Knowledge for Software Security

Sean Barnum and Gary McGraw
Cigital

A critical challenge facing software security today is the dearth of experienced practitioners. Approaches that rely solely on apprenticeship as a method of propagation won't scale quickly enough to address this burgeoning problem, so as the field evolves and establishes best practices, knowledge management can play a central role in encapsulating and spreading the emerging discipline more efficiently. This article is about the kinds of security knowledge that can provide a solid foundation for software security practices.

Experience, expertise, and security

Knowledge is more than merely a list of things we know or a collection of facts. Simply put, information and knowledge aren't the same thing, and it's important to understand the difference. Knowledge is information in context. A checklist of potential security bugs in C and C++ is information; that same information built into a static analysis tool is knowledge.

Software security practitioners place a premium on knowledge and experience. In a field in which most practitioners are still learning the basics (think checklists and basic coding rules), the value of master craftsmen who have "been there, done that," learned several lessons the hard way, and are able to transfer their experience to others is extremely high.

The bad news is that software security doesn't have enough master craftsmen to effectively apprentice and train the world's developers, architects, and software security newbies. But the good news is that critical software security knowledge and expertise can be compiled and shared widely. This possibility yields a potentially higher return than the pervasive one-to-one method of mentoring practiced today. Through the aggregation of knowledge from several experienced craftsmen, knowledge management can give a new software security practitioner access to the knowledge and expertise of all the masters, not just one or two.

Software security knowledge is multifaceted and applicable in diverse ways. As the software life cycle unfolds, for example, we can directly and dynamically apply security knowledge through the use of knowledge-intensive best practices. During professional training and resource development, we can draw on it for pedagogical application, and during academic training, it can inform basic coding and design curricula.

Security knowledge: A unified view

For reasons of clarity and ease of application, we can organize security knowledge according to the architecture introduced in the "Software security unified knowledge architecture" sidebar. We start by grouping seven knowledge catalogs (principles, guidelines, rules, attack patterns, historical risks, vulnerabilities, and exploits) into three knowledge categories (prescriptive, diagnostic, and historical), all described in more detail in the sidebar.

Two of the seven catalogs are likely to be familiar to software developers who possess even just a passing familiarity with software security—vulnerabilities and exploits. These catalogs have been in common use for years, and have even resulted in collection and cataloging efforts that serve the security community at large, including the CVE and Bugtraq. Similarly, principles (stemming from Jerome Saltzer and Michael Schroeder's seminal work[1]) and rules (identified and captured in static analysis tools such as ITS4[2]) are fairly well understood. More recently identified knowledge catalogs include guidelines (often built into prescriptive frameworks for technologies such as .NET and J2EE), attack patterns,[3] and historical risks. Together, these various knowledge catalogs provide a basic foundation for a unified knowledge architecture that supports software security.

Security knowledge and best practices

We can apply software security knowledge at various stages throughout the entire software development life cycle (SDLC). One effective way is through the use of software security best practices.[4] Rules, for example, are extremely useful for static analysis and code inspection activities.

Software development processes

Software security unified knowledge architecture

The basic schema introduced in the main text describes a way to organize and interrelate software security knowledge. Figure A shows the seven distinct knowledge catalogs (the boxes) that we divide into three knowledge categories.

The category *prescriptive knowledge* includes three knowledge catalogs: principles, guidelines, and rules. These sets span a continuum of abstraction from high-level architectural principles at the philosophical level (for example, the principle of least privilege[1]) to very specific and tactical code-level rules (for example, "avoid the use of the library function `gets()` in C"). Guidelines fall somewhere in the middle of this continuum (for example, "make all Java objects and classes `final()`, unless there's a good reason not to"[2]). As a whole, the prescriptive knowledge category offers advice for what to do and what to avoid when building secure software.

