Seven Pernicious Kingdoms:
A Taxonomy of Software Security Errors

ABSTRACT

Coding errors are a critical problem in software security. This Research Report shows the common types of coding errors organized into a simple taxonomy to help developers and security practitioners recognize the categories of problems that lead to vulnerabilities and help them identify existing errors as they build software.

The information contained in our taxonomy is most effectively enforced via a tool. In fact, all of the errors included in our taxonomy are amenable to automatic identification using static source code analysis techniques.

Throughout these pages, we demonstrate why our taxonomy is not only simpler, but also more comprehensive than other modern taxonomy proposals and vulnerability lists. We provide an in-depth explanation and one or more code-level examples for each of the errors on a companion web site: www.fortifysoftware.com/vulncat

INTRODUCTION

Software developers play a crucial role in building secure computer systems. In defining this taxonomy of coding errors, our primary goal is to organize sets of security rules that can be used to help software developers understand the kinds of errors that have an impact on security. We believe that one of the most effective ways to deliver this information to developers is through the use of tools. Our hope is that, by better understanding how systems fail, developers will better analyze the systems they create, more readily identify and address security problems when they see them, and generally avoid repeating the same mistakes in the future.

When put to work in a tool, a set of security rules organized according to this taxonomy is a powerful teaching mechanism. Because developers today are by and large unaware of the myriad ways they can introduce security problems into their work, publication of a taxonomy like this should provide tangible benefits to the software security community.

Defining a better classification scheme can also lead to better tools: a better understanding of the problems will help researchers and practitioners create better methods for ferreting them out.

We propose a simple, intuitive taxonomy, which we believe is the best approach for our stated purpose of organizing sets of software security rules that will teach software developers about security. Our approach is an alternative to a highly specific list of attack types and vulnerabilities offered by CVE (Common Vulnerabilities and Exposures) [7], which lacks in the way of categorization and is operational in nature. Our classification scheme is amenable to automatic identification and can be used with static analysis tools for detecting real-world security vulnerabilities in software. Our approach is also an alternative to a number of broad classification schemes that focus exclusively on operating-system-related vulnerabilities [1,2,3,12,19]. We discuss these taxonomies in Section 2.

Section 3 motivates our work and discusses the relationship between coding errors and corresponding attacks. It also defines terminology used throughout the rest of this paper. Section 4 describes the scheme we propose. We refer to a type of coding error as a phylum and a related set of phyla as a kingdom. A complete description of each phylum is available on this paper's companion web site [8]. Section 5 draws parallels between two other vulnerability lists [11,17]. Section 6 concludes.

RELATED WORK

All scientific disciplines benefit from a method for organizing their topic of study, and software security is no different. The value of a classification scheme is indisputable: a taxonomy is necessary in order to create a common vocabulary and an understanding of the ways computer security fails. The problem of defining a taxonomy has been of great interest since the mid-1970s. Several classification schemes have been proposed since then [4].

One of the first studies of computer security and privacy was the RISOS (Research Into Secure Operating Systems) project [1]. RISOS proposed and described seven categories of operating system security defects. The purpose of the project was to understand

security problems in existing operating systems, including MULTICS, TENEX, TOPS-10, GECOS, OS/MVT, SDS-940, and EXEC-8, and to determine ways to enhance the security of these systems. The categories proposed in the RISOS project include:

- Incomplete Parameter Validation
- Inconsistent Parameter Validation
- Implicit Sharing of Privileges / Confidential Data
- Asynchronous Validation / Inadequate Serialization
- Inadequate Identification / Authentication / Authorization
- Violable Prohibition / Limit
- Exploitable Logic Error
- The study shows that there are a small number of fundamental defects that recur in different contexts.

The objective of the Protection Analysis (PA) project [3] was to enable anybody (with or without any knowledge about computer security) to discover security errors in the system by using a pattern-directed approach. The idea was to use formalized patterns to search for corresponding errors. The PA project was the first project to explore automation of security defects detection. However, the procedure for reducing defects to abstract patterns was not comprehensive, and the technique could not be properly automated. The database of vulnerabilities collected in the study was never published.

