White paper

# The Evolution of a Taxonomy:

## Ten Years of Software Security

HP Security Research

# Table of Contents

# The Evolution of a Taxonomy: Ten Years of Software Security

Prajakta Jagdale, Yekaterina O'Neil, Joe Sechman, Jacob West
HP Security Research
{jagdale, katrina, sechman, jbw}@hp.com

**Abstract**—The effectiveness of a software security initiative depends heavily on the framework used to identify, characterize, and communicate security information across different phases of the development lifecycle. We describe the *Seven Pernicious Kingdoms* taxonomy, which models the details and impact of mistakes that commonly lead to security vulnerabilities. When used to organize secure coding rules, the taxonomy provides consistent terminology across vulnerability assessment techniques and roles ranging from security practitioners to architects and developers, as well as up the management chain. Initiatives, particularly those that employ automated testing tools, can use the taxonomy to communicate security findings to development groups in language meaningful to them as opposed to terminology specific to security or a particular assessment technique. The taxonomy is organized hierarchically to reinforce the commonality between related issues affecting different areas of program functionality.

## 1. Introduction

The success of a software security initiative depends on its approach to managing the vulnerabilities it tries to find, remediate, and prevent. To ensure that developers can identify and address risk in their systems, the terminology used in training and education must be parallel to that used by automated tools and other vulnerability assessment efforts. Furthermore, the framework used to describe vulnerabilities must be flexible enough to grow as the industry develops and discovers new vulnerability types.

Understanding threats and their impact on software systems, without understanding the underlying vulnerabilities that they exploit, does little to help developers remediate problems or prevent similar issues from occurring. Since vulnerabilities can result from a variety of different causes—including poor security requirements, flawed designs, implementation bugs, and insecure deployments—the model used to represent security issues must account for security considerations beyond the software itself, and should extend to consider the entire software development lifecycle and its greater context.

These challenges prompted the initial creation of our taxonomy, which was designed to support the precise communication of software security vulnerabilities. While our primary audience for this information was developers, we also wanted to provide meaningful vulnerability information to designers, testers and others involved in the software development process. This was the guiding principle behind the conception and evolution of our taxonomy - to communicate details that could be used to resolve errors that cause software vulnerabilities, and have the information understood by a diverse audience.

The Seven Pernicious Kingdoms taxonomy [15] was conceived as a simple hierarchical organization of security rules, classified according to code-level errors known to commonly impact the security of software systems. This hierarchical arrangement allowed us to make several enhancements to the taxonomy, which has now evolved into a platform for sharing security information across various software development lifecycle (SDLC) phases. In this paper we detail enhancements

that enable the taxonomy to cover the complete development lifecycle by analyzing relationships between mistakes made during the design, development and deployment of software systems, and the security vulnerabilities that result.

To achieve its design goals, the taxonomy adopts a nomenclature largely influenced by programming concepts, a characteristic that distinguishes it from the threat-focused approaches assumed by other taxonomies like the Web Application Security Consortium Threat Classification [30]. Also, the effectiveness of our approach is rooted in its careful organization of the secure coding rules, an attribute that does not apply to the list-based methods like MITRE's Common Vulnerabilities and Exposures (CVE) [5], which was built to detail information about specific vulnerability instances. Furthermore, unlike frameworks such as the Common Weakness Enumeration (CWE) [8], the taxonomy described in this paper advocates a categorization approach governed, not by the desire for rigorous comprehensiveness, but by its relevance and accessibility for developers. Section 2 reviews these and other related software security classifications.

Section 3 covers the design goals that drove the creation and development of our taxonomy. Section 4 defines the core terminology and describes the classification scheme in greater detail. The enhancements that have further strengthened this scheme since its conception are covered in section 5; we summarize our conclusions in Section 6.