The *diagnostic knowledge* category includes three knowledge catalogs: attack patterns, exploits, and vulnerabilities. Rather than prescriptive statements of practice, diagnostic knowledge helps practitioners (including those working in operations) recognize and deal with common problems that lead to security attacks. Vulnerability knowledge includes descriptions of software vulnerabilities experienced and reported in real systems (often with a bias toward operations). Exploits describe how instances of vulnerabilities are leveraged into particular security compromise for particular systems. Finally, attack patterns describe common sets of exploits in a more abstract form that's applicable across multiple systems. Such diagnostic knowledge is particularly useful in the hands of a security analyst, although its value as a resource to be applied during development is considerable (consider, for example, the utility of attack patterns to abuse case development.)

The category *historical knowledge* includes the knowledge catalog historical risks (and, in some cases, vulnerabilities such as the collection in the CVE list; www.cve.mitre.org/). Rather than derivations or abstractions, this catalog represents detailed descriptions of specific issues uncovered in real-world software development efforts; it must also include a statement of impact on the business or mission proposition. As a resource, this knowledge offers tremendous value in helping identify similar issues in new software efforts without starting from scratch. It also provides a continuing source for identifying new instances of other knowledge catalogs described here, including principles, guidelines, rules, vulnerabilities, and attack patterns.

References

1. J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.
2. G. McGraw and E. Felten, *Securing Java: Getting Down to Business with Mobile Code*, John Wiley & Sons, 1999.

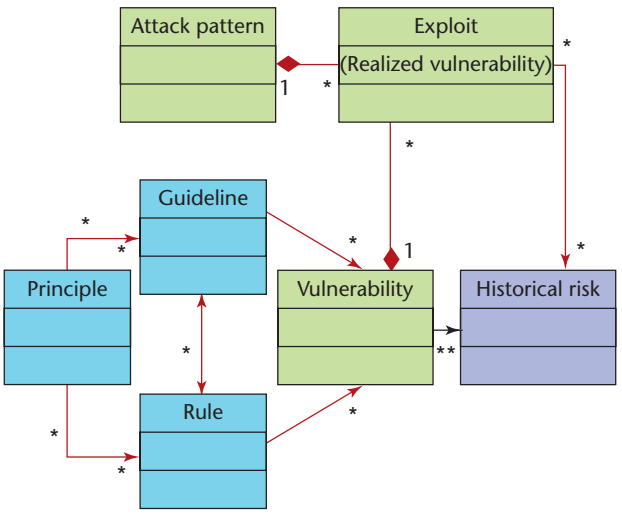

Figure A. Software security knowledge objects and a basic interrelating architecture.

as diverse as the waterfall model, IBM's Rational Unified Process, extreme programming, Agile, spiral development, Capability Maturity Model integration, and any number of other processes involve the creation of a common set of *software artifacts* (the most common artifact being code). Figure 1 shows an enhanced version of the SDLC best practices we've used as the backbone of this series. In the figure, we identify the activities and artifacts most clearly impacted by the knowledge catalogs described here.

Let's examine each knowledge catalog and look at sample schemas for tracking instances and some corresponding SDLC artifacts affected by knowledge.

Principles

A principle is a statement of general security wisdom derived from experience. Although principles exist at the philosophical level, they stem from practitioners' real-world experience in building secure systems. Principles are useful for both diagnosing architectural flaws in software and practicing good security engineering. A sample high-level schema for a principle includes title, definition (with a description, examples, and references), related guidelines, and related rules. Relevant SDLC artifacts include security requirements and the software architecture.

Guidelines

A guideline is a recommendation

Knowledge catalog examples

To make the notion of knowledge catalogs concrete, let's examine two example entries from two different knowledge catalogs.

Principle catalog

An entry in the principle catalog includes the principle and a concrete example. The entry also includes references for those accessing the entry.

Item:

Principle of Least Privilege

Description:

Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need to know" is an example of this principle.

Concrete example:

A good software specific example is a mail server that accepts mail from the Internet, and copies the messages into a spool directory; a local server will complete delivery. It needs rights to access the appropriate network port, to create files in the spool directory, and to alter those files (so it can copy the message into the file, rewrite the delivery address if needed, and add the appropriate "Received" lines). It should surrender the right to access the file as soon as it has completed writing the file into the spool directory, because it does not need to access that file again. The server should not be able to access any user's files or any files other than its own configuration files.

References:

M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2002.