Landwehr, Bull, McDermott, and Choi [12] classify each vulnerability from three perspectives: genesis (how the problem entered the system), time (at which point in the production cycle the problem entered the system), and location (where in the system the problem is manifest). Defects by genesis were broken down into intentional and inadvertent, where the intentional class was further broken down into malicious and non-malicious. Defects by time of introduction were broken down into development, maintenance, and operation, where the development class was further broken down into design, source code, and object code. Defects by location were broken down into software and hardware, where the software class was further broken down into operating system, support, and application. A very similar scheme was proposed by Weber, Karger, and Paradkar [21]. However, their scheme classifies vulnerabilities only according to genesis.

The advantage of this type of hierarchical classification is the convenience of identifying strategies to remedy security problems. For example, if most security issues are introduced inadvertently, increasing resources devoted to code reviews becomes an effective way of increasing security of the system. The biggest disadvantage of this scheme is inability to classify some existing vulnerabilities. For example, if it is not known how the vulnerability entered the system, it cannot be classified by genesis at all.

Another scheme relevant to our discussion is ODC (Orthogonal Defect Classification) [19] proposed and widely used at IBM. ODC categorizes defects according to error type (a low-level programming mistake) and trigger event (environment characteristics that caused a defect). Additionally, each defect is characterized by severity and symptom. However, ODC focuses on operating system quality issues rather than security issues.

breadth of the categories making classification ambiguous. In some cases, one issue can be classified in more than one category. The category names, while useful to some groups of researchers, are too generic to be quickly intuitive to a developer in the context

of day-to-day work. Additionally, these schemes focus mostly on operating system security problems and do not classify the ones associated with user-level software security. Furthermore, these taxonomies mix implementation-level and design-level defects and are not consistent about defining the categories with respect to the cause or effect of the problem.

The work done by Landwehr, Bull, McDermott, and Choi was later extended by Viega [20]. In addition to classifying vulnerabilities according to genesis, time, and location, he also classifies them by consequence (effects of the compromise resulting from the error) and other miscellaneous information, including platform, required resources, severity, likelihood of exploit, avoidance and mitigation techniques, and related problems. Each category is discussed in detail and provides specific examples, including, in some cases code excerpts. This “root-cause” database, as Viega calls it, strives to provide a lexicon for the underlying problems that form the basis for the many known security defects. As a result, not all of the issues in this taxonomy are security problems. Furthermore, the “root-cause” database allows the same problem to be classified differently depending upon the interests of the person doing the classification.

A good list of attack classes is provided by Cheswick, Bellovin, and Rubin [5]. The list includes:

- Stealing Passwords
- Social Engineering
- Bugs and Back Doors
- Authentication Failures
- Protocol Failures
- Information Leakage
- Exponential Attacks—Viruses and Worms
- Denial-of-Service Attacks
- Botnets
- Active Attacks

A thorough description with examples is provided for each class. These attack classes are distinguishes the list from other classification schemes. The classes are simple and intuitive. However, this list defines attack classes rather than categories of common coding errors that cause these attacks. A similar, but a more thorough list of attack patterns is given by Hoglund and McGraw [10]. 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. A taxonomy of coding errors is more positive in nature. This kind of thing is most useful to the white-hat side of the software security world. In the end, both kinds of approaches are valid and necessary.

The classification scheme proposed by Aslam [2] is the only precise scheme discussed here. In this scheme, each vulnerability belongs to exactly one category. The decision procedure for classifying an error consists of a set of questions for each vulnerability category. Aslam’s system is well-defined and offers a simple way for identifying defects by similarity. Another contribution of Aslam’s taxonomy is that it draws on software fault studies to develop its categories. However, it focuses exclusively on implementation issues in the UNIX operating system and offers categories that are still too broad for our purpose.

The most recent classification scheme we are aware of is PLOVER (Preliminary List of Vulnerability Examples for Researchers) [6], which is a starting point for the creation of a formal enumeration of WIFFs (Weaknesses, Idiosyncrasies, Faults, Flaws) called CWE (Common WIFF Enumeration) [13]. Twenty-eight main categories that comprise almost three hundred WIFFs put Christey's and Martin's classification scheme at the other end of the ambiguity spectrum—the vulnerability categories are much more specific than in any of the taxonomies discussed above. Their bottom-up approach is complimentary to our efforts. PLOVER and CWE are extensions of Christey's earlier work in assigning CVE (Common Vulnerabilities and Exposures) [7] names to publicly known vulnerabilities. An attempt to draw parallels between theoretical attacks and vulnerabilities known in practice is an important contribution and a big step forward from most of the earlier schemes.

