Protecting ICS Software With Secure Coding Practices

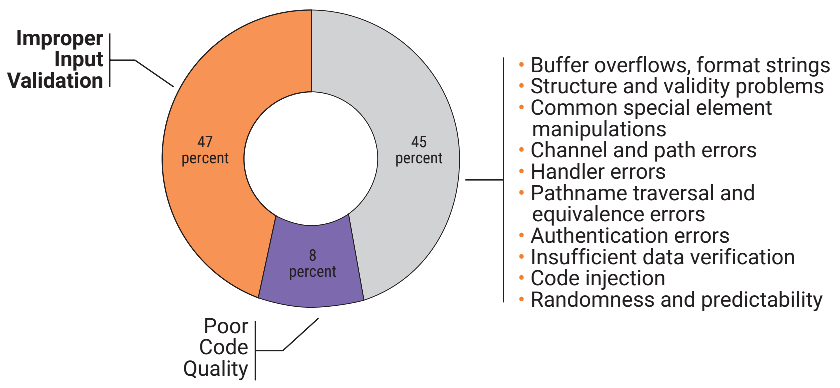
Industrial control systems (ICSs) are cyberphysical information systems used to control a wide variety of industrial processes ranging from manufacturing, product handling and production to distribution of utilities and monitoring of transportation. Industrial control systems include supervisory control and data acquisition (SCADA) systems used to control geographically dispersed assets as well as distributed controls systems (DCSs) and smaller control systems using programmable logic controllers (PLCs) to control localized processes. Regardless of the type and application of an ICS, software is the central component in controlling and monitoring critical infrastructure and industries around the world, including the distribution of electricity and water, the manufacturing of volatile chemicals, and the safe operation of mass transit.[1] Failure to protect this software from cyberattacks can lead to serious global consequences ranging from economic disruptions to environmental damage and the loss of human life. Therefore, it is essential that such software be secure and reliable.

Software vulnerabilities in general have increased as much as 70 percent year over year in the period between 2000 and 2010.[2, 3] Today's ICS software incorporates many common off the shelf (COTS) and open-source elements (e.g., Windows operating systems and modern programming frameworks such as Java and .Net) but operational constraints often prevent the incorporation of security patches and version upgrades. Thanks to exposure to the Internet, an increase in cyberattacks and the extensibility of modern programming frameworks, ICS software is particularly vulnerable (**figure 1**), making the adoption of secure coding procedures critical.[4]

Developers can no longer depend on the existence of ideal circumstances for the execution of their code due to platform mobility and continually evolving system environments.[5] They must accept that risk is pervasive and adopt secure coding practices that align with the overall software development life cycle, resulting in clear security strategies or security development life cycles.[6, 7, 8] This strategy must include defining security use cases,[9, 10] predicting threats,[11, 12] and performing risk analyses before the implementation of any specific coding practices.[13, 14] In 2011, the US Department of Homeland Security (DHS) found that almost half of all vulnerabilities identified in ICS software could be addressed by a single secure coding technique.[15] The adoption of

FIGURE 1

Sources of ICS Software Vulnerabilities



Sources: Adapted from Michard, I.; "How Secure Equipments in Your ICS Network Need to Be? An Approach to Select the 'Just Secure Enough'," 46th Annual Institute of Electrical and Electronics Engineers (IEEE)/International Federation for Information Processing (IFIP) International Conference on Dependable Systems and Networks, 2016, Toulouse, France, *https://hal.archives-ouvertes.fr/hal-01318167*; US Department of Homeland Security, *Common Cybersecurity Vulnerabilities in Industrial Control Systems*, USA, May 2011, *https://www.cisa.gov/uscert/sites/default/files/recommended_practices/DHS_Common_Cybersecurity_Vulnerabilities_ICS_2010.pdf;* and Common Weakness Enumeration (CWE), "Viewing Customized CWE Information," *https://cwe.mitre.org/index.html*

MATTHEW J. SCOTT | CICP, CM, LEAN SIX SIGMA BLACK BELT

Is a technology leader at a regional high-capacity public transit agency serving the greater Seattle, Washington, USA, area. He delivers innovative solutions addressing the challenges of operating, maintaining and securing legacy industrial control systems for high-demand transit assets, and combines traditional industrial automation with IT deployment strategies. Through industry and market research, road maps and engagement with customers and fellow automation builders, he creates a close, team-oriented problem-solving environment, resulting in improvements to industrial control systems software security, reliability and quality while respecting the constraints inherent in critical infrastructure.