## 2. Related Work

Enumerated reference lists, hierarchical classification schemes, ranking systems and ontological frameworks are some of the common approaches employed by the industry or proposed by academia to manage the growing population of security threats and vulnerabilities. Foundational works discussing vulnerability classification techniques were largely focused on Operating Systems vulnerabilities [1, 24, 25]. Other schemes focused specifically on vulnerabilities in network protocols [18, 20, 23] and security incidents [14]. Genesis, time of introduction and location were the proposed taxonomic criteria in a highly influential work by Landwehr *et al* [3] that addressed the classification of program security faults. This work inspired several subsequent classification schemes applied to software vulnerabilities [8, 11, 15, 27].

Krsul [13] and Bishop *et al* [18] focused their work on further refining the definition of a security taxonomy to address the ambiguity in prior classification schemes. As the population of vulnerabilities and the nature of automated vulnerability assessment evolved, we saw a gradual shift towards more useful and practical systems. A few efforts at addressing vulnerability classification within focused domains also emerged including Web Services [4] and Service Oriented Architecture vulnerabilities [17]. We examine the efforts closest to our current work in the following paragraphs.

The Web Application Security Consortium (WASC) Threat Classification [30] system was developed to standardize the terminology used to describe attacks against web applications. The system organizes known attacks into six top-level classes:

- Authentication
- Authorization
- Client-side Attacks
- Command Execution
- Information Disclosure
- Logical Attacks

Moreover, it uses the top-level classes to categorize 24 specific attack types. WASC subsequently released a second version that abandoned the hierarchy and instead offers two separate views for browsing the list of attacks. This version defined 25 attacks as well as 15 weaknesses that could potentially expose web sites to risk. The "Enumeration View" presented all the attacks and weaknesses in a simple list form. The "Development Phase View" offered a grid view that loosely associated the attacks and weaknesses to the development phase where they were most likely to be introduced.

Like the WASC Threat Classification, MITRE's Common Attack Pattern Enumeration and Classification (CAPEC) [2] is a taxonomy built to organize attack patterns. It aims to provide developers with an attacker's perspective. It can be used as a reference during threat modeling activities, but offers limited guidance about secure practices at the design or code level.

On the other hand, MITRE's Common Weakness Enumeration (CWE) [8] is an extensive collection of weaknesses known to expose software systems to security risks. While the "Development Concept View" borrows its organizational structure from several existing approaches (including the Seven Pernicious Kingdoms) the "Research Concepts View" uses an organization technique unique to CWE. This view arranges weaknesses according to the "abstractions of software behavior" and the resources that bear the consequences of these behaviors. This results in a hierarchical arrangement of weaknesses several levels deep, with each weakness characterized by a suitable level of abstraction – *Class*, *Base* or *Variant*. The relationships between the weaknesses model the inter-dependencies revealed through their abstraction. The Research view is designed purely for facilitating vulnerability research and identifying missing weakness definitions indicated by gaps between abstraction levels. This view is, by design, devoid of any information relevant to developers.

The Common Vulnerabilities and Exposures (CVE) on the other hand is a repository of specific vulnerabilities in commercial and open source third-party applications that have been disclosed publicly [5]. It uses a simple reference identifier system with each CVE entry consisting of details on the vulnerability and information about affected products. It acts as a useful asset during the operations and maintenance phases of the software lifecycle to help identify vulnerable software components and ensure timely installation of security patches. It is, however, focused on security defects in vendor software and hence does not aim to guide developers in the creation of custom applications securely.

The OWASP Top 10 [22] and CWE/SANS Top 25 [9] strive to rank web application and software security risks respectively. Through the evaluation of several criteria, including the number of applications vulnerable to a particular defect, the probability of the defect being exploited, and the severity of the damage it could cause, these projects pare down the overwhelmingly large number of vulnerabilities to a select few that have been deemed the most critical. The obvious shortcoming of this approach is the limited breadth of exposure it offers developers, which could lead to a false sense of security if misinterpreted.

The attack-based terminology utilized by some of the aforementioned strategies is intended for an audience of security practitioners who are responsible for assessing applications for software security errors. Consequentially, their effectiveness as a medium for communicating security risks in an enterprise setting is undermined by requirement that developers familiarize themselves with this terminology.

