Securing Connected Systems

Chris Tapp, LDRA

The adoption of networking in embedded systems is widespread and includes everything from domestic audio/video systems to SCADA (supervisory control and data acquisition) systems. Users are generally aware that viruses, worms and malware are attempting to infiltrate their personal computers, but most are totally oblivious to the fact that the devices and infrastructure they have come to rely on in their everyday lives are also under similar attack; they simply see these devices as “black boxes” and do not appreciate that they often contain complex software systems. Because of this, these systems need to be hardened so that they are impervious to attack without relying on the user “doing the right thing” to protect them.

The security of these networked devices is therefore a major concern, and there are two main aspects that need to be considered during their implementation and deployment. Firstly, the physical network itself needs to be secure. Since network security is a “standard” infrastructure issue, it will not be considered further here. This article will focus on the second issue, that of security within the firmware of the devices themselves. It will show how the security of this firmware can be enhanced by applying the guidelines contained within the “CERT C Secure Coding Standard.”

During 2008 there had been a great deal of positive talk about the emergence of a new breed of programming guidelines from the U.S. federally funded organization, CERT. Languages such as C, C++ and Java are being tackled, with the goal of producing safe, secure, and reliable systems. Currently, though, it is CERT C that has made a splash. In October 2008, version 1.0 of the standard made its debut at the Software Development Best Practices exhibition in Boston.

Both C++ and Java secure coding standards are a work in progress at present, but for this discussion, we focus entirely on the CERT C Secure Coding Standard.

What is CERT C and where did it come from?

CERT was created by DARPA (Defense Advanced Resource Projects Agency) in November 1988 in response to the damage caused by the Morris Worm. Its coordination center (CERT/CC) is located at Carnegie Mellon University’s Software Engineering Institute (SEI).

Although intended purely as an academic exercise to gauge the size of the Internet, defects within the code lead to damaging denial of service events. This unintentional side-effect of the Morris Worm had repercussions throughout the worldwide Internet community and thousands of machines owned by many organizations were infected.

Consequently, software vulnerabilities came under the microscope of various bodies, including the U.S. government. The SEI CERT/CC was primarily established to deal with Internet security problems in response to the poor perception of its

security and reliability attributes. Over a period of 12-15 years the CERT/CC studied cases of software vulnerabilities and compiled a database of them. The Secure Coding Initiative, launched in 2005, used this database to help develop secure coding practices in C.

Purpose and aims of CERT C

The CERT C Secure Coding Standard provides guidelines for secure coding in the C programming language. Following these guidelines eliminates insecure coding practices and undefined behaviors that can lead to exploitable software vulnerabilities. Developing code in compliance with the CERT C secure coding standard leads to higher quality systems that are robust and more resistant to attack.

Network connectivity is clearly a primary source of malicious attacks on software systems. These attacks are now focusing on the components that provide, distribute and consume the associated network traffic. A dependency on connected software systems is not just relevant to corporations or individuals, but also government and civil infrastructure (industrial plants, power generation and transmission, etc.). More and wider connectivity is becoming integral to the way people work and live. There is a need, therefore, for systems to be impenetrable, whether from IT-based hackers or from devices and equipment under the control of those with malicious intent. On this basis, CERT C considers the overall and far-reaching need for secure coding practices.

The primary aim of CERT C is to enumerate the common errors in C language programming that lead to software defects, security flaws, and software vulnerabilities. The standard then provides recommendations about how to produce secure code. Although the CERT guidelines shares traits with other coding standards, such as identifying non-portable coding practices, the primary objective is to eliminate vulnerabilities.

And so, what is software vulnerability? The CERT/C describes a “vulnerability” as a software defect which affects security when it is present in information systems. The defect may be minor, in that it does not affect the performance or results produced by the software, but nevertheless may be exploited by an attack from an intruder that results in a significant breach of security. CERT/C estimates that up to ninety percent of reported security incidents result from the exploitation of defects in software code or design.

The aim of the Secure Coding Initiative is to work with developers and their organizations to reduce the number of vulnerabilities introduced into secure software by improving coding practices through the provision of guidelines and training. To this end, one of the collaborations CERT has formed is with the SANS (SysAdmin, Audit, Network, Security) Institute, a leading computer security training organization.

What security issues are there with networked devices?

