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.

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 knowing 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...
Jammed into a structure, and yanked out. There are many types of security holes all 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 format-string attack. An arithmetic expression? An integer overflow.

Allocation function? A buffer-overflow attack.


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 format-string attack. An arithmetic expression? An integer overflow.

Allocation function? A buffer-overflow attack.


If a database consumes that data, you might get a SQL-injection attack. A format-string attack. An arithmetic expression? An integer overflow.

Allocation function? A buffer-overflow attack.


Static Analysis, Security Holes, & Networking Code

Andy Chou:

(continued from page 14)
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. Even 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’s not 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 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. Even 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’s not 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 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. Even 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’s not 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.
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 interprocedural 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.

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 available and scalable to guarantee the absence of categories of security holes.

If your goal is to find security holes with static analyzers, I prepared to spend a lot of time analyzing the output. With some basic tools, the false-positive rate can be more than 100 percent (at least 10 false positives for every bug), so each bug might have limited information about why it could be a security hole in your code. They might be generic descriptions such as “If X is true, then there might be a security hole Y.” But that doesn’t tell you nearly enough to know if it holds, and it doesn’t tell you enough to know if the security hole Y is exploitable for real. You need a expert to do that thing a static analyzer can’t: Really think about your code with all of the invariants, context assumptions, a configuration defaults nobody ever bothered to write down.

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