print | close

Safety of Critical Connected Devices is Impossible Without Security

Electronic Design
Jay Thomas
Thu, 2016-09-22 10:42

Safety-critical developers traditionally haven't concerned themselves with security. Typically, it's not because it wasn't important to them. Rather, their part was so deeply embedded into a larger system that there was no way to access it or make it do something outside of its intended operation mode.

The world has changed and now these systems are connected, and they—especially control systems—have the ability to not only malfunction, but do great harm if compromised. These systems are increasingly spread across connected hubs, private local networks, the internet, and the cloud. The result is an enormous and ever-expanding attack surface that may include the most innocuous-seeming devices.

Breaching such devices, however, may open the way to more vital and vulnerable systems, e.g., automobiles, power plants, public utilities, and commercial databases. And thus begins a never-ending duel between hackers and security defenders, and what's at stake is often safety and human lives.

Related

Enforcing Programming Standards Protects Against Software Security Risks

Top Five Things to Know About Software Testing

11 Myths About Software Qualification and Certification

There are a number of software process standards, including DO-178C for military and aerospace applications and ISO 26262 in the automotive industry, aimed at reliability and safety. These process standards spell out specifics in terms of mandating traceability between the requirements and code, as well as adequate testing with respect to the requirements.

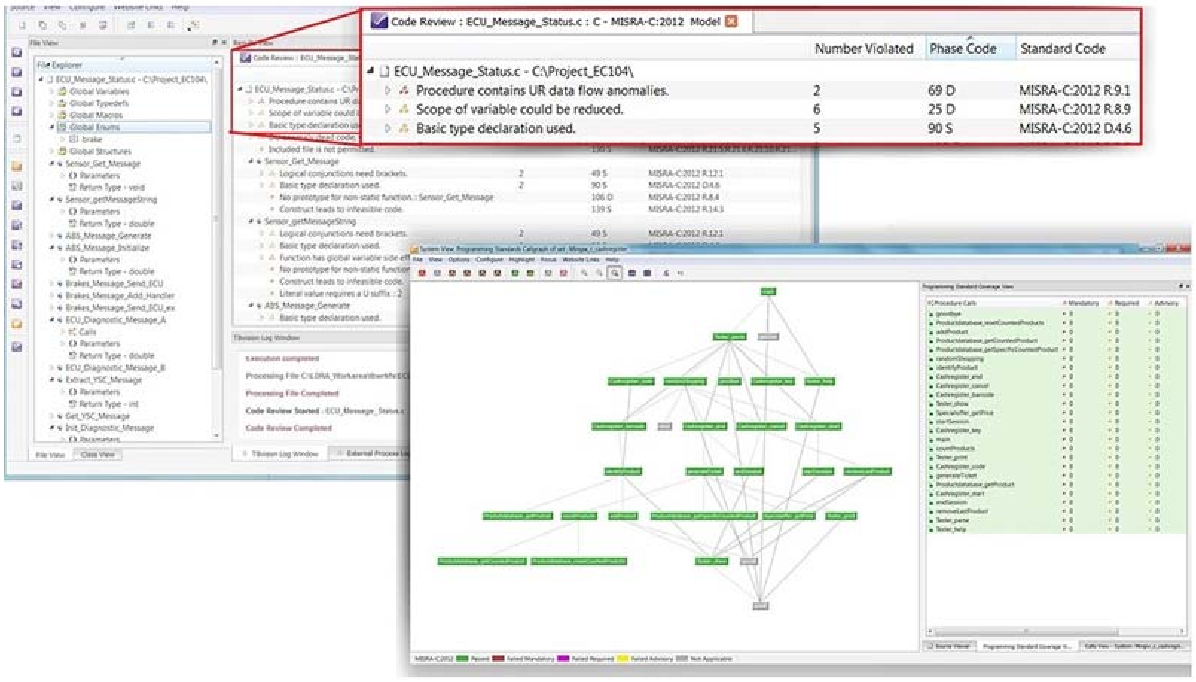
Because DO-178C and ISO 26262 are process standards, they don't explicitly specify which security or coding standards to use. However, DO-178C does require that a coding standard be used, and artefacts are typically provided in terms of use of automated tools to demonstrate compliance. ISO 26262 again recommends use of a coding standard and specifically calls out a set of rules to be used as part of the compliance process. These processes are at the heart of functional safety. To ensure the system is safe and secure, we should draw upon these processes, including any specifically called out rules, and extend them to include security focused rules.

The most attractive and up-to-date C coding standard, and one that addresses both safety and security, was developed for the automotive industry by MISRA. This MISRA C standard is a subset of the C language for developing applications with high-integrity and high-reliability requirements. MISRA guidelines aim to litate code safety, security, portability, and reliability in the context of embedded systems.