The problems can be broadly classified in to two groups based on the content that is received over the network.

Firstly, the device can react incorrectly to valid data, possibly because it contains forged or spoofed security credentials. Information flow analysis can help to identify regions of code which do not adequately validate such data.

Secondly, the device can react inappropriately when it receives invalid data. Such failures are often associated with data and/or information flow errors within the code, with the incorrect handling of error conditions leading to unexpected control flow path execution as the data is processed. These paths may result in the incorrect initialization of variables if the expected paths are not taken, often due to assumptions that have been made about the expected data values.

Incorrect processing of input data can lead to stack or buffer overrun errors, with the risk that this leads to the execution of arbitrary code injected as part of a deliberate attack on a system. It is common for the initial processing of network data to take place within the kernel of an operating system. Thus means that any arbitrary code execution is likely to take place at an elevated privilege level (akin to having “superuser” or “admin” rights) within the system, giving it the potential to access all data and/or hardware regardless of whether it is related to the communications channel that was used to initiate the attack. From this, it is obvious that any vulnerability in low-level communications software may lead to the exposure of sensitive data or allow the execution of arbitrary code at an escalated privilege level.

It is possible for an external agent to deliberately target and exploit vulnerabilities within a networked system. This could allow eavesdropping of sensitive data, denial of service attacks, redirection of communications to locations containing malware and/or viruses, man-in-the-middle attacks, etc. In fact, a quick Internet search will show that organizations such as CERT, SANS, etc. have many reports of network vulnerabilities that facilitate such attacks. These include integer overflows leading to arbitrary code execution and denial of service, and specially crafted network packets allowing remote code execution with kernel privileges and local information disclosure.

While many of the identified vulnerabilities are eliminated from future code releases, it is common for deployed systems not to benefit from this work via updates. PC drivers are generally (at least within corporate environments) updated as part of the rollout of routine security updates. However, embedded devices are often “ignored” as the updates are often more difficult to apply and some devices are simply “forgotten about.”

Does it really happen?

A recent case has come to highlight the importance of embedded security. In June of 2010 a new worm (now known as “Stuxnet”) was detected infiltrating the SCADA networks of a well-known industrial control system.

A hard-coded password (a common practice in more than half of the control systems) used within a Windows component of the system leaked on to the Internet creating a known vulnerability. This was not, in itself, considered a high-priority

issue as it could only be exploited from the local system which used an internal network with no external connections. It was only when a suitable attack vector was identified by attackers that this vulnerability became exploitable. Stuxnet provided this vector.

A USB pendrive containing Stuxnet was created to exploit a vulnerability within Windows shortcut (.lnk) files to install rootkit-like software on an infectable system (which, at the time, was any Windows system). The virus was then able to install two low-level device drivers that were accepted as trusted because they appeared to have been signed using the digital certificate of a well-known semiconductor supplier. Once the initial system was infected, the worm was able to spread over the internal networks and exploit the vulnerability within the SCADA system.

Both of the vulnerabilities associated with the attack have now been fixed, but users of the SCADA system have been warned that cleanup of any infections can, in the short term, cause more disruption than the infection itself.

How does CERT C help?
The CERT C guidelines define rules that must be followed to ensure that code does not contain defects which may be indicative of errors and which may in turn give rise to exploitable vulnerabilities. For example, guideline EXP33-C says “Do not reference uninitialized memory,” protecting against a common programming error (often associated with the maintenance of complex code). Consider the following example:

```
int SignOf ( int value )
{
int sign;

if ( value > 0 )
{
sign = 1;
}
else if ( value < 0 )
{
sign = -1;
}

return sign;
}
```

The *SignOf()* function has a UR dataflow anomaly associated with *sign*. This means that, under certain conditions (i.e. if *value* has a value of ‘0’), *sign* is Undefined before it is Referenced in the *return* statement. This defect may or may not lead to an error. For example:

```
int MyABS ( int value )
{
return ( value * SignOf ( value ) );
```

```
}
```

MyABS() will work exactly as expected during testing, even though the return value of *SignOf()* will contain whatever value was in the storage location allocated to *sign* when it is called (zero multiplied by any integer value is zero). The code is said to be “coincidentally correct” as it works, even though it contains a defect.