secure coding practices for ICS software ensures secure and reliable systems, regardless of their surroundings.[16]

Securing ICS software requires a strategic approach. Simply applying secure coding practices during or after the writing of code (i.e., defense coding) is a key part of ICS software security, but it is not enough because it results in only incremental improvements in security in the absence of a comprehensive strategy.[17, 18, 19] The best way to secure ICS software is to apply a security perspective throughout the software development life cycle—from design to code development through verification and operation.[20, 21] Applying a three-step process for ICS secure coding can facilitate more resilient protection of critical infrastructure through ICS software with security by design, not merely as an afterthought.

Security Development Life Cycle

The security development life cycle (**figure 2**) is a result of applying a security perspective based on lessons learned to the software development life cycle—specifically, to the requirements, design, coding, and testing and operation stages of software development. This is what the software engineering profession labels security by design.[22]

Requirements

Throughout requirements development, each functional feature (e.g., control loop, alarm handling procedure, controlled shutdown routine) of the software identified must be evaluated from a security perspective. Although many functional features defined in the requirements stage of software development are routine and repetitive, every instance of a functional feature can have both apparent and nascent vulnerabilities that necessitate an explicit set of security requirements. The development of security use cases—or abuse cases—that define how the system will react during a security exploitation exposes nascent vulnerabilities.[23] These security use cases determine the levels of protection required to maintain the security of each instance of software functional features, and the relevance of security for each instance depends on how important it is to the overall function of the system and the access required by both internal and external system actors. Throughout the security requirements stage, all functional system threats and vulnerabilities must be identified and listed to enable threat modeling in the design stage.[24]

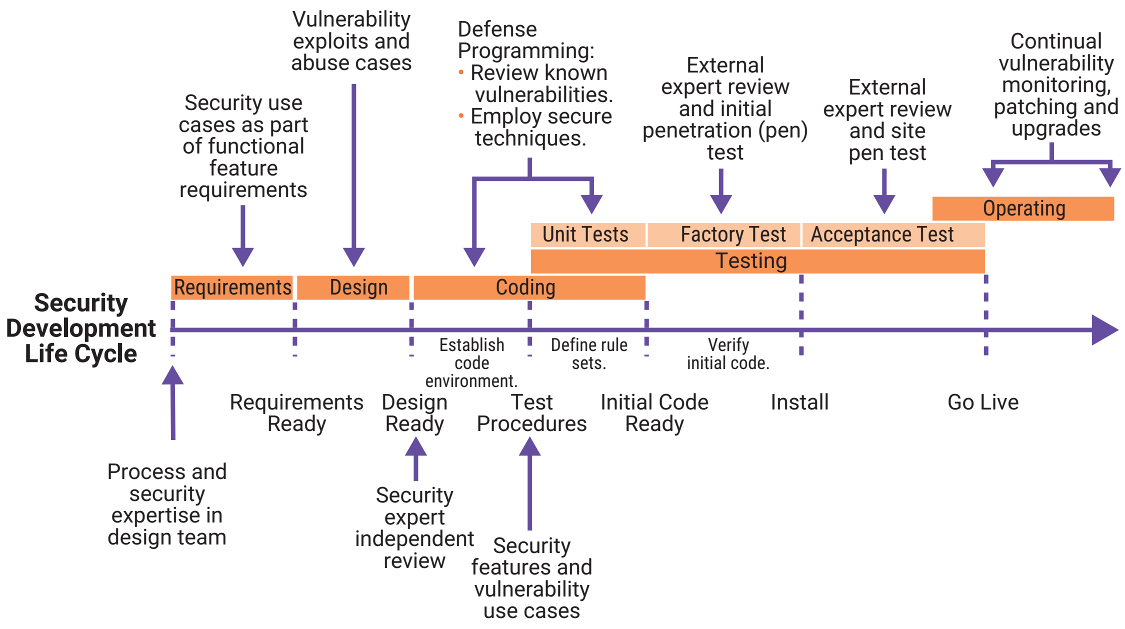
Design

The design stage of the security development life cycle, like the software design life cycle, is where success or failure is ultimately determined. Although some threats and vulnerabilities may have been identified in the security requirements stage, it is possible that others exist.[25] Security errors or omissions occurring in the design stage are unlikely to be successfully resolved in later stages. Even if they are resolved, the costs of rework—including redevelopment and retesting, and possible loss of system availability—increase with each successive stage until the errors are finally addressed. In the design stage, the threats to the system and associated vulnerabilities are identified and defined. Feedback from actual operational security exploitations must be included in the design exercise. Threat modeling and risk analysis ensure that the design is free from the root causes of previous exploitations. Design complexity is an indicator of potential vulnerabilities; thus, the design stage seeks to lessen complexity to ensure security.[26] The design stage produces specifications and documentation to guide the

Throughout requirements development, each functional feature…of the software identified must be evaluated from a security perspective.

