Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors

Katrina Tsipenyuk and Brian Chess
Fortify Software

Gary McGraw
Cigital

Taxonomies can help software developers and security practitioners understand the common coding mistakes that affect security. The goal is to help developers avoid making these mistakes and more readily identify security problems whenever possible. Because developers today are by and large unaware of the security problems they can (unknowingly) introduce into code, a taxonomy of coding errors should provide a real tangible benefit to the software security community.

This approach represents a striking alternative to taxonomies of attack patterns[1] or simple-minded collections of specific vulnerabilities (such as MITRE's CVE, www.cve.mitre.org). Attack-based approaches are based on knowing your enemy and assessing the possibility of similar attack—they represent the black hat side of the software security equation. However, a taxonomy of software security errors is more positive in nature—it's most useful to the white hat side of the software security world. In the end, both approaches are valid and necessary.

Although the taxonomy proposed here is incomplete and imperfect, it provides an important first step. It focuses on collecting common errors and explaining them in a way that makes sense to programmers. This new taxonomy is made up of two distinct kinds of sets, which we're stealing from biology: a *phylum* (a type of coding error, such as illegal pointer value) and a *kingdom* (a collection of phyla that shares a common theme, such as input validation and representation). Both kingdoms and phyla naturally emerge from a soup of coding rules relevant to enterprise software, and it's for this reason that this taxonomy is likely to be incomplete and might lack certain coding errors. In some cases, it's easier and more effective to talk about a category of errors than to talk about any particular attack. Although categories are certainly related to attacks, they aren't the same as attack patterns.

On simplicity: Seven plus or minus two

We've all seen lots of security taxonomies over the years, and they all share one unfortunate property—they're too complex. People are good at keeping track of seven things (plus or minus two),[2] so we used this figure as a hard constraint and tried to keep the number of kingdoms down to seven (plus one). In order of importance to software security, these kingdoms are

- input validation and representation,
- API abuse,
- security features,
- time and state,
- errors,
- code quality,
- encapsulation, and
- environment.

Let's look at each of these a little more closely.

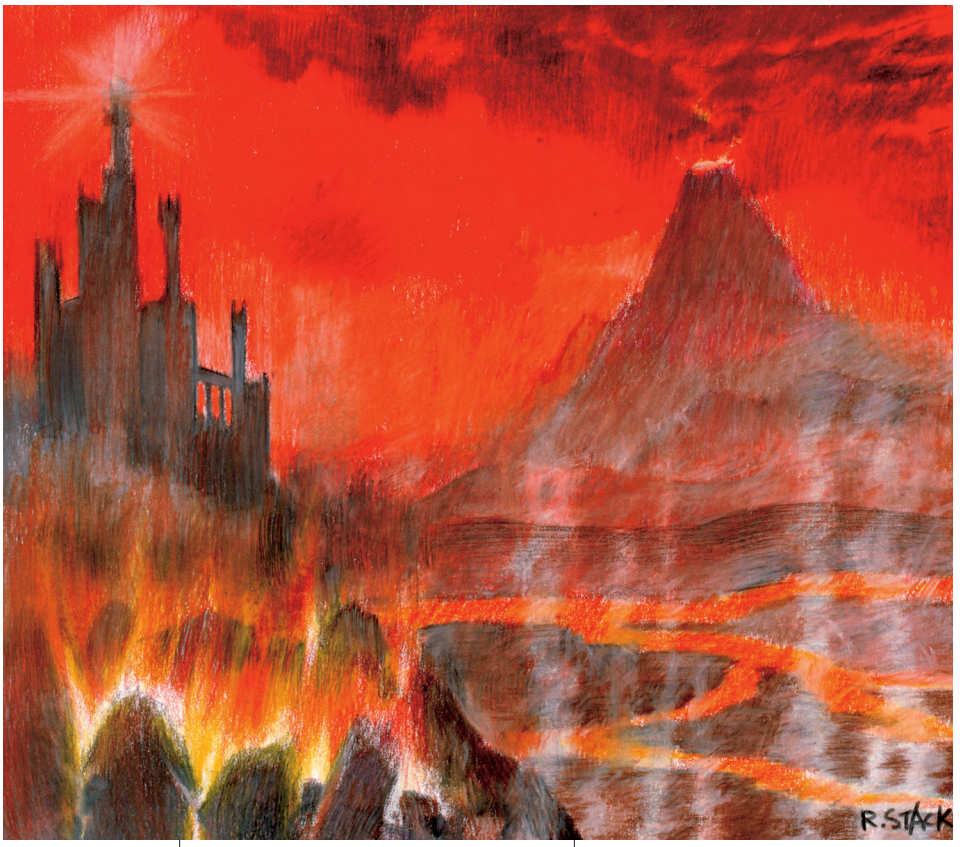
Input validation and representation

Metacharacters, alternate encodings, and numeric representations cause *input validation and representation* problems. Of course, sometimes people just forget to do any input validation at all. If you do choose to do input validation, use a white list, not a black list.