A system for classifying software vulnerabilities according to their causes into a two level hierarchy was previously proposed by Piessens [11]. They apply the time of introduction, a criteria first pioneered by Landwehr *et al* [3], at the top level to classify software errors according to the SDLC phase they were introduced in. The categories proposed in this study were, per the author's own admission, overly broad and in need of refinement.

The hierarchical arrangement of the Seven Pernicious Kingdoms taxonomy is specifically designed with the goal of educating software developers about software security issues using terms and concepts that are meaningful to them. As a corollary to this goal, the development-focused terminology that formed the basis of this classification scheme also serves as a platform for sharing security information across the different phases of the development lifecycle. Furthermore, the concepts used by all of the above organization systems can be mapped to the top-level classes of the hierarchy in our taxonomy thus enabling security practitioners, fluent in attack-based terminology, to translate and communicate requisite information about security findings to the entities involved in their mitigation.

## 3. Motivation

Organizations need an intuitive vocabulary for security defects that can be accurately and easily understood throughout the software development lifecycle. This vocabulary must effectively encapsulate both risk and remediation information, without requiring that software developers become security experts.

Software security defects can originate in any phase of the development lifecycle. Most mature software security initiatives employ multiple automated and manual assessment techniques throughout the development lifecycle to ensure comprehensive coverage. Outcomes for an initiative can be vastly improved by addressing a few critical challenges.

### 3.1 Terminology

An effective software security initiative typically involves multiple diverse sources of vulnerabilities. The results produced by different analysis approaches do not always use consistent terminology, which compounds the typical challenges around timing and process faced when addressing any defect. In the absence of an agreed upon terminology, divergence in security findings reported from different sources can hinder review efforts, prevent the generation of risk metrics, and complicate the prioritization of remediation efforts. Most damaging of all, confusing terminology can result in misinterpretation of the reported findings thus leading to incorrect or incomplete mitigations.

### 3.2 Information exchange

One of the toughest challenges security initiatives face is managing the vulnerability workflow from discovery to remediation. The process spans identifying findings, which are often discovered through numerous analysis techniques and have root causes ranging from design flaws to misconfiguration, all the way through to verifying fixes are properly deployed. The process requires communicating vulnerability details, their impact, and appropriate remediation techniques to a diverse group that often includes architects, developers, testers, and operations staff. Not all members of this audience will be well versed in security.

### 3.3 Requirements definition

NIST research shows [21] that the later a defect is identified in the development lifecycle, the more expensive it is to fix. Minimizing the cost to fix security defects therefore demands a proactive process that pushes the prevention or discovery of vulnerabilities as far upstream in the SDLC as possible. Without an efficient mechanism to organize the ever-expanding population of vulnerabilities, organizations struggle to identify the necessary security controls at the design phase to avoid or mitigate security defects in implementation.

### 3.4 Assessment automation

Automation plays an important role in assessing and improving the security of software. The effectiveness of an automated

solution is governed, in large part, by its ability to accurately communicate details about its findings. Without a common and consistent vocabulary for describing software security defects, users of automated solutions face a massive learning curve in order to comprehend and effectively act on findings from new or different sources.

### 3.5 Developer education

Educating developers about secure coding practices is a crucial component of any software security initiative. In addition to formal training, automated solutions can provide a valuable platform for teaching developers the secure way to implement different functionality. A common vocabulary for software security defects allows tool findings to reinforce concepts covered in training and permits organizations to more accurately assess what knowledge is transferred and retained by developers.

These five challenges inspired and shaped the design of our taxonomy. Specifically, we set out to establish a taxonomical framework that can:

- Express security defects in a way that is consistent and accessible to diverse stakeholders throughout the SDLC.
- Establish a common terminology applicable across disparate analysis techniques, both manual and automated.
- Promote advances in the capabilities and use of automated analysis by classifying security rules in structured ways.
- Describe software security mistakes with the goal of preventing them in design or avoiding their reintroduction.