FIGURE 2

Security Development Life Cycle



Vulnerability exploits and abuse cases
Defense Programming:
• Review known vulnerabilities.
• Employ secure techniques.
External expert review and initial penetration (pen) test
External expert review and site pen test
Continual vulnerability monitoring, patching and upgrades
Security use cases as part of functional feature requirements
Operating
Unit Tests
Factory Test
Acceptance Test
Testing
Requirements
Design
Coding
Security Development Life Cycle
Establish code environment.
Define rule sets.
Verify initial code.
Requirements Ready
Design Ready
Test Procedures
Initial Code Ready
Install
Go Live
Process and security expertise in design team
Security expert independent review
Security features and vulnerability use cases

Source: Adapted from Howard, M.; D. Leblanc; *Writing Secure Code, 2nd Edition*, Microsoft, USA, 2002, *http://www.microsoft.com/MSPress/books/5957.aspx9780735617223*

selection and use of coding methods and procedures that ensure a level of security that matches the functionality and features of the system. The resultant threat modeling and risk analysis then feed into the testing stage to verify security against specific threats and vulnerabilities, in addition to standard testing.[27]

Coding

The security development life cycle enhances the coding stage of the software development life cycle with the use of specific coding techniques and procedures.[28] Original equipment manufacturers (OEMs) and developers in the ICS industry label this "defense programming."[29] These programming and coding practices are based on the results of the requirements and design stages of the security development life cycle.[30] Among defensive programming techniques are:

- Simple validation tests of all field equipment input signals for out-of-range values
- Rate-of-change comparisons of control output signals or remote setpoints (randomness and predictability)
- More complex error-checking of values resulting from a calculation for register overflows resulting in rounding errors
- Authentication functions to identify uncharacteristic messages from networked PLCs or supervisory system servers (equivalence and privilege)

The specific defensive programming techniques utilized derive from the best mitigation of the abuse cases identified in the requirements and design phase.

Testing and Operation

There are two steps to the testing stage of the security development life cycle. First, standard security testing is conducted. Then additional testing based on the risk analysis and threat modeling from the design stage is conducted. Specific threats are simulated, and all testing traces back to security requirements and security use or abuse cases. Penetration testing (pen testing) is part of this stage. It is conducted first in the factory acceptance test (FAT) where the production environment is simulated using artificial inputs or forcing logic completion. While this test is conducted in a controlled and offline environment, external experts are utilized to review the code execution and conduct realistic attempts to exploit vulnerabilities.

Following shipment and installation, pen testing is conducted a second time in the operational environment. Real inputs are utilized with production

Because it controls much of today's critical infrastructure, it is essential that ICS software be developed using secure coding practices.

assets operating, and vulnerability exploitation attempts are conducted from outside the system and facility utilizing live network interfaces and operating telecommunications if indicated. All pen testing is guided by the risk analysis output from the design stage, and it is the best indicator of the actual performance of the secure coding practices employed.

The security development life cycle does not end when the software becomes operational. All actual exploitations in operational systems must be documented. The details of these exploitations provide feedback that can be used in threat modeling for software upgrades or new deployments of software.

Secure Coding Practices

Secure coding practices are the heart of the security development life cycle. However, they must align with the results of the requirements and design stages to ensure that the selected practices will secure the specific software for specific purposes. There are three steps in the employment of secure coding practices:[31]

1. Establish the security coding environment.
2. Define the appropriate rule sets.
3. Verify the initial code.

Establishing the Security Coding Environment

This first step is project- or function-specific. First, collect the security requirements identified during the design stage. Then select the code base. The software functional feature security requirements are key factors in the selection of a code base.[32] Once the code base has been selected, it must be evaluated for known vulnerabilities and weaknesses. International security standards such as the Open Web Application Security Project (OWASP),[33] Common Weakness Enumeration (CWE)[34] and US National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD)[35] are repositories of code base and architectural vulnerabilities and weaknesses found by code developers around the world. Consulting these repositories can help teams eliminate software and architectural vulnerabilities outright or, if a software functional feature demands use of a weak element, apply industry verified mitigations (workarounds or preventive measures) to alleviate the vulnerability. The final step is to select software analysis tools to verify the code. These tools must be capable of configuration for the specific code base and established coding rules. Selecting these tools in the initial step prevents the discovery of a lack of capability during final verification.