Recently, MISRA C:2012 has been enhanced to ensure greater security for embedded systems in general. This push at the coding level recognizes that while software may be functionally correct and reliable, it's neither safe nor reliable if it's vulnerable to outside attack. Although originally developed for the automotive industry, the language subset has over the years gained widespread acceptance for safety-, life-, and mission-critical applications in aerospace, telecom, medical devices, defense, railway, and other industries.

An Evolving Coding Standard

The C language has been evolving, and it's natural that the coding standard will evolve as well. The latest version, MISRA C:2012, contains 143 rules, each of which is now classified as mandatory, required, or advisory. Mandatory rules are rules that must be met, while required rules must also be met or applied with a formal, defined deviation. Advisory rules may be ignored without a formal deviation, but the deviation must be recorded in any documentation. Each rule is checkable using static code analysis. That latter point is vital given the complexity of most code and the detailed requirements of the MISRA rules *(Fig. 1)*.



Code Review : ECU_Message_Status.c : C - MISRA-C:2012 Model

	Number Violated	Phase Code	Standard Code
ECU_Message_Status.c - C:\Project_EC104\			
Procedure contains UR data flow anomalies.	2	69 D	MISRA-C:2012 R.9.1
Scope of variable could be reduced.	6	25 D	MISRA-C:2012 R.8.9
Basic type declaration used.	5	90 S	MISRA-C:2012 D.4.6

One example of a typical mandatory MISRA C:2012 rule involves freeing allocated memory:

Rule 22.2: A block of memory shall only be freed if it was allocated by means of a Standard Library function.

Many systems designed to operate in low-memory environments have their own memory-allocation systems. This often overrides the built-in standard-library functionality and extends the C syntax. While it's technically possible to call "free" on any block of memory, it can be erroneous and cause a runtime failure. Another classic variation is marking the same block of memory as free multiple times. Both may not show up with a debug runtime environment, but can halt system operation during normal operation. However, the MISRA rules see it as a potential danger and disallow the practice *(Fig. 2)*.

```
void fn ( void )
{
int32_t a;
free ( &a ); /* Non-compliant - a does not point to allocated storage */
}
```

Existing rules have been revisited, refined, adjusted, and justified so that MISRA C:2012 enhances the established concept of “rationale” descriptions of why each rule is a good idea. This approach benefits those looking to implement MISRA C:2012 in its entirety, as well as those looking to use it as the basis for in-house standards. Static-analysis tools should be able to check both for MISRA compliance and any additional coding standards defined by the organization. For example:

Rule 12.1: The precedence of operators within expressions should be made explicit.

This replaces a rule from the 2004 language subset that stated: “Limited dependence should be placed on C’s operator precedence rules in expressions.” The improved precision in the wording and in the form of a decision table provided with the new language subset will lead to both users and (importantly) tool developers having a more consistent interpretation.

Though C does have an order of precedence, many operators have the same level of precedence. Therefore, seemingly simple arithmetic can yield different results in different compilers and runtime environments. Examples are also called out, so if you look at the standards document, you will see some clear examples that demonstrate the potential errors of operator precedence.

Security Measures Guard Safety

After the publication of MISRA C:2012, the committee responsible for maintaining the C standard published the ISO/IEC 17961:2013 C language Security Guidelines. Their purpose was to establish a baseline set of requirements for analyzers, including static-analysis tools and C language compilers, to be applied by vendors who wish to diagnose insecure code beyond the requirements of the language standard.

All rules are meant to be enforceable by static analysis. The criterion for selecting these rules is that analyzers implementing them must be able to effectively discover secure coding errors without generating excessive false positives.

In response, the MISRA committee released MISRA C:2012 Amendment 1 to support these new requirements for security. The amendment is an enhancement to, and is fully compatible with, all existing editions of the MISRA language guidelines and becomes the standard approach for all future editions of the MISRA guidelines.

MISRA C:2012 Amendment 1 does for security what the main MISRA standard does for reliability and safety. By helping developers avoid coding practices that can introduce security vulnerabilities and write code that’s more understandable, it lets them more thoroughly analyze their code. This also makes it possible to assure regulatory authorities that they followed safe and secure coding practices. This is becoming particularly critical in industries such as defense, automotive, aerospace, and medical, where security threats have led to stringent requirements by OEMs for developers to prove that their software meets the highest standards for security as well as safety.

Documenting and proving compliance may not yet be required in all application areas. However, the ability to so—especially using cost-effective, automated tools to generate credible reports following established es—can be a competitive advantage for developers.