Our taxonomy's benefits are rooted in its approach for classifying security rules according to the coding errors responsible for security vulnerabilities. The hierarchy used by the taxonomy for classifying security rules, and the use of a nomenclature driven by programming concepts, allows this simple, intuitive scheme to be incorporated into the SDLC as a consistent information exchange platform. At the same time, the architecture of the taxonomy means that it is infinitely extensible and is therefore highly amenable to future expansion.

## 4. Approach

### 4.1 The Seven Pernicious Kingdoms

The classification strategy used by the Seven Pernicious Kingdoms taxonomy is based on code-level security issues that occur in software applications. We have previously shared the design principles behind a simple hierarchy-based taxonomy developed to help teach developers about software security issues through a functional organization of security rules [16].

The taxonomy organizes security rules into a two-level hierarchy. The topmost level defines the kingdoms, each of which is a collection of security issues related to a certain security principle or class of mistakes. The second level offers further detail by categorizing every security issue into a phylum that represents a specific type of coding error.

Seven of the kingdoms are dedicated to errors in source code, and the final kingdom is related security issues caused by the program's configuration or environment. We present the kingdoms in order of importance to software security:

#### 1. Input Validation and Representation

The failure to account for untrusted input, when developing user-driven functionality, is responsible for some of the most severe security vulnerabilities. These issues include: buffer overflows, cross-site scripting attacks, SQL injection and many others. This kingdom covers issues caused by improper input filtering, alternate encodings and numeric representations.

### 2. API Abuse

An API establishes a set of rules that both callers and callees need to follow. It specifies what actions a caller is expected to take and what properties a callee needs to have in order for the API to be used securely. Disobeying such rules results in API abuse. Examples of relevant defects include cases of both the caller violating the agreement (e.g. not checking the value returned from the callee) and the callee breaking the rules (e.g. missing a check for a null parameter inside an equals() method).

### 3. Security Features

Availability of security features for protecting critical operations of a software system accelerates the development of secure software. Insecure use of such features can, however, completely defeat their purpose and leave the software system unprotected. Here, we are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management.

### 4. Time and State

As software systems become more complex and distributed, especially with the arrival of cloud and mobile computing, secure interactions between and within the components of the system that happen through shared state over a period of time become more imperative than ever before. Race conditions, synchronization issues, and data leakage between user sessions are examples of defects represented by this kingdom.

### 5. Errors

Error messages play an important part in facilitating the reconnaissance activities conducted by attackers. Application misbehaviors resulting from poor or missing error handling belong to this kingdom of security vulnerabilities. Owing to their omnipresence and proven track record of facilitating security compromises they were deemed to deserve an exclusive kingdom.

### 6. Code Quality

There is a fine line between quality and security. Poor code quality does not always imply insecure code, but can often point at serious security vulnerabilities. The infamous “Apple goto fail” vulnerability [7] is an excellent example that emphasizes the importance of code quality issues like dead code, for security. In the same way, memory leaks are indicators of potential denial of service problems.

### 7. Encapsulation

Errors that allow functionality or data to cross trust boundaries established to prevent contamination are grouped under this kingdom. Escalation of data access privileges, insecure import of untrusted code and exposure of information through unprotected resources are a few examples that characterize this kingdom.

### *. Environment

The environment kingdom groups classes of security issues that result from actions performed outside of code implementation such as insecure deployment practices. Configuration mistakes resulting in security vulnerabilities are a prime example. Such issues lie outside of the source code and as such are not covered by the rest of the kingdoms.

## 4.2 Examples

The hierarchy in the taxonomy provides multiple levels of granularity in the way it categorizes security issues. Kingdoms offer a coarse-grained classification of security issues according to the functionality they impact and security principles they violate. The rules that govern the assignment of issues to a specific phylum, on the other hand, offer a more finely grained categorization. The following examples are designed to further clarify this distinction.