J.H. Saltzer and M.D. Schroeder "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 9, no. 63, 1975, pp. 1278–1308.

J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.

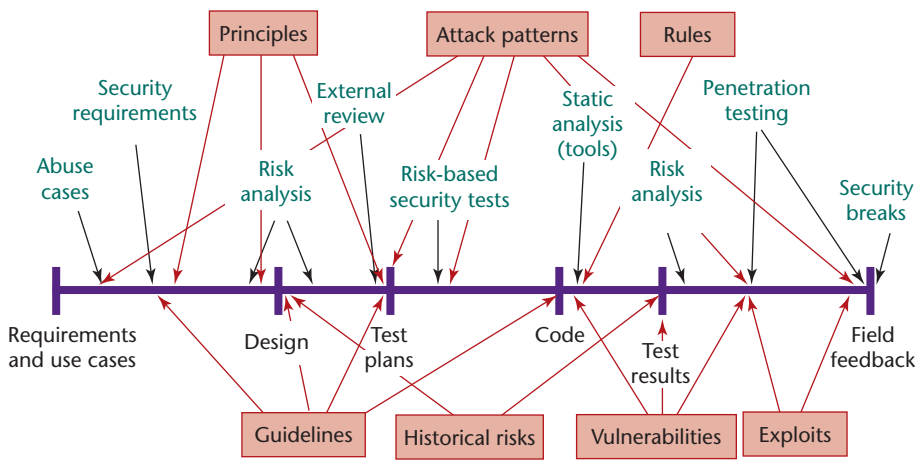

Figure 1. The software development life cycle. Mapping software security knowledge catalogs to various software artifacts and software security best practices shows us how various knowledge catalogs can be applied throughout the SDLC.

for things to do or avoid during software development, described at the semantic level. Guidelines exist for a specific technical context (for example, J2EE, .NET, Linux kernel modules, and so on), but they are best enforced and evaluated through human analysis. Guidelines can help uncover both architectural flaws and implementation bugs. A sample high-level schema for a guideline includes a context description (platform, operating system, language, and so on), title, type, objective, development scenario, description, related API, reference, related principles, related rules, security requirements, and software design. The SDLC artifact most closely aligned is code.

Rules

A rule is a recommendation for things to do or to avoid during software development, described at the syntactic level. Rules can be verified through lexical scanning or constructive software parsing (source or binary); they exist for

Rule Catalog

The rule catalog shows a rule for the particular system call.

Item:

Use of `creat()`

Context:

C/C++

Attack Category:

TOCTOU

Description:

The `creat(char *pathname,mode_t theMode)` function either creates a new file or prepares to rewrite using pathname as the filename. The call `creat(theName,theMode)` is equivalent to `open(theName,O_WRONLY | O_CREAT | O_TRUNC, theMode)` If the file exists, the length is truncated to zero and the mode and owner are unchanged. This function is a problem, because it is possible to unintentionally delete a file or enter a potentially unstable race condition. `creat()` is vulnerable to TOCTOU attacks.

Using automated scanning tools, the existence of a call to this function should be flagged regardless if a "check" function precedes it.

Method of Attack:

The `creat()` call is a "use" category call that when preceded by a "check" category call can indicate a TOCTOU vulnerability.

Solution:

Consider using a safer set of steps for opening and creating files as outlined in *Building Secure Software*. If this call must be used, create a directory only accessible by the UID of the running program and only manipulate files in that directory.

Signature:

Presence of the `creat()` function.

Code example:

In the following case, the contents of the file passed into the `creat` function are destroyed.

```
char filename[] = "rightFile.txt";
    strcpy(filename,"wrongfile.txt");
    creat(filename,theMode);
```

If the results of the function call are used before completion, then the results can also be unstable.

References:

J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001, pp. 220–222.

Unix man page for `creat()`

Microsoft Developer Network Library (MSDN)

specific programming languages and can help uncover implementation bugs. A rule's high-level schema includes a context description (platform, operating system, language, and so on), ID, title, attack category, vulnerability category, location, description, method of attack, solution, signature, example, reference, related principles, and related guidelines. The SDLC artifact is code.