In addition to the amendment to incorporate the 14 new rules, the MISRA committee also released MISRA C:2012 Addendum 2, which maps the overall coverage by MISRA C:2012 of ISO/IEC 17961:2013 and justifies the viewpoint that MISRA C is equally applicable in a security-related environment as it is in a safety-related one.

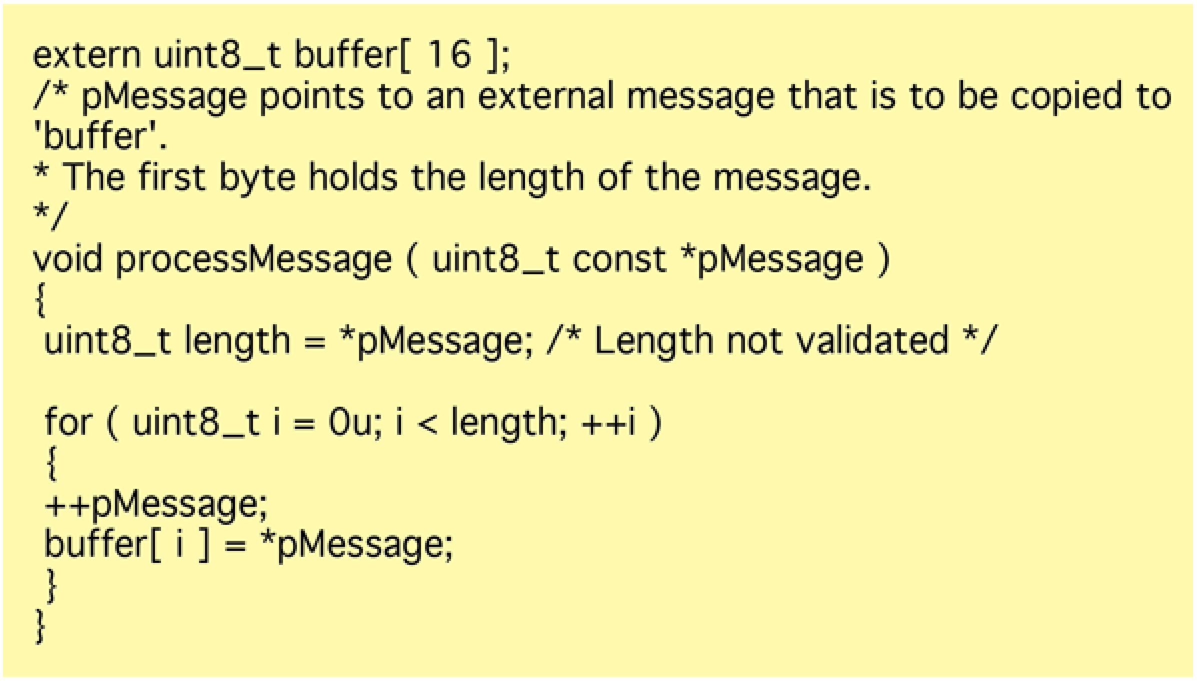
MISRA C:2012 Amendment 1 establishes 14 new guidelines for secure C coding to improve the coverage of the security concerns highlighted by the ISO C Secure Guidelines. Several of these guidelines address specific issues pertaining to the use of "tainted" data—a well-known security vulnerability. A couple of specific examples illustrate common vulnerabilities and how the new guidelines address them.

Example 1: Don't open the door to give out your password

Dir. 14.4: The validity of values received from external sources shall be checked"

This example controls what data can be sent to external sources, which has become critical as more and more devices are connected to each other or to the internet. If code is written incorrectly, internal data can be exposed to external sources. This rule presumes that the system knows certain things about the data that it will be expected to send to an external source, such as length.

The following sample code is dangerous because the length of a message received from an external source isn't validated. Taking this to the logical extreme—by not validating the length of data to send back—an attacker could extract the entire contents of a computer's memory remotely. In practice, variations of this attack have been used to extract cleartext passwords from network servers' internal memory buffers. Recently, it was leveraged to download large volumes of customer financial information, but it could be used to capture almost any type of data *(Fig. 3)*.