MOTIVATION

Most existing classification schemes, as is evident, begin with a theoretical and comprehensive approach to classifying security defects. Most research to date has been focusing on making the scheme deterministic and precise, striving for a one-to-one mapping between a vulnerability and the category the vulnerability belongs to. Another facet of the same goal has been to make classification consistent for different levels of abstraction: the same vulnerability should be classified into the same category regardless of whether it is considered from a design or implementation perspective.

Most of the proposed schemes focus on classifying operating-systems-related security defects rather than the errors in software security. Furthermore, categories that comprise many of the existing taxonomies were meant to be both broad and rigorously defined instead of intuitive and specific. Overall, most of the schemes cannot easily be applied to organizing security rules used by a software developer who wants to learn how to build secure software.

To further our goal of educating software developers about common errors, we forgo the breadth and complexity essential to theoretical completeness in favor of practical language centered on programming concepts that are approachable and meaningful to developers.

Before we proceed, we need to define the terminology borrowed from Biology which we use to talk about our classification scheme throughout the rest of the paper.

Definition 1. By phylum we mean a specific type of coding error. For example, Illegal Pointer Value is a phylum.

Definition 2. A kingdom is a collection of phyla that share a common theme. For example, Input Validation and Representation is a kingdom.

In defining our taxonomy, we value concrete and specific problems that are a real concern to software security over abstract and theoretical ones that either have not been seen in practice or are a result of high-level unsafe specification decisions. We did not make it a goal to create a theoretically complete classification scheme. Instead, we offer a scheme that is open-ended and amenable to future expansion. We expect the list of important phyla to change over time. We expect the important kingdoms to change too, though at a lesser rate. Any evolution will be influenced by trends in languages, frameworks, and libraries; discovery of new types of attacks; new problems and verticals toward which

software is being applied; the regulatory landscape, and social norms.

We value simplicity over parallelism in order to create kingdoms that are intuitive to software developers who are not security experts. As opposed to most of the classification schemes discussed in Section 2, our taxonomy focuses on code-level security problems that occur in a range of software applications rather than errors that are most applicable to specific kinds of software, such as operating systems. For example, Buffer Overflow and Command Injection [8] are a part of our taxonomy, while analysis of keystrokes and timing attacks on SSH [18], as well as other kinds of covert-channel-type attacks, are not included. There is no reason to believe that the kingdoms we have chosen would not work for operating systems or other types of specialized software, however there are many more developers working on business applications and desktop programs than on operating systems.

To better understand the relationship between the phyla our taxonomy offers, consider a recently found vulnerability in Adobe Reader 5.0.x for Unix [9]. The vulnerability is present in a function UnixAppOpenFilePerform() that copies user-supplied data into a fixed-size stack buffer using a call to sprintf(). If the size of the user-supplied data is greater than the size of the buffer it is being copied into, important information, including the stack pointer, is overwritten. By supplying a malicious PDF document, an attacker can execute arbitrary commands on the target system. The attack is possible because of a simple coding error—the absence of a check that makes sure that the size of the user-supplied data is no greater than the size of the destination buffer. In our experience, developers will associate this check with a failure to code defensively around the call to sprintf(). We classify this coding error according to the attack it enables—Buffer Overflow. We choose Input Validation and Representation as the name of the kingdom Buffer Overflow phylum belongs to because the lack of proper input validation is the reason the attack is possible.

The coding errors represented by our phyla can all be detected by static source code analysis tools. Source code analysis offers developers an opportunity to get quick feedback about the code that they write. We see great potential for educating developers about coding errors by having them use a source code analysis tool.

THE TAXONOMY

We now provide a summary of our taxonomy, which will also appear in McGraw's new book [14]. We split the phyla into "seven-plus-one" high-level kingdoms that should make sense to a majority of developers. Seven of these kingdoms are dedicated to errors in source code, and one is related to configuration and environment issues. We present them in order of importance to software security:

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Errors
6. Code Quality
7. Encapsulation
8. *. Environment

Brief descriptions of the kingdoms and phyla are provided below. Complete descriptions

with source code examples are available on the internet at http://vulncat.fortifysoftware.com.

Our taxonomy includes coding errors that occur in a variety of programming languages. The most important among them are C and C++, Java, and the .NET family including C# and ASP. Some of our phyla are language-specific because the types of errors they represent are applicable only to specific languages. One example is the Double Free phylum. It identifies incorrect usage of low-level memory routines. This phylum is specific to C and C++ because neither Java nor the managed portions of the .NET languages expose low-level memory APIs.