Attack patterns

An attack pattern is developed by reasoning over large sets of software exploits. Attack patterns help identify and qualify the risk that a given exploit will occur in a software system. They're also useful for designing misuse and abuse cases as well as specific security tests. An attack pattern's high-level schema includes a context description (platform, operating system, language, and so on), title, attack category, description, example, reference, related guidelines, and related rules. SDLC artifacts include abuse cases, software design documents, security test plans (and tests), and penetration tests.

Historical risks

A historical risk is a risk identified in the course of an actual software development effort. At its core, a risk is a pairing of a condition and an event, with a quantification of the likelihood that it will occur and the impact it could have. Historical risks are good resources for early identification of potential issues in a software development effort; they offer potential clues to effective mitigations and ideas for improving the consistency and quality of risk management in the software development process. A high-level schema for a historical risk includes title, type (business or technical), subcategories (via taxonometric sorting), author, owner, project, risk status, likelihood, impact, severity, risk context, risk description, realization indicators, impact description, estimated impact date, potential cost, contingency plans and workarounds, related business goals, related risks, related mitigations, and diagnostic methods. SDLC artifacts include software architecture, software design, test plans, and fielded software.

Vulnerabilities

A vulnerability is the result of a software defect that an attacker can use to gain illegal access to—or negatively affect the security of—a computer system. A vulnerability's high-level schema includes a context description (platform, operating system, language, and so on), title, description, severity, vulnera-

bility type, loss type, and reference. SDLC artifacts include code, software architecture, software design, penetration tests, and the fielded system.

Exploits

An exploit is a particular instance of an attack on a computer system that leverages a specific vulnerability or set of vulnerabilities. An exploit's high-level schema includes a context description (platform, operating system, language, and so on), title, description, preconditions, motivation, exposure type, exploit code, blocking solution, and related vulnerabilities. SDLC artifacts include penetration tests and the fielded system.

Applications

The "Knowledge catalog examples" sidebar on p. 76 provides examples of items found in two knowledge catalogs. Principles, given their philosophical level of abstraction, bring significant value to early lifecycle activities, including the definition of security requirements, software architecture risk analysis, and design reviews. Rules, given their tactical, specific syntactic nature, are primarily applicable during code review and are particularly well-suited for inclusion in a static analysis tool. This opportunity for automation means that rules have an implicit requirement for encapsulation in a deterministic definition language so that they can be consumed by automated code-scanning software.

The set of software security knowledge catalogs we've identified here offers an excellent foundation for integrating security knowledge into the full SDLC.

Efforts to identify and define knowledge constructs for software security are in their infancy, but our hope is that a wider population of thought leaders and key software security practitioners will help refine and validate this knowledge architecture, build consensus, and eventually move us toward standardization. Such discussion and collaboration is critical to the success of software security as a unified practice. As we work to gain consensus, we'll continue to collect real-world examples of content to fill out the breadth and depth of the catalogs identified here. We'll also work to identify further opportunities for directly applying these catalogs in the SDLC. □

References

1. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 9, no. 63, 1975, pp. 1278–1308.
2. J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code," *Proc. Ann. Computer Security Applications Conf.* (ASAC 02), IEEE CS Press, 2000, pp. 257–269.
3. G. Hoglund and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.
4. G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80–83.

Sean Barnum *is Director of Knowledge Management at Cigital. His technical interests include software quality and process improvement, software security, risk management, knowledge architecture, and collaborative technologies. Barnum has a BS in computer science from Portland State University. He is a member of the IEEE Computer Society and OASIS, and is involved in numerous knowledge standards-defining efforts. Contact him at sbarnum@cigital.com.*

Gary McGraw *is chief technology officer of Cigital. His real-world experience is grounded in years of consulting with major corporations and software producers. McGraw is the coauthor of* Exploiting Software *(Addison-Wesley, 2004),* Building Secure Software *(Addison-Wesley, 2001),* Java Security *(John Wiley & Sons, 1996), and four other books. He has a BA in philosophy from the University of Virginia and a dual PhD in computer science and cognitive science from Indiana University. Contact him at gem@cigital.com.*