Big problems result from putting too much trust in input: buffer overflows, cross-site scripting attacks, SQL injection, cache poisoning, and basically all the low-hanging fruit the script kiddies love so much.

API abuse

An API is a contract between a caller and a callee: the most common forms of *API abuse* occur when the caller fails to honor its end of the contract. If a program fails to call `chdir()` after calling `chroot()`, for example, it violates the contract that specifies how to securely change the active root directory. Another good example of library abuse is expecting the callee to return trustworthy DNS information to the caller. In this case, the caller abuses the callee API by making certain assumptions about its behavior. Really bad people also violate the caller–callee contract from the other side—for example, if you



subclass `SecureRandom` and return a not-so-random value, you're not following the rules. API abuse categories are very common.

Security features

Software security isn't security software. All the magic crypto fairy dust in the world won't make your code secure, but it's also true that you can drop the ball when it comes to essential security features. Let's say you decide to use the Secure Sockets Layer (SSL) to protect traffic across the network, but you really screw things up. Unfortunately, this happens all the time. When we chunk together *security features*, we're concerned with topics like authentication, access control, confidentiality, cryptography, privilege management, and all that other stuff on the CISSP exam. This stuff is hard to get right.

Time and state

Distributed computation is about time and state—that is, for more than one component to communicate, state must be shared (somehow), which takes time. Playing with *time and state* is the biggest untapped natural attack resource on the planet right now.

Most programmers anthropomorphize (or, more accurately, only solipsistically ponder) their work. They think of themselves as the single omniscient thread of control, plodding along and carrying out the entire program in the same way they would if forced to do the job manually. How quaint. Modern computers switch between tasks very quickly, and in multi-core, multi-CPU, or distributed systems, two events can occur at exactly the same time. Defects related to unexpected interactions between threads, processes, time, and information rush to fill the gap between the programmer's model of how a program executes and what happens in reality. Several interactions happen via shared state—semaphores, variables, the file system, the universe, and, basically, anything that can store information.

Errors

Want to break software? Throw some junk at a program and see what *errors* you cause. Errors are not only a great source of "too much information" from a program, they're also a source of inconsistent thinking that can be gamed. It gets worse: in modern object-oriented systems, the notion of exceptions has reintroduced the banned concept of `goto`. Alas.

Errors and error handlers represent a class of programming contract, so in some sense, errors represent the two sides of a special form of API. However, the security defects related to error handling are so common they deserve a special kingdom all to themselves. As with API abuse, you can blow it in one of two ways: forgetting to handle errors at all (or handing them so roughly that they get all bruised and bloody) or producing errors that give out way too much information (to possible attackers) or are so radioactive that nobody wants to handle them.

Code quality

Security is a subset of reliability, just as all future TV shows are a subset of monkeys banging out scripts on zillions of keyboards. If you can completely specify your system and all of its positive and negative security possibilities, then security is a subset of reliability. Poor *code quality* leads to unpredictable behavior, and from a user's perspective, this often manifests itself as poor usability. For an attacker, bad quality provides an opportunity to stress the system in unexpected ways.

Encapsulation

Encapsulation is about drawing strong boundaries around things and setting up barriers between them. In a Web browser, this might mean ensuring that mobile code can't whack your hard drive arbitrarily. On a Web services server, it might mean differentiating between valid authenticated data and mystery data found stuck like yesterday's gum to the bot-

tom of a desk. Some of the most important boundaries today come between classes with various methods. Trust models require careful and meticulous attention to boundaries—in other words, keep your hands off my stuff!

Environment

Hey, it turns out that software runs on machines with certain bindings and connections to the bad, mean universe. Getting outside the software is important. This kingdom is the kingdom of outside→in, and it includes all the stuff outside your code that's still critical to the security of the software you create.

The phyla

To better understand the relationship between kingdoms and phyla, consider a recently discovered vulnerability in Adobe Reader 5.0.x for Unix. The vulnerability is found in a function called `UnixAppOpenFilePerform()`, which copies user-supplied data into a fixed-size stack buffer via a call to `sprintf()`. If the size of the user-supplied data is greater than the size of the buffer it's copied into, important information (including the stack pointer) is overwritten. Thus by supplying a malicious PDF file, an attacker can execute arbitrary commands on a target system. This attack is possible because of a simple coding error—the absence of a check that ensures that the user-supplied data is no larger than the size of the destination buffer. Developers associate this check with a failure to code defensively around the call to `sprintf()`. We classify it according to the attack it enables—buffer overflow. In our taxonomy, we would choose *input validation and representation* as the kingdom to which the buffer overflow phylum belongs because the lack of proper input validation is the attack's root cause. (Due to space constraints, we can't list all the phyla here; visit http://vulncat.fortifysoftware.com for full descriptions listed under their respective kingdoms.)