In addition to being language-specific, some of our phyla are framework-specific. For example, the Struts phyla apply only to the Struts framework and the J2EE phyla are only applicable in the context of the J2EE applications. Log Forging, on the other hand, is a more general phylum.

Our phylum list is certainly incomplete, but it is adaptable to changes in trends and discoveries of new defects that will happen over time. We focus on finding and classifying security-related defects rather than more general quality or reliability issues. The Code Quality kingdom could potentially contain many more phyla, but we feel that the ones that we currently include are the ones most likely to affect software security. Finally, we concentrate on classifying errors that are most important to real-world enterprise developers—we derive this information from the literature, our colleagues, and our customers.

1. **Input Validation and Representation**
   Input validation and representation problems are caused by metacharacters, alternate encodings and numeric representations. Security problems result from trusting input. The issues include: Buffer Overflows, Cross-Site Scripting attacks, SQL Injection, and many others.

   - **Buffer Overflow.** Writing outside the bounds of allocated memory can corrupt data, crash the program, or cause the execution of an attack payload.
   - **Command Injection**. Executing commands from an untrusted source or in an untrusted environment can cause an application to execute malicious commands on behalf of an attacker.
   - **Cross-Site Scripting.** Sending unvalidated data to a Web browser can result in the browser executing malicious code (usually scripts).
   - **Format String.** Allowing an attacker to control a function's format string may result in a buffer overflow.
   - **HTTP Response Splitting.** Writing unvalidated data into an HTTP header allows an attacker to specify the entirety of the HTTP response rendered by the browser.
   - **Illegal Pointer Value.** This function can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer may have unintended consequences.
   - **Integer Overflow.** Not accounting for integer overflow can result in logic errors or buffer overflows.
   - **Log Forging.** Writing unvalidated user input into log files can allow an attacker to forge log entries or inject malicious content into logs.
   - **Path Manipulation.** Allowing user input to control paths used by the application may enable an attacker to access otherwise protected files.

- **Process Control.** Executing commands or loading libraries from an untrusted source or in an untrusted environment can cause an application to execute malicious commands (and payloads) on behalf of an attacker.
- **Resource Injection.** Allowing user input to control resource identifiers may enable an attacker to access or modify otherwise protected system resources.
- **Setting Manipulation.** Allowing external control of system settings can disrupt service or cause an application to behave in unexpected ways.
- **SQL Injection**. Constructing a dynamic SQL statement with user input may allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.
- **String Termination Error.** Relying on proper string termination may result in a buffer overflow.
- **Struts: Duplicate Validation Forms.** Multiple validation forms with the same name indicate that validation logic is not up-to-date.
- **Struts: Erroneous validate() Method.** The validator form defines a validate() method but fails to call super.validate().
- **Struts: Form Bean Does Not Extend Validation Class.** All Struts forms should extend a Validator class.
- **Struts: Form Field Without Validator.** Every field in a form should be validated in the corresponding validation form.
- **Struts: Plug-in Framework Not In Use.** Use the Struts Validator to prevent vulnerabilities that result from unchecked input.
- **Struts: Unused Validation Form.** An unused validation form indicates that validation logic is not up-to-date.
- **Struts: Unvalidated Action Form.** Every Action Form must have a corresponding validation form.
- **Struts: Validator Turned Off.** This Action Form mapping disables the form's validate() method.
- **Struts: Validator Without Form Field.** Validation fields that do not appear in forms they are associated with indicate that the validation logic is out of date.
- **Unsafe JNI.** Improper use of the Java Native Interface (JNI) can render Java applications vulnerable to security bugs in other languages.
- **Unsafe Reflection.** An attacker may be able to create unexpected control flow paths through the application, potentially bypassing security checks.
- **XML Validation.** Failure to enable validation when parsing XML gives an attacker the opportunity to supply malicious input.

2. **API Abuse**

An API is a contract between a caller and a callee. The most common forms of API abuse are caused by the caller failing to honor its end of this contract. For example, if a program fails to call chdir() after calling chroot(), it violates the contract that specifies how to change the active root directory in a secure fashion. 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 (that the return value can be used for authentication purposes). One can also violate the caller-callee contract from the other side. For example, if a coder subclasses SecureRandom and returns a non-random value, the contract is violated.

- **Dangerous Function.** Functions that cannot be used safely should never be used.
- **Directory Restriction.** Improper use of the chroot() system call may allow

attackers to escape a chroot jail.