Expression Language Injection: Applications perform inadequate input validation can be exposed to remote code execution attacks through injection of Expression Language (EL) constructs [10, 26]. This vulnerability is a result of an unsafe double evaluation of EL expressions performed by the EL interpreter. An attacker can target any application that accepts user input through EL constructs of the form
<spring:message scope="${param.foo}"/>,
where the value of the scope attribute is read from the request parameter named foo. By passing malicious EL expressions through this parameter value, an attacker can exploit the double evaluation flaw and remotely execute code within the context of the target application.

In addition to a bug in the EL interpreter, this vulnerability also highlights the application's failure to validate user-supplied input resulting in the EL expressions. By classifying this issue into the Expression Language Injection phylum under the Input Validation and Representation kingdom, the taxonomy helps us establish a relationship between the specific attack type of EL injection and the coding error that enables this attack – lack of proper validation.

Server Side Request Forgery: Often security testers and automated tools employ dynamic analysis techniques to identify vulnerabilities. This approach involves subjecting an application to different attack payloads that expose a specific type of vulnerability. Consider the following examples:

1. Port scanning attacks enabled through the exploitation of the WordPress pingback XMLRPC API [6].
2. Remote command execution through PHP fastcgi [29] using the "data" URL scheme.
3. An SMB relay attack against the Java HTTP-client running on Microsoft Windows [28].

Each of the attacks mentioned above employs a specific exploit payload to attempt a specific type of compromise. The first scenario can be used by an attacker for the purposes of conducting reconnaissance, the second remotely executes PHP commands, and the third allow the theft of NTLM credentials. Despite differences in the exploit vectors and the nature of the final compromise, all of these attacks are enabled by a single security flaw – the failure of an application to validate user-supplied input before allowing it to specify the source URL for subsequent network activity. By classifying all of these attack instances into the *Server-Side Request Forgery* phylum under the *Input Validation and Representation* kingdom, the taxonomy draws developer attention to the underlying security defect responsible for the compromise rather than a particular payload that led to its discovery or exploitation [12].

In contrast with the previous example, demonstrating the role of a phylum in representing a specific vulnerability type,

kingdoms are used to group together distinct classes of vulnerabilities. *Expression Language Injection*, for instance, represents a vulnerability type that allows attackers to execute arbitrary dynamic code inside the target application's context. On the other hand, the *Server-Side Request Forgery* issue is often exploited to gain control of outbound network connections from a target server. Despite this distinction, by grouping both types of issues under the *Input Validation and Representation* kingdom, the taxonomy focuses on the common root cause responsible for both of weaknesses and allows developer to learn about multiple damaging outcomes caused by a single kind of coding error.

The ability to group security issues according to common code-level errors and also identify specific attack types that arise from programming mistakes comprise the biggest benefit of our taxonomy's hierarchical classification. Through better understanding of these relationships, developers can assess their code during development and avoid repeating mistakes that are known to cause specific classes of security issues.

Using a terminology based on programming concepts encourages a proactive approach to building secure software, which is highly desirable since the earlier a defect is discovered, the less expensive it is to repair. To understand how the taxonomy helps accomplish this, consider the challenge of designing software that deals with sensitive information. A robust access control system is crucial to ensuring secure handling of such information. Our taxonomy classifies the numerous mistakes that could allow attackers to gain unauthorized access to sensitive information under the *Access Control* phylum. This phylum belongs to several different kingdoms each indicating the kind of design, code or configuration-related mistakes to avoid. A careful study of all the rules classified under this phylum helps the stakeholders responsible for the various stages make secure choices, ensure safe implementation, and adhere to secure deployment practices. By learning about and explicitly stating the access control requirements to be met at each stage, development teams can avoid mistakes that would otherwise be discovered much later in the process, when they may incur heavy costs.

Further benefits of our taxonomy are illustrated through its application to the organization of rules that guide automated vulnerability assessment activities. Since software security issues originate at various stages of the software development lifecycle, ensuring comprehensive detection coverage requires use of multiple assessment techniques, including manual analysis and penetration testing. With no means to achieve consistency in reporting the issues discovered by the multiple methods, the task of auditing results can quickly become overwhelming. By using a classification scheme such as ours, automated solutions can reap the benefit of an organizational platform that enables automated sharing of information. It also offers a consistent terminology that can accurately communicate findings identified using multiple techniques and is completely agnostic to the dissimilarities between discovery methods.