Static source-code analysis tools can help detect phyla-represented coding errors. Source-code analysis also gives developers the opportunity to get quick feedback about the code they're writing as they're writing it. Our new taxonomy includes coding errors that occur in a variety of programming languages, the most important of which are C and C++, Java, and the .NET family (including C# and ASP). Some of the phyla are language-specific because the types of errors they represent are applicable only to specific languages, and some are framework-specific (for the same reason). The phylum list as it exists is certainly incomplete, but it's adaptable to changes in the trends and discoveries of new defects that are bound to happen over time, unlike various existing classification schemes. Classifying which errors are most important to real-world enterprise developers is the most important goal of this taxonomy—most of the information in it is derived from the literature, various colleagues, and hundreds of customers.

Lists, piles, and collections

The idea of collecting and organizing information about computer security vulnerabilities has a long history, and several practitioners have even developed "top 10" lists and other related collections based on experience in the field. The taxonomy introduced here negotiates a middle ground between rigorous academic studies and ad hoc collections based on experience.

Two of the most popular and useful lists are the "19 sins" and the "Open Web Application Security Project (OWASP) top 10." The first list is carefully described in the new book *19 Deadly Sins of Software Security*,[3] and the OWASP Top 10 Most Critical Web Application Security Vulnerabilities is available at www.owasp.org/documentation/topten.html. The main limitation of both lists is that they mix specific types of errors and vulnerability classes and talk about them all at the same level of abstraction. The 19 sins, for example, include both "buffer overflows" and "failing to protect network traffic" categories at the same level, even though the first is a very specific coding error, whereas the second is a class comprising various kinds of errors. Similarly, OWASP's top 10 includes "cross-site scripting (XSS) flaws" and "insecure configuration management" at the same level. This unnecessary level confusion is a serious problem that can lead to confusion among practitioners. Although our classification scheme is quite different from the lists described earlier, we can easily map the categories that comprise these lists to our kingdoms (see Table 1).

The seven pernicious kingdoms offer a simple, effective organizing tool for software security coding errors, and with more than 60 clearly defined phyla,[4] this taxonomy is both powerful and useful. But taxonomy work is an ongoing quest, and further refinement and evolution is necessary. Please send any feedback regarding this taxonomy to brian@fortifysoftware.com. ▢

References

1. G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.
2. G. Miller, "The Magic Number Seven, Plus or Minus Two," *The Psychological Rev.*, vol. 63, 1956, pp. 81–97; www.well.com/user/smalin/miller.html.
3. M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security*, McGraw-Hill, 2005.
4. K. Tsipenyuk, B. Chess, and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," to be pub-

Table 1. Mapping 19 sins and the OWASP top 10 to the proposed kingdoms.

KINGDOMS	19 SINS	OPEN WEB APPLICATION SECURITY PROJECT (OWASP) TOP 10
Input validation and representation	Buffer overflows, command injection, cross-site scripting, format string problems, integer range errors, SQL injection	Buffer overflows, cross-site scripting flaws, injection flaws, unvalidated input
API abuse	Trusting network address information	
Security features	Failing to protect network traffic, failing to store and protect data, failing to use cryptographically strong random numbers, improper file access, improper use of SQL, use of weak password-based systems, unauthenticated key exchange	Broken access control, insecure storage
Time and state	Signal race conditions, use of "magic" URLs and hidden forms	Broken authentication and session management
Errors	Failure to handle errors	Improper error handling
Code quality	Poor usability	Denial of service
Encapsulation	Information leakage	
Environment		Insecure configuration management

lished in *Proc. NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM)*, US Nat'l Inst. Standards and Technology, 2005.

Katrina Tsipenyuk *is the founding member of the Security Research Group at Fortify Software. She has a BS and an MS in computer science with an emphasis on security from the University of California, San Diego. Her thesis work includes research on mobile agent security. Contact her at katrina@fortifysoftware.com.*

Brian Chess *is chief scientist at Fortify Software. His work focuses on practical methods for creating secure systems. Chess has a PhD in computer engineering from the University of California, Santa Cruz, where he applied his background in integrated circuit test and verification to the problem of identifying security-relevant defects in software. Contact him at brian@fortifysoftware.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.*