely that anything built into a boxed static-analysis product will only give you shallow or incomplete analysis of the security implications of the default configuration of a system. And no static-analysis tool will help you debug your application's documentation or online help to make it clear to users what the security implications are of different configuration items.

More generally, the context of the deployment and the configuration used in that context are of huge importance when evaluating the security risks in code. All of the generalizations made about secure code might be totally wrong, depending on context. Does code going into the next-generation attack helicopter need to be "secure"? It depends—is it really necessary to have passwords and authentication when the thing is going to be embedded into a machine guarded by a platoon of beefy marines packing machine guns 24/7? Maybe. Or maybe it's more of a risk to lose or forget the password, or to get locked out after three tries while your fingers are shaking because of the bombs going off all around. Perhaps a special physical key should be used instead—as long as you don't lose it. Context is important when evaluating security. Static analyzers typically don't have nearly enough intelligence or information to take context into account.

Of course, there are plenty of static analyzers for security. There are millions of venture capital dollars being invested in static-analysis tools for security. Yet most of that money is being spent on pretty interfaces for managers; I give credit to the management types out there for not buying more of such products. The simple rule with pretty reports is that garbage in equals garbage out. Are the "potential vulnerabilities" actually vulnerabilities or just false positives? If they're false positives, the management report you're staring at is just the static analyzer telling you where it isn't smart enough to figure out that you don't have any vulnerabilities. You would fire a human who gave you such a report ("Here's a pie chart showing the thousands of places where I couldn't tell if you didn't have a security hole, oh look—your TCP/IP stack code looks pretty bad, I think..."), so why pay as much as a person's salary to purchase such a tool? There are plenty of free static analyzers (RATS [1] and ITS4 [2] come to mind) that are essentially glorified versions of grep with a vulnerability database attached (a database of the form "function *A* might be bad because of *Y*"). These tools are no substitute for having security expertise, but they can be helpful as developer aids—constant, hit-you-over-the-head-every-time style reminders to developers to not screw up. The "advanced" static analyzers for finding security holes usually aren't enough of a step up from the free tools to warrant their cost. If you want a prettier ITS4, wait long enough and the open-source community will build one.

Where Can Static Analysis Help?

Despite the limitations, static analyzers can be important tools within a security manager's arsenal. They're actually excellent at finding

Example 1: *A security hole in the FreeBSD kernel.*

```
File: sys/compat/linprocfs/linprocfs.c
Function: linprocfs_doproccmdline

Call to function "copyin" TAINTS argument "pstr":
793 error = copyin((void *)p->p_sysent->sv_psstrings, &pstr,
794                sizeof(pstr));
795 if (error)
796     return (error);
TAINTED user-pointer "(pstr).ps_argvstr" dereferenced
797 for (i = 0; i < pstr.ps_nargvstr; i++) {
798     sbuf_copyin(sb, pstr.ps_argvstr[i], 0);
799     sbuf_printf(sb, "%c", '\0');
800 }
```

important defects, inconsistencies, and other ugliness in your code—and on occasion exploitable security holes. Static analysis for defect detection, in general, is substantially easier because there are many more defects than security holes, and defects are much easier to identify and report with a relatively low false-positive rate. It's still no cakewalk, by any means, to analyze millions of lines of code with an inter-procedural analysis taking into account calling context, false paths, and fields—not to mention the inherent difficulties in producing easy-to-understand error reports for specific defects. But it's possible today with existing techniques and some amount of cleverness.

The context of the deployment and the configuration used in that context are of huge importance when evaluating the security risks in code

Static-analysis tools can find some security holes. For example, Example 1 shows a security hole in the FreeBSD kernel that was found using Coverity's security analyzers (CVE name CAN-2004-1066). In this example, the analyzer found that data being copied from outside the kernel (using `copyin` on line 793) was being dereferenced in line 798 without a protecting check—a surefire way to get a denial of service. Fortunately, this security hole is only exploitable if the default FreeBSD configuration is changed. This is an easy example; a silly mistake made locally that can lead to disaster. What you can't expect is for a static analyzer to do too much, for it to dig too deeply into the code like a human security audit can. And you certainly can't expect almost any analyzer that's availab and scalable to guarantee the absence of categories of security hole

If your goal is to find securi holes with static analyzers, l prepared to spend a lot of tin analyzing the output. With son basic tools, the false-positive rate c be more than 1000 percent (at lea 10 false positives for every bug), ar each bug might have limite information about why it could be security hole in your code. The might be generic descriptions suc as "If *X* is true here, then there ma be a security hole *Y*." But that doesn tell you nearly enough to know if holds, and it doesn't tell you enoug to know if the security hole *Y* is exploitable for real. You need a expert to do the one thing a static analyzer can't: Really think abo your code with all of the invariants, context assumptions, an configuration defaults nobody ever bothered to write down.

References

[1] Rough Auditing Tool for Security (RATS); http://ww .securesoftware.com/resources/tools.html.

[2] http://www.cigital.com/its4/. ❑