



New threats demand new code analysis tools

Developers cannot depend on old code review processes and testing tools to find all the bugs and potential vulnerabilities in today's connected devices. Modern static analysis tools, however, provide a complete system view and detect any weaknesses in the code that could lead to a malicious attack.

By Gwyn Fisher

Whether it's a Net-savvy, grocery-ordering super fridge or a wireless-capable, life-saving defibrillator, devices powered by embedded software are not only adding amazing new capabilities, they are also increasingly plugged into the global, public network. But by gaining connectivity and exposing – if not broadcasting – their presence to the world, these devices are becoming enticing targets for nefarious troublemakers.

Just because it's a seemingly innocuous fridge doesn't mean that a black hat with the time, motivation, and misplaced talent won't exploit the latent vulnerabilities an embedded developer unwittingly left in the device's software. The bad guys might turn the Web-enabled appliance into a slave processor or, worse, have the device appear to operate as intended while performing a variety of aggravating or illegal tasks behind the scenes.

Embedded developers have always had to worry about coding defects more than most mainstream developers. Embedded software powers devices that, upon failing

or even so much as hiccupping, can cost millions of dollars, ruin a company's reputation, or put lives at risk. But with built-in connectivity becoming common for many devices, embedded developers now have a whole new set of design and coding constraints to add to their already exhaustive checklist. As illustrated in Figure 1, the interaction of embedded device components makes code quality critical.

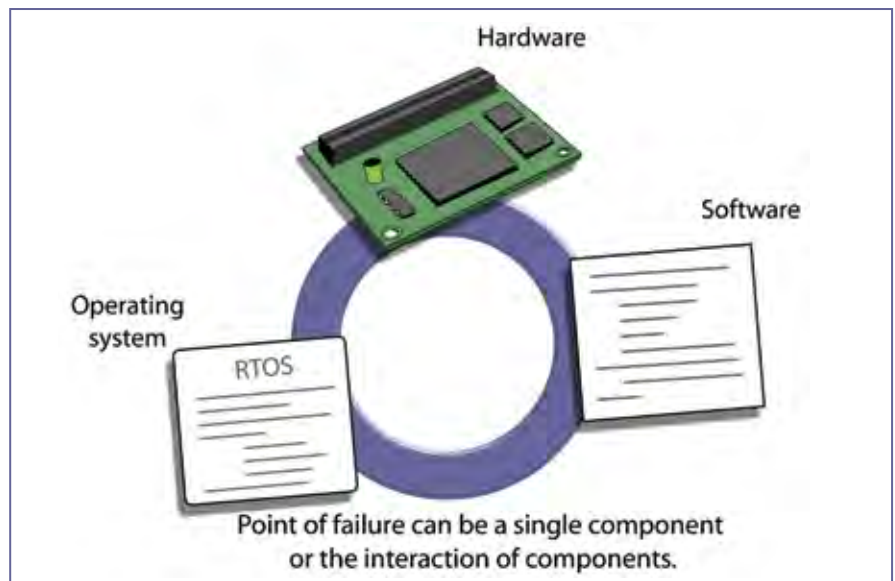


Figure 1

What makes it so hard to ensure embedded solutions are secure? The answer is that at least three components – the hardware and its connectivity to the outside world, a real-time or hardened operating system, and the software running on it – must be secure and stable and not crash or enable external exploits when interacting. While each component may be relatively defect-free and resistant to attack, the entire package may not be.

Automated code analysis tools needed

Embedded developers usually rely on their own coding skills, manual code reviews, unit tests, and post-build testing and quality assurance to hunt down software defects. However, with relatively small embedded software solutions consisting of 100,000 lines of code and larger projects containing millions of lines of code, the old code review processes and testing tools can't possibly be expected to find all the bugs and potential vulnerabilities. Neither can developers rely on these processes and tools to uncover the subtle issues that arise from the interplay of the various subsystems that make up a typical modern device.

It's already common practice to rigorously peer review only the main threads of execution within a code base, leaving the quality and security of many ancillary code paths up to the talent of the individual developer. But in the world of online devices, manufacturers can't be confident in their product's lack of vulnerability if the developers haven't applied rigor to every possible code path.

Faced with this type of challenge, it makes sense to look to automated tools for help. In the past, however, source code analysis tools have been very inaccurate, resulting in developers spending additional time combing through lengthy defect reports to determine if the issues the tool reported are even valid. The good news is that code analysis technology has advanced.

Not your father's lint

We've all done it. We've run lint – the open source static analysis tool and its variants – on that new module we're so proud of and watched the ensuing avalanche of warning messages fill our monitors. It's not pretty and, more importantly, not useful. Yet hidden in all that noise is real information – information about errors that developers need to know about. These errors could transform a new device from a state-of-the-art, revenue-generating, must-have product into a financial and public relations disaster that would fuel the development blogs for weeks.

Because lint and its offspring lack fairly basic capabilities and deliver high numbers of false positives, the tools' effectiveness in highlighting real concerns and thus ability to save developers' time is limited. What's worse, the poor-performing lint family tree has sullied the reputation of the entire concept of source code analysis.

Embedded developers need to know that lint does not represent the state of modern static analysis tools. Advances in technology and the thinking behind how the tools work have given rise to more useful, effective solutions. These modern tools are much more than lint with a pretty graphical shell. The best are capable of parsing millions of lines of code and delivering accurate, human-readable results. The tools not only make automated analysis useful, but also will soon make it an essential part of any embedded development process. In other words, modern automated static analysis tools no longer demonstrate their own shortcomings; they now efficiently tell developers what's wrong with the code. Figure 2 shows the new embedded development process with integrated static analysis.