Transparent interaction between security testers and development teams is a key prerequisite to quick resolution of security bugs, especially when multiple automated and manual bug hunting strategies are employed to detect security defects. For example, when a security team performs an automated assessment and identifies a privilege escalation exploit, depending on the nature of the underlying vulnerability, they might need to take one or more of the following actions:

- Communicate their finding to the designers and/or architects of the system to highlight the lack of a robust authorization system in the design
- Convey the error (that could allow attackers to execute critical functionality without the necessary privileges) in the application logic to the developers
- Alert the operations team regarding the deployment of an outdated version of an authorization framework known to be susceptible to publicly-disclosed vulnerabilities

- Convey the perceived risk to executive management in order to adequately prioritize and authorize remediation activities

Performing such a detailed root cause analysis and clearly communicating the results to appropriate stakeholders requires a highly motivated group of security experts with ample time to analyze each individual finding - an impractical strategy when dealing with a large number of applications as is often the case in a modest sized enterprise. Instead, through the integration of our taxonomy, an automated assessment tool can categorize the findings to clearly indicate the underlying source of the issue. Additionally, the solution can use the terminology to offer comprehensive analysis and mitigation advice, regardless of the method of detection.

### 4.3 A Unified Taxonomy

The hierarchical arrangement of the Seven Pernicious Kingdoms taxonomy detailed previously has not only allowed the taxonomy to successfully realize the primary design goals established during its conception, but has also allowed flexibility that has enabled us to successfully apply it to new challenges.

The original design of our taxonomy laid the foundation for a framework that, with minor enhancements, enables the automatic categorization of vulnerabilities and supplies a vocabulary capable of withstanding the varying interpretations of different roles throughout the development lifecycle. A key requirement established during the development of the Seven Pernicious Kingdoms taxonomy was to forgo theoretical completeness in favor of simplicity. This, however, does not act as a deterrent to the further expansion of the taxonomy's coverage beyond code-level errors. The hierarchical structure and the nested levels have been specifically designed to make the taxonomy adaptable to new categories of vulnerabilities, new domains as well as the application of software to new verticals. This same property fuels the ability of the taxonomy to classify security issues independent from the lifecycle phase where the vulnerability was introduced, as well as when and how it was discovered.

To understand the rationale behind the enhancements made to the classification scheme, consider the access control example mentioned previously. Gaps in access control mechanisms can result from either a design flaw in the form of missing authentication, a failure to account for manipulated user inputs during the implementation of an authentication feature, or configuration mistakes made during deployment. Such dissimilarities in the kinds of errors that span the architecture, implementation and deployment activities stress the need for a versatile classification system. The flexibility ingrained into the hierarchical design of our taxonomy has allowed us to extend it for exactly such purposes. By enhancing the *Environment* kingdom with the following access control related phyla, the taxonomy's classification now accounts for deployment-related mistakes in addition to code-level errors, including:

- Access Control: Administrative Interface. Failure to restrict access to administrative interfaces can allow attackers to perform unauthorized administrative functions.
- Access Control: Cookie Authentication Bypass. Authentication mechanisms relying on static cookies can allow attackers to easily spoof session identifiers and gain unauthorized access to resources.
- Access Control: Information Disclosure. Failure to restrict access to files could lead to exposure of sensitive information or interfaces.
- Access Control: Missing Authentication. Failure to authenticate users requesting access to sensitive information or privileged functionality can allow attackers to perform unauthorized actions.