- **Heap Inspection.** Do not use realloc() to resize buffers that store sensitive information.
- **J2EE Bad Practices: getConnection().** The J2EE standard forbids the direct management of connections.
- **J2EE Bad Practices: Sockets.** Socket-based communication in web applications is prone to error.
- **Often Misused: Authentication.** Do not rely on the name the getlogin() family of functions returns because it is easy to spoof.
- **Often Misused: Exception Handling.** A dangerous function can throw an exception, potentially causing the program to crash.
- **Often Misused: File System.** Passing an inadequately-sized output buffer to a path manipulation function can result in a buffer overflow.
- **Often Misused: Privilege Management.** Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities.
- **Often Misused:** Strings. Functions that manipulate strings encourage buffer overflows.
- **Unchecked Return Value.** Ignoring a method's return value can cause the program to overlook unexpected states and conditions.

3. **Security Features**

Software security is not security software. Here we're concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management.

- **Insecure Randomness.** Standard pseudo-random number generators cannot withstand cryptographic attacks.
- **Least Privilege Violation.** The elevated privilege level required to perform operations such as chroot() should be dropped immediately after the operation is performed.
- **Missing Access Control.** The program does not perform access control checks in a consistent manner across all potential execution paths.
- **Password Management.** Storing a password in plaintext may result in a system compromise.
- **Password Management: Empty Password in Config File.** Using an empty string as a password is insecure.
- **Password Management: Hard-Coded Password.** Hard coded passwords may compromise system security in a way that cannot be easily remedied.
- **Password Management: Password in Config File.** Storing a password in a configuration file may result in system compromise.
- **Password Management:** Weak Cryptography. Obscuring a password with a trivial encoding does not protect the password.
- **Privacy Violation.** Mishandling private information, such as customer passwords or social security numbers, can compromise user privacy and is often illegal.

4. **Time and State**

Distributed computation is about time and state. That is, in order for more than one component to communicate, state must be shared, and all that takes time.

Most programmers anthropomorphize their work. They think about one thread of control carrying out the entire program in the same way they would if they had to do the job themselves. Modern computers, however, switch between tasks very quickly, and in

multi-core, multi-CPU, or distributed systems, two events may take place at exactly the same time. Defects rush to fill the gap between the programmer's model of how a program executes and what happens in reality. These defects are related to unexpected interactions between threads, processes, time, and information. These interactions happen through shared state: semaphores, variables, the file system, and, basically, anything that can store information.

- **Deadlock.** Inconsistent locking discipline can lead to deadlock.
- **Failure to Begin a New Session upon Authentication.** Using the same session identifier across an authentication boundary allows an attacker to hijack authenticated sessions.
- **File Access Race Condition: TOCTOU.** The window of time between when a file property is checked and when the file is used can be exploited to launch a privilege escalation attack.
- **Insecure Temporary File.** Creating and using insecure temporary files can leave application and system data vulnerable to attack.
- **J2EE Bad Practices: System.exit().** A Web application should not attempt to shut down its container.
- **J2EE Bad Practices: Threads.** Thread management in a Web application is forbidden in some circumstances and is always highly error prone.
- **Signal Handling Race Conditions.** Signal handlers may change shared state relied upon by other signal handlers or application code causing unexpected behavior.

5. Errors

Errors and error handling represent a class of API. Errors related to error handling are so common that they deserve a special kingdom of their own. As with API Abuse, there are two ways to introduce an error-related security vulnerability: the most common one is handling errors poorly (or not at all). The second is producing errors that either give out too much information (to possible attackers) or are difficult to handle.

- **Catch NullPointerException.** Catching NullPointerException should not be used as an alternative to programmatic checks to prevent dereferencing a null pointer.
- **Empty Catch Block.** Ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed.
- **Overly-Broad Catch Block.** Catching overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.
- **Overly-Broad Throws Declaration.** Throwing overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.

6. Code Quality

Poor code quality leads to unpredictable behavior. From a user's perspective that often manifests itself as poor usability. For an attacker it provides an opportunity to stress the system in unexpected ways.

- **Double Free.** Calling free() twice on the same memory address can lead to a buffer overflow.
- **Inconsistent Implementations.** Functions with inconsistent implementations across operating systems and operating system versions cause portability problems.
- **Memory Leak.** Memory is allocated but never freed leading to resource

exhaustion.