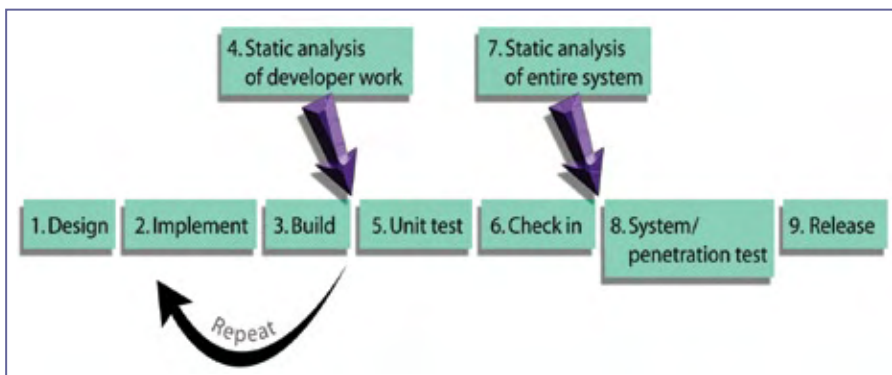


Figure 2

Static analysis solutions deliver more insight

Modern source code analysis tools provide a wide variety of bug and vulnerability detection capabilities. They highlight issues that have a real and measurable impact on the security and quality of an end product, including NULL pointer detection, memory management problems, array bounds violations, concurrency validation and deadlocks, buffer overflows, and tainted data propagation.

Using advanced code path analysis that leverages a complete system view, invalid code paths, inaccessible boundary conditions, and unlikely execution scenarios can all be disqualified from consideration, leaving developers with a scrubbed, viable, and accurate picture of what's actually wrong with the code. Consider the simplified example in Listing 1:

```
void f(int x, int y) {
    int value, *p, *q;
    p = (x & 1) ? NULL : &value;
    if ( p ) *p = 1;
    q = p;
    if (y & 1) *q = 1;
}
```

Code path analysis can easily follow the NULL assignment, aliasing, and subsequent invalid dereferencing that occurs here whenever both x and y contain odd values. But split this into several functions over several modules, and though code path analysis can still spot the problem, traditional methods and low functionality analysis tools may fail to spot the dependency, as shown in Listing 2.

File: m1.c

```
static int value;
int* f(int x) {
    return (x & 1) ? NULL : &value;
}
```

File: m2.c

```
void g(int* ptr, int y) {
    if (y & 1) *ptr = 1;
}
```

File: m3.c

```
int main() {
    g(f(3), 1);
}
```

Now consider that the problems being isolated are not just quality defects of this type but rather subtle security vulnerabilities that could leave the device open to an external attack. Passing maliciously constructed incoming data to an operating

Modern automated static analysis tools no longer demonstrate their own shortcomings; they now efficiently tell developers what's wrong with the code.

system function that causes stack frame corruption because of invalid assumptions is not something that can necessarily be caught in quality assurance. Since most development organizations cannot emulate a motivated hacker's devious mind, automated code analysis tools provide a safety net that will catch problems current tools and processes fail to uncover.

Steps to safer and secure code

To achieve heightened security, developers first must make a fundamental shift in mentality. They must embrace the assumption that software will be under constant attack from malicious systems and hackers. Having a healthy fear of software being compromised – not because of malfunctions or crashes but as a result of coherent, intensive attacks – should be the norm. Remember, the bad guys aren't necessarily out to crash a device. The talented ones want to hijack the system and turn it into an insecure conduit for illegal pursuits.

Building on this healthy and necessary paranoia, developers should complement manual code reviews and other testing activities with automated source code analysis. There simply is not enough time or available resources to manually validate each possible code execution thread. Only automated tools can map the dependencies, plot the execution paths and data flow, and flag anything that might potentially cause a problem in the software or among the various device components.

The chosen code analysis solution must be accurate, fast, and, most importantly, must work the way each individual developer works. For example, a command line tool isn't going to mesh with an integrated development environment user's process. Likewise an Emacs devotee is going to have a hard time with a Visual Studio plug-in. But with the proper tool in hand, the solution can be fully integrated into the development process, allowing developers to analyze and uncover a significant percentage of vulnerabilities and defects before checking in the code.

As soon as a system is connected, it's under attack. But with a new mindset and a modern tool firmly in place, developers will be ready to deliver safer and more secure code than ever thought possible. **ECD**

Gwyn Fisher is Klocwork's CTO. Prior to joining Klocwork, Gwyn served as Senior VP of R&D for LumaPath, where he led the creation of LumaPath's Active Integration product suite, and as Senior VP of R&D for Hummingbird Ltd., where he was responsible for more than 350 employees and five product lines within 11 different locations across the United States, Canada, and Europe. In addition to working as a strategy consultant for many international software firms, Gwyn also held positions at PC DOCS, Inc., Fulcrum Technologies, Ltd., TRIP Systems International, and Paralog Ltd. He has a BS in Computer Science from Royal Holloway, University of London in the United Kingdom.



To learn more, contact Gwyn at:

Klocwork

35 Corporate Drive • Burlington, MA 01803

866-556-2967

gwyn@klocwork.com • www.klocwork.com