However, all uses may not be so fortunate:

```
void Action ( int value )
{
if ( -1 == SignOf ( value ) )
{
NegativeAction ( );
}
else
{
PostiveAction ( );
}
}
```

The code in *Action()* is most likely to call *PositiveAction()* when it is called with a value of ‘0’ (which is likely to be correct), as only a return value of (exactly) ‘-1’ will cause it to do otherwise. It is likely that a latent defect will be present in released code as there is little chance of it causing an error during testing, even if that testing is robust.

Such an error could potentially be manipulated by an attacker to invoke the incorrect behaviour (e.g. by causing the stack on which *sign* is created to be pre-loaded with ‘-1’).

While compliance with the CERT C guidelines can, in theory, be demonstrated by manual checking, this is not practical for large or complex systems. To that end, tools are available to automate the compliance checking process. Figure 1 shows an example of how the LDRA tool suite reports the dataflow anomaly in the *SignOf()* function.

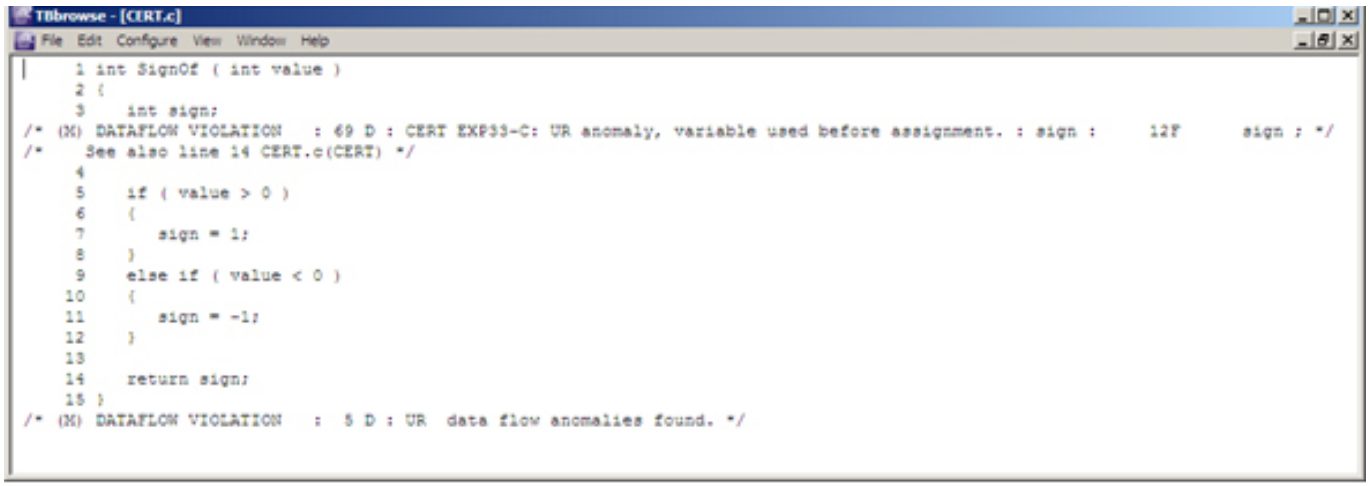

Figure 1 - LDRA Testbed detection of EXP33-C violation

Automated tool checking for compliance is a cost-effect and efficient method to demonstrate compliance with the requirements of CERT C. Such a demonstration provides evidence to support any claims that are made with respect to the inherent security of a networked product.

Conclusions

Adoption of the guidelines contained within CERT C will significantly reduce the number of vulnerabilities within networked systems. However, these benefits will only be realized if tools that automatically check code for compliance with the standard are available to the developers.

CERT launched the Vulnerability Discovery Project, which promotes the use of test tools and techniques, to help ensure that suitable tools are available. The aim is to develop a process which may be used systematically by developers and testers to discover and eliminate all vulnerabilities in software. As part of their vision, the project hopes to encourage and assist tool vendors to include support in their tools for the CERT C Secure Coding Standard.

The author

Chris Tapp is a Field Applications Engineer at LDRA with more than 20 years experience of embedded software development. He graduated from the University of Durham in 1987 and has spent most of his career working within the automotive, industrial control and information technology industries, mainly as a self-employed consultant. He has been involved with MISRA since 2001 and is currently chairman of the MISRA C++ working group. He has been with LDRA since 2007, where he specializes in programming standards.