Defining the Appropriate Security Rule Sets

This step ensures the fulfillment of common security requirements using common identical coding. It requires the cataloging of security requirements by type, including software functional feature characteristics and risk. The master rule set catalog controls all coding activities. All coders should be familiar with the master rule set catalog prior to commencing any coding. Configuration of the verification tool occurs with the rule sets for use in scanning the initial code.

Verifying the Initial Completed Code

The preconfigured verification tool scans all code to determine adherence to the master rule sets. There may be architectural elements or custom security controls that the verification tool cannot analyze. Therefore, a secure coding expert must manually review all ICS software.

Conclusion

Because it controls much of today's critical infrastructure, it is essential that ICS software be developed using secure coding practices. For secure coding practices to be effective, a security development life cycle is required. This security development life cycle expands and enhances the software development life cycle by ensuring an explicit focus on security during the requirements, design, coding, and testing and operation stages. The result is ICS software that is secure by design.

Endnotes

1 Michard, I.; "How Secure Equipments in Your ICS Network Need to Be? An Approach to Select the 'Just Secure Enough'," 46th Annual Institute of Electrical and Electronics Engineers (IEEE)/ International Federation for Information Processing (IFIP) International Conference on Dependable Systems and Networks, 2016, Toulouse, France, *https://hal.archives-ouvertes.fr/hal-01318167*

2 McGraw, G.; "Building Secure Software: Better Than Protecting Bad Software," *IEEE Software*, vol. 19, iss. 6, 2002, p. 57–58, *https://doi.org/10.1109/MS.2002.1049391*

3 Chowdhury, I.; M. Zulkernine; "Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities," *Journal of Systems Architecture*, vol. 57, iss. 3, 2011, p. 294–313, *https://doi.org/10.1016/j.sysarc.2010.06.003*

4 *Ibid*.

5 Howard, M.; D. Leblanc; *Writing Secure Code, 2nd Edition*, Microsoft, USA, 2002, *https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223*

6 *Op cit* Michard

7 Howard, M.; S. Lipner; *The Security Development Lifecycle*, Microsoft Press, USA, 2006, *https://doi.org/10.1109/tim.1969.4313808*

8 Mohaddes Deylami, H.; I. Ardekani; R. C. Muniyandi; H. Sarrafzadeh; "Effects of Software Security on Software Development Life Cycle and Related Security Issues," *International Journal of Computational Intelligence and Information Security*, vol. 6, iss. 8, 2015

9 *Op cit* Howard and Leblanc

10 *Op cit* Chowdhury and Zulkernine

11 Chowdhury, I.; M. Zulkernine; "Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities," *Journal of Systems Architecture*, vol. 57, iss. 3, 2011, p. 294–313, *https://doi.org/10.1016/j.sysarc.2010.06.003*

12 Shin, Y.; L. Williams; "An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics," ESEM '08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 2008, p. 315–317, *https://doi.org/10.1145/1414004.1414065*

13 *Op cit* McGraw

14 *Op cit* Howard and Leblanc

15 *Op cit* Michard

16 *Op cit* Howard and Leblanc

17 *Ibid*.

18 *Op cit* Michard

19 *Op cit* Howard and Leblanc

20 *Ibid*.

21 *Op cit* Epstein

22 *Op cit* Howard and Leblanc

23 *Op cit* Epstein

24 *Op cit* Howard and Lipner

25 *Op cit* Chowdhury and Zulkernine

26 *Ibid*.

27 *Op cit* Epstein

28 *Ibid*.

29 *Op cit* Michard

30 *Op cit* Howard and Leblanc

31 *Op cit* Michard

32 *Op cit* Howard and Lipner

33 The Open Web Application Security Project (OWASP), *https://owasp.org*

34 Common Weakness Enumeration (CWE), *https://cwe.mitre.org*

35 National Vulnerability Database (NVD), National Institute of Standards and Technology (NIST), USA, *https://nvd.nist.gov*