California
DEPARTMENT OF TECHNOLOGY
Office of Information Security

(916) 445-5239
HTTPS://CDT.CA.GOV/SECURITY/

SECURE SOFTWARE
INFORMATION SHEET NO. 1
MAY 2017 (REVISED)

SECURE CODING PRACTICES

1. **Adopt a secure coding standard.** Develop and/or apply a secure coding standard for your target development language and platform.
2. **Define security requirements.** Identify and document security requirements early in the development life cycle and make sure that subsequent development artifacts are evaluated for compliance with those requirements. When security requirements are not defined, the security of the resulting system cannot be effectively evaluated.
3. **Architect and design for security policies.** Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
4. **Keep it simple.** Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.
5. **Adhere to the principle of least privilege.** Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach limits the damage that can result from inadvertent error, unauthorized use, and reduces the opportunities an attacker has to execute arbitrary code with elevated privileges.
6. **Default deny.** Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted.
7. **Validate input.** At a minimum, validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files.
8. **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.
9. **Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of

SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.

10. **Model threats.** Use threat modeling to anticipate the threats to which the software will be subjected. Threat modeling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases.
11. **Practice defense in depth.** Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment.
12. **Use effective quality assurance techniques.** Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Penetration testing, fuzz testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions.

The following organizations provide additional information and resources for establishing secure coding practices:

- *Build Security In* at https://buildsecurityin.us-cert.gov/
- *Secure Coding Standards* at http://www.cert.org/secure-coding/standards/index.cfm
- *Open Web Application Security Project* at https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

CREDIT AND ACKNOWLEDGEMENTS:

- Seacord, Robert, et al, Top 10 Secure Coding Practices, (last edited 3/27/08), U.S. CERT
- Cited References:
  - [Saltzer 74] Saltzer, J. H. “Protection and the Control of Information Sharing in Multics.” *Communications of the ACM 17*, 7 (July 1974): 388-402
  - [Saltzer 75] Saltzer, J. H. & Schroeder, M. D. “The Protection of Information in Computer Systems.” *Proceedings of the IEEE 63*, 9 (September 1975), 1278-1308
  - [Seacord 05] Seacord, R. *Secure Coding in C and C++*. Upper Saddle River, NJ: Addison-Wesley, 2006 (ISBN 0321335724)
  - [Swiderski 04] Swiderski, F. & Snyder, W. *Threat Modeling*. Redmond, WA: Microsoft Press, 2004