Top 10 Secure Coding Practices

Top 10 Secure Coding Practices

1. **Validate input.** 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 [Seacord 05].
2. **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws.
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 [Saltzer 74, Saltzer 75]. 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. **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 [Saltzer 74, Saltzer 75].
6. **Adhere to the principle of least privilege.** Every process should execute with the the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75].
7. **Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems [C STR02-A] 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.
8. **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 [Seacord 05].
9. **Use effective quality assurance techniques.** Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration 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 [Seacord 05].
10. **Adopt a secure coding standard.** Develop and/or apply a secure coding standard for your target development language and platform.

Bonus Secure Coding Practices

1. **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.
2. **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 [Swiderski 04].

Bonus Photograph

We like the following photograph because it illustrates how the easiest way to break system security is often to circumvent it rather than defeat it (as is the case with most software vulnerabilities related to insecure coding practices).

The photograph depicted a street (named Konsequenz) in the University Bielefeld, Germany, at lat/long. 52.036818, 8.491467. It is visible via Google Street View.

We don't know who took this photograph. If you do, please let us know in the comments!

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.