- **Null Dereference.** The program can potentially dereference a null pointer, thereby raising a NullPointerException.
- **Obsolete.** The use of deprecated or obsolete functions may indicate neglected code.
- **Undefined Behavior.** The behavior of this function is undefined unless its control parameter is set to a specific value.
- **Uninitialized Variable.** The program can potentially use a variable before it has been initialized.
- **Unreleased Resource.** The program can potentially fail to release a system resource.
- **Use After Free.** Referencing memory after it has been freed can cause a program to crash.

7. Encapsulation

Encapsulation is about drawing strong boundaries. In a web browser that might mean ensuring that your mobile code cannot be abused by other mobile code. On the server it might mean differentiation between validated data and unvalidated data, between one user's data and another's, or between data users are allowed to see and data that they are not.

- **Comparing Classes by Name.** Comparing classes by name can lead a program to treat two classes as the same when they actually differ.
- **Data Leaking Between Users.** Data can "bleed" from one session to another through member variables of singleton objects, such as Servlets, and objects from a shared pool.
- **Leftover Debug Code.** Debug code can create unintended entry points in an application.
- **Mobile Code:** Object Hijack. Attackers can use Cloneable objects to create new instances of an object without calling its constructor.
- **Mobile Code:** Use of Inner Class. Inner classes are translated into classes that are accessible at package scope and may expose code that the programmer intended to keep private to attackers.
- **Mobile Code:** Non-Final Public Field. Non-final public variables can be manipulated by an attacker to inject malicious values.
- **Private Array-Typed Field Returned From a Public Method.** The contents of a private array may be altered unexpectedly through a reference returned from a public method.
- **Public Data Assigned to Private Array-Typed Field.** Assigning public data to a private array is equivalent giving public access to the array.
- **System Information Leak.** Revealing system data or debugging information helps an adversary learn about the system and form an attack plan.
- **Trust Boundary Violation.** Commingling trusted and untrusted data in the same data structure encourages programmers to mistakenly trust unvalidated data.

*. Environment

This section includes everything that is outside of the source code but is still critical to the security of the product that is being created. Because the issues covered by this kingdom are not directly related to source code, we separated it from the rest of the kingdoms.

- **ASP .NET Misconfiguration: Creating Debug Binary.** Debugging messages

help attackers learn about the system and plan a form of attack.

- **ASP .NET Misconfiguration: Missing Custom Error Handling.** An ASP .NET application must enable custom error pages in order to prevent attackers from mining information from the framework's built-in responses.
- **ASP .NET Misconfiguration: Password in Configuration File.** Do not hardwire passwords into your software.
- **Insecure Compiler Optimization.** Improperly scrubbing sensitive data from memory can compromise security.
- **J2EE Misconfiguration: Insecure Transport.** The application configuration should ensure that SSL is used for all access-controlled pages.
- **J2EE Misconfiguration: Insufficient Session-ID Length.** Session identifiers should be at least 128 bits long to prevent brute-force session guessing.
- **J2EE Misconfiguration: Missing Error Handling.** A Web application must define a default error page for 404 errors, 500 errors and to catch java.lang.Throwable exceptions to prevent attackers from mining information from the application container's built-in error response.
- **J2EE Misconfiguration: Unsafe Bean Declaration.** Entity beans should not be declared remote.
- **J2EE Misconfiguration: Weak Access Permissions.** Permission to invoke EJB methods should not be granted to the ANYONE role.

SEVEN PLUS OR MINUS TWO

There are several other software security problem lists that have been recently developed and made available. The first is called the 19 Deadly Sins of Software Security [11]. The second is the OWASP Top Ten Most Critical Web Application Security Vulnerabilities available on the web [17]. Both share one unfortunate property—an overabundance of complexity. People are good at keeping track of seven things (plus or minus two) [16]. We used this as a hard constraint and attempted to keep the number of kingdoms in our taxonomy down to seven (plus one).

By discussing these lists with respect to the scheme we propose, we illustrate and emphasize the superiority of our taxonomy. The main limitation of both lists is that they mix specific types of errors and vulnerability classes, and talk about them at the same level of abstraction. The nineteen deadly sins include the Buffer Overflows and Failing to Protect Network Traffic categories at the same level, even though the first is a very specific coding error, while the second could be a class comprised of various kinds of errors. OWASP's Top Ten includes Cross Site Scripting (XSS) Flaws and Insecure Configuration Management at the same level as well.