- Access Control: Open Proxy Access. A proxy server allowing CONNECT or GET requests without any restriction can enable an attacker to hide the origin of their attacks or access resources internal to the network.
- Access Control: Unprotected Directory. Insecure deployment and exposure of directories can supply attackers with invaluable information about the application enabling them to conduct targeted attacks.
- Access Control: Unprotected File. Permitting undeterred access to files could reveal critical information to attackers.
- Access Control: Weak Authentication. Flawed authentication checks could allow attackers unauthorized access to sensitive functions or alter system behavior.

Not all issues can be reliably detected through analysis of source code. Certain categories of issues require analysis of application behavior. The following phyla introduced under the *Encapsulation*, *Security Features* and *Environment* kingdoms illustrate the taxonomy's ability to support such cases:

- Transport Layer Protection: Insecure OAuth Communication Channel. Insecure transmission and storage of OAuth tokens or secrets could disclose authentication information needed to take privileged actions on behalf of the user.
- Transport Layer Protection: Insecure Transmission. Transferring any sensitive information over a non-secure channel, could expose it to theft by attackers.
- Transport Layer Protection: Secure Section Must Require SSL Access. Allowing communication with sensitive components of the application over an unencrypted channel could compromise access restrictions and lead to the leakage of information.
- Transport Layer Protection: SSL Policy Enforcement Issue. An insecure environment where secure content is delivered alongside non-secure content could allow attackers to hijack secure communications.
- Transport Layer Protection: Weak SSL Cipher Detected. Using a weak cipher or encryption key could allow an attacker to defeat the protection mechanism and steal or modify sensitive information.
- Transport Layer Protection: Weak SSL Protocol Detected. Use of insecure protocol versions weakens the strength of the transport protection and could allow an attacker to steal or modify sensitive information.
- Web Server Misconfiguration: Missing Content-Type. Failure to specify the content type of an HTTP response could allow an attacker to declare a content type of their own choosing to execute malicious code on client systems.
- Web Server Misconfiguration: Missing Unicode Charset. Allowing an attacker to control the character set, can lead to client-side execution of malicious payloads that might not be caught by the server's input filter.

As software security threats evolve, so do security analysis approaches. Various static code analysis techniques are helping developers identify security-relevant errors during implementation. Runtime analysis and dynamic security assessments are aiding testers in recognizing insecure application behavior and assisting operations teams in locating insecurely-configured environments. The maturity of these approaches has prompted advances in interactive automated assessment. Adoption of a common vocabulary is a fundamental requirement for solutions that propose to exchange of security knowledge and observations automatically. The flexible hierarchy of our taxonomy allows it to capture such progression in the field and the changes described in this section ensure that it accounts for the breadth of software errors identified by different analysis techniques. Using this taxonomy to encode their security findings, automated tools can begin to communicate observations in real-time without risk of misinterpretation.

Lastly, we recognize the difficulty of attaining a holistic interpretation of the risk to enterprise software systems when multiple methods of security assessment are leveraged. The use of automated solutions that employ multiple techniques to scrutinize software systems for security vulnerabilities is driven by the desire for broad analysis coverage. However,

inconsistent terminology used to describe the findings discovered using distinct assessment methods contributes to the complexity of collectively analyzing the results and assessing the overall health of software systems. The core design of our taxonomy and the enhancements described here lend a consistent terminology and the classification support needed to cover various types of vulnerabilities that demand the use of different analysis techniques. Adoption of such taxonomy for the classification of security issues across different solutions can help alleviate the challenges posed by risk assessment endeavors involving automated tools.

## 5. Conclusion

For over ten years, the Seven Pernicious Taxonomy has helped developers learn about the coding errors responsible for software security flaws. Its simple intuitive hierarchy has allowed it to seamlessly represent security issues in new programming languages, technologies, and frameworks. Furthermore, its ability to support issues identified using distinct detection techniques makes it an effective tool for capturing security weaknesses across all phases of the development lifecycle.

The taxonomy will continue evolving to further narrow the gap between the people and technologies involved in creating secure software. Future work will remain developer-focused approach to ensure consistent interpretation of findings from automated assessments and enabling automated information exchange between different security analysis techniques and lifecycle phases.