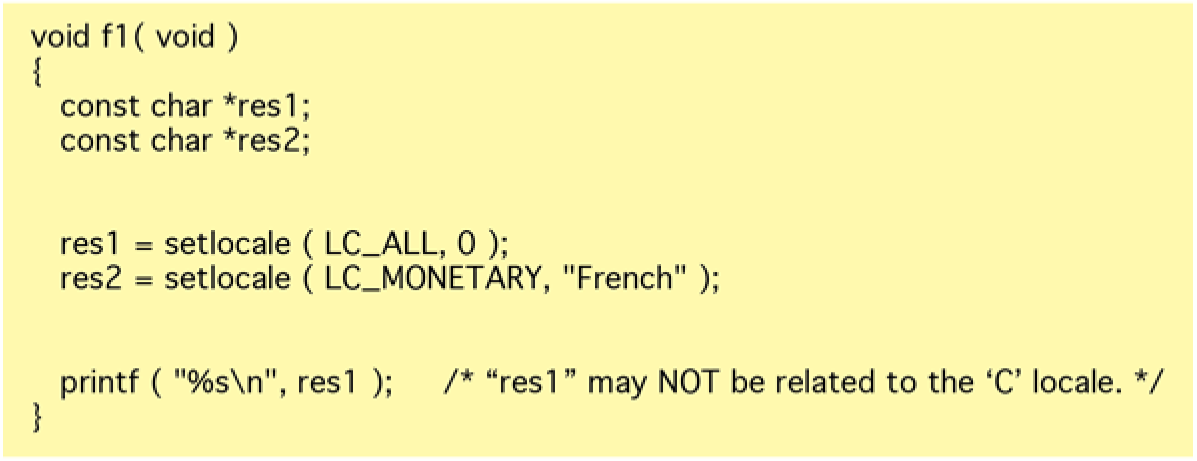

```
extern uint8_t buffer[ 16 ];
/* pMessage points to an external message that is to be copied to
'buffer'.
* The first byte holds the length of the message.
*/
void processMessage ( uint8_t const *pMessage )
{
 uint8_t length = *pMessage; /* Length not validated */

 for ( uint8_t i = 0u; i < length; ++i )
 {
 ++pMessage;
 buffer[ i ] = *pMessage;
 }
}
```

Example 2: If someone can control how you talk, they can make you say anything they want

e 21.19: The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, ıleconv, getenv, setlocale, or strerror shall not be used following a subsequent call to the same function.

Local messages control the formatting of strings—effectively defining how the software “talks.” If hackers can take control of how strings are formatted, they can make them say whatever they want, including controlling whatever programs the first program interacts with. That means these strings can be used to execute arbitrary commands and take over a system.

The following sample code may not work as expected, because the second call to setlocale may lead to the string referenced by “res1” being the same as the one referenced by “res2.” Beyond correctness, controlling locale has been used in exploits where strings have been reformatted to facilitate the execution of commands on a remote computer *(Fig. 4)*.



```
void f1( void )
{
  const char *res1;
  const char *res2;


  res1 = setlocale ( LC_ALL, 0 );
  res2 = setlocale ( LC_MONETARY, "French" );


  printf ( "%s\n", res1 );    /* “res1” may NOT be related to the ‘C’ locale. */
}
```

Assuring security, especially for small, connected devices, is an ongoing battle. As technology advances, hackers seem to be able to find new avenues of vulnerability. Developers must also quickly detect these and form strategies to counter them.

For all of this, it’s still necessary to ensure that these strategies are also correctly coded—both in terms of coding standards and in terms of correct functionality in the overall application. Transfer protocols such as transport layer security (TSL)—which is an improvement over the secure sockets layer (SSL), the secure file transfer protocol (SFTP), and other protocols—are now widely used, but are often acquired from outside the organization.

Other strategies include the use of secure device drivers. Strategies are also in play for remote implementation of secure and encrypted firmware upgrades, let alone personal verification strategies such as passwords, retina scans, and RFID chips to secure access. Other layered security strategies allow only selected access to parts of the system. But using these—either by importing them or writing them from scratch—can also introduce flaws that are able to be exploited if not detected. You have to know that your clever security strategy really works as intended.

It’s imperative that security not be an afterthought; it must be planned and implemented from the ground up. As a result, in addition to the traditional development tools, other methods such as static and dynamic analysis should be applied from the beginning, along with requirements traceability for coverage analysis and unit integration and testing. Adherence to coding standards is a vital part of the overall system test and analysis operation, which is increasingly needed to assure reliable, safe, and secure operation as well as meet requirements for certification in a range of application areas.

mming It Up

As developers continue to add features to their products within already constrained budgets and schedules, software has become the weak link that allows malicious entities to gain access to sensitive data and take over systems. Thus, writing secure code has become essential, even for non-safety-critical systems.

Use of the MISRA C language standard, including the MISRA C:2012 Amendment 1, helps developers write safer, more secure, and more maintainable code. With the aid of appropriate checking tools, checking security becomes a cost-effective way of increasing code quality and reducing defects down the road. Any time you spend up front to make sure your code is safe and secure is saved many times over when your code works as you expect.

Source URL: http://electronicdesign.com/dev-tools/safety-critical-connected-devices-impossible-without-security