Our classification scheme consists of two hierarchical levels: kingdoms and phyla. The kingdoms represent the classes of errors, while the phyla that comprise the kingdoms represent specific errors. We would like to point out that even though the structure of our classification scheme is different from the structure of the lists described above, the categories that comprise these lists can be easily mapped to our kingdoms. Here is the mapping for the nineteen sins:

1. Input Validation and Representation
Buffer Overflows
Command Injection
Cross-Site Scripting
Format String Problems
Integer Range Errors
SQL Injection
2. API Abuse
Trusting Network Address Information
3. 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 SSL
Use of Weak Password-Based Systems
Unauthenticated Key Exchange
4. Time and State
Signal Race Conditions
Use of “Magic” URLs and Hidden Forms
5. Errors
Failure to Handle Errors
6. Code Quality
Poor Usability
7. Encapsulation
Information Leakage
***. Environment**

Here is the mapping for the OWASP Top Ten:

1. Input Validation and Representation
Buffer Overflows
Cross-Site Scripting (XSS) Flaws
Injection Flaws
Unvalidated Input
2. API Abuse
3. Security Features
Broken Access Control
Insecure Storage
4. Time and State
Broken Authentication and Session Management
5. Errors
Improper Error Handling
6. Code Quality
Denial of Service
7. Encapsulation
***. Environment**
Insecure Configuration Management

CONCLUSION

We present a simple, intuitive taxonomy of common coding errors that affect security. We discuss the relationship between vulnerability phyla we define and corresponding attacks, and provide descriptions of each kingdom in the proposed taxonomy.

We point out the important differences between the scheme we propose and those discussed in related work. The classification scheme we present is designed to organize security rules, and thus be of help to software developers who are concerned with writing secure code and being able to automate detection of security defects. These goals make our scheme simple, intuitive to a developer, practical rather than theoretical and comprehensive, amenable to automatic identification of errors with static analysis tools, as well as adaptable with respect to changes in trends that can happen over time.

ABOUT FORTIFY SOFTWARE

Fortify Software products protect companies from today's greatest security risk: the software applications that run their businesses.

Combining deep application security expertise with extensive software development experience, Fortify Software has defined the market with award-winning products that span the software development cycle. Today, Fortify Software fortifies the software for the most demanding customer deployments, including the world's largest, most varied code bases.

Fortify Software is the software security vendor of choice of government agencies and Fortune 500 companies in a wide variety of industries such as energy, financial services, healthcare, e-commerce, media, telecommunications, publishing, insurance, systems integration, and information management.

For More Information
For information about Fortify Software products, plus a free Security Assessment to help you determine your exposure to security risk, visit our Web site or contact us today:

Web site: www.fortifysoftware.com
Telephone: 650.213.5600
Email: contact@fortifysoftware.com

REFERENCES

[1] R.P. Abbott, J. S. Chin, J.E. Donnelley, W.L. Konigsford, S. Tokubo, and D.A. Webb. Security Analysis and Enhancements of Computer Operating Systems. NBSIR 76-1041, National Bureau of Standards, ICST, Washington, D.C., 1976.
[2] T. Aslam. *A Taxonomy of Security Faults in the Unix Operating System*. Master's Thesis, Purdue University, 1995.
[3] R. Bisbey and D. Hollingworth. Protection Analysis Project Final Report. ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, 1978.
[4] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, December 2002.

[5] W. Cheswick, S. Bellovin, and A. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*, Second Edition. Addison-Wesley, 2003.
[6] S. Christey. PLOVER—Preliminary List of Vulnerability Examples for Researchers. Draft, August 2005. http://cve.mitre.org/docs/plover/.
[7] CVE – Common Vulnerabilities and Exposures. http://www.cve.mitre.org/.
[8] Fortify Descriptions. http://vulncat.fortifysoftware.com.
[9] Fortify Extra. Adobe Reader for Unix Remote Buffer Overflow. http://extra.fortifysoftware.com/archives/2005/07/adobe_reader_fo_1.html.
[10] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, February 2004.
[11] M. Howard, D. LeBlanc, and J. Viega. *19 Deadly Sins of Software Security*. McGraw-Hill Osborne Media, July 2005.
[12] C. E. Landwehr, A. R. Bull, J. P. McDermott, W. S. Choi. A Taxonomy of Computer Program Security Flaws, with Examples. *ACM Computing Surveys*, Vol. 26, No. 3, September 1994, pp. 211-254.