## 6. References


[1] Aslam, T. 1995. A Taxonomy of Security Faults in the Unix Operating System. M.S. thesis, Purdue University.

[2] CAPEC – Common Attack Pattern Enumeration and Classification. http://capec.mitre.org/.

[3] 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.

[4] C. V. Berghe, J. Riordan, F. Piessens. A Vulnerability Taxonomy Methodology applied to Web Services. In Helger Lipmaa, Dieter Gollman, editor, Proceedings of the 10th Nordic Workshop on Secure IT Systems (NordSec 2005).

[5] CVE – Common Vulnerabilities and Exposures. http://www.cve.mitre.org/.

[6] CVE-2013-0235. Server Side Request Forgery issue in Wordpress XMLRPC API. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0235.

[7] CVE-2014-1266. Apple SSL bug. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266.

[8] CWE – Common Weakness Enumeration. https://cwe.mitre.org/.

[9] CWE/SANS TOP 25 Most Dangerous Software Errors. http://www.sans.org/top25-software-errors/.

[10] Expression Language Injection Advisory - http://support.springsource.com/security/cve-2011-2730.

[11] F. Piessens. A taxonomy (with examples) of causes of software vulnerabilities in internet software. Technical Report 346, Department of Computer Science, K.U.Leuven, 2002.

[12] HP Fortify Taxonomy: Software Security Errors. http://www.hpenterprisesecurity.com/vulncat/en/vulncat/index.html

[13] I. V. Krsul. Software Vulnerability Analysis. Ph.D. Dissertation, Computer Sciences Department, Purdue University, Lafayette, IN, May, 1998.

[14] J. Howard, T. Longstaff. A Common Language for Computer Security Incidents. Sandia National Laboratories, Albuquerque, N.M., 1998.

[15] J. Viega. The CLASP Application Security Process. Volume 1.1 Training Manual.
[16] K. Tsipenyuk, B. Chess, G. McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," IEEE Security & Privacy, vol. 3, no. 6, pp. 81-84.
[17] L. Lowis and R. Accorsi. On a Classification Approach for SOA Vulnerabilities. In International Computer Software and Applications Conference, pages 439-444, 2009.
[18] M. Bishop. A Taxonomy of Unix System and Network Vulnerabilities, Technical Report CSE-9510, Department of Computer Science, University of California at Davis, May 1995.
[19] M. Bishop and D. Bailey. A critical analysis of vulnerability taxonomies. Technical Report 96-11, Department of Computer Science, University of California at Davis (Sep. 1996).
[20] Neumann, P.G. Computer System Security Evaluation. National Computer Conference (AFIPS Conference Proceedings 47), pp. 1087–1095 (June 1978).
[21] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing, May 2002.
[22] OWASP Top Ten Most Critical Web Application Security Vulnerabilities.
[23] Pothamsetty V, Akyol, BA (2004): A vulnerability taxonomy for network protocols: Corresponding engineering best practice countermeasures. Communications, Internet, and Information Technology 2004. St. Thomas, US Virgin Islands.
[24] R. Bisbey and D. Hollingworth. Protection Analysis Project Final Report. ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, 1978.
[25] 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.
[26] Remote Code Execution with Expression Language Injection. http://danamodio.com/application-security/discoveries/spring-remote-code-with-expression-language-injection/.
[27] S. Weber, P.A. Karger, and A. Paradkar, “A Software Flaw Taxonomy: Aiming Tools at Security,” Proc. Workshop Software Eng. for Secure Systems, 2005.
[28] SMBRelay Bible 7: SSRF + Java + Windows = Love. http://erpscan.com/press-center/smbrelay-bible-7-ssrf-java-windows-love/.
[29] V. Vorontsov, A. Golovko. SSRF attacks and sockets: smorgasbord of vulnerabilities. http://2012.zeronights.org/includes/docs/Vorontsov%20Golovko.pdf.
[30] Web Application Security Consortium Threat Classification. http://projects.webappsec.org/w/page/13246978/Threat Classification.