

Studying Software Vulnerabilities

Dr. Robin A. Gandhi, Dr. Harvey Siy, and Yan Wu
The University of Nebraska at Omaha

There have been several research efforts to enumerate and categorize software weaknesses that lead to vulnerabilities. To consolidate these efforts, the Common Weakness Enumeration (CWE) is a community-developed dictionary of software weakness types and their relationships. Yet, using the CWE to study and prevent vulnerabilities in specific software projects is difficult. This article presents a novel approach for using the CWE to organize and integrate the vulnerability information recorded in large project repositories.

Many vulnerabilities in today's software products are rehashes of past vulnerabilities. Developers are often unaware of past problems or they are unable to keep track of vulnerabilities that others have reported and solved. Interestingly, this is not because of a scarcity of information. In fact, a plethora of information about past vulnerabilities is available to developers. Most software development projects dedicate some effort to documenting, tracking, and studying reported vulnerabilities. This information is recorded in project repositories, such as change logs in source code version control systems, bug tracking system entries, and mailing list communication threads. As these repositories were created for different purposes, it is not straightforward enough to extract useful vulnerability-related information. In large projects, these repositories store vast amounts of data, oftentimes burying the relevant information. Therefore, efforts to summarize lessons learned from past vulnerabilities in a software project are essentially non-existent. In the face of growing software complexity, it is even more critical to improve the mental model of the software developer to sense the possibility of vulnerability.

The CWE standardization effort provides a unified and measureable set of software weaknesses for use in software assurance activities [1]. CWE is a community-driven and continuously evolving taxonomy of software weaknesses. According to [1], the CWE vision is twofold, enabling:

- A more effective discussion, description, selection, and use of software security tools and services that can find weaknesses in source code and operational systems.
- A better understanding and management of software weaknesses related to architecture and design.

However, the CWE is often compared to a kitchen sink, as it aggregates weakness categories from many different vulnerabil-

ity taxonomies, software technologies and products, and categorization perspectives. While the CWE is comprehensive, using its highly tangled web of weakness categories is a daunting task for stakeholders in the software development life cycle (SDLC).

The unique characteristics of a weakness—its preceding design or programmer errors, resources/locations that the weakness occurs in, and the consequences that follow the weakness (such as unauthorized information disclosure, modification, or destruction)—are either expressed together within a single CWE category or spread across multiple categories. Such complexity makes it difficult to trace the information expressed in the CWE to the information about a discovered vulnerability in multiple project-specific sources (such as a log of code changes, source code differences, developer mailing list discussions around bugs, bug-tracking databases, vulnerability databases, and public media releases). Therefore, to facilitate CWE use in the study of vulnerabilities, we have developed easy-to-understand templates for each conceptually distinct weakness type. This template can then be readily applied to aggregate and study project-specific vulnerability information from source code repositories.

Each template is a collection of concepts related to a single weakness type. The concepts are identified by extracting and distilling information from all relevant CWE categories for a particular weakness type. Since the concepts in the templates provide meaning to the usage of certain words and sentences that describe vulnerability information, we call them *semantic templates*.

While the CWE is a collection of abstract categories, the Common Vulnerability Enumeration (CVE) is an ever-growing compilation of actual known information security vulnerabilities and exposures, as reported by software development organizations, coordination centers, developers, and individuals at

large. CVE assigns a common standard identifier for each discovered vulnerability to enable data exchange between security products and provide a baseline for evaluating coverage of tools and services [2].

In this article, we outline the process of building a semantic template to study the injection software weakness type. In recent times, injection is the single most exploited weakness type. It occurs upon failure to adequately filter user-controlled input data for syntax that can have unintended consequences on the program execution. As stated in CWE-74, a distinguishing characteristic of the injection weakness is that “the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism” [3]. For example, consider a Web application that accepts user input to dynamically construct a Web page that is instantly accessible to other users. Web blogs, guest books, user comments, and discussion pages typically provide such functionality. If the user input that gets included in the dynamic construction of a Web page is not appropriately sanitized for HTML and other executable syntax (e.g., JavaScript), then active user-chosen Web content (such as redirection to malicious Web pages) can be injected into the Web application and later served to other clients that load the tainted Web page in their browsers. This instantiation of the injection weakness is most commonly referred to as cross-site scripting (XSS). As observed in CWE-79 [4], the structure of the dynamically generated Web page is altered by sending code (HTML and JavaScript) in through legitimate user input channels to the Web application.

We also discuss the application of the injection semantic template to study artifacts related to a confirmed XSS vulnerability (CVE-2007-5000, see [5]) in the Apache HTTP Server project. For the interested reader, we have previously elaborated on the buffer overflow semantic template in [6].

Building a Semantic Template

When it comes to security vulnerabilities, we face an interesting paradox. On one end, we are inundated with discovered vulnerability information from its detection to its fix. On the other end, there is most often a lack of security knowhow among stakeholders in the SDLC. We realize that during software development, especially in the implementation stage, the details a programmer has to remember to avoid security vulnerabilities can be enormous. The mere existence of long checklists and guides (such as the CWE) is not enough. To deal with enormous details, the use of long checklists needs to be facilitated by simple cognitive guides or templates. Therefore, to effectively and quickly study the large amounts of information associated with vulnerabilities, we ask the following four fundamental questions:

1. What are the software faults? In other words, what are the concrete manifestations of errors in the software program and design related to omission (lack of security function), commis-

sion (incomplete security function), or operational (improper usage) categories that can precede the weakness?

2. What are the defining characteristics of the weakness?
3. What are the resources and locations where the weakness commonly occurs?
4. What are the consequences? In other words, what are the failure conditions violating the security properties that can be preceded by the weakness?

Answers to these questions are highly tangled in current CWE documentation. For each major class of weakness (such as injection), a large number of CWE categories can be identified to find answers to these questions. As a result, a significant amount of work is needed to identify the trail of CWE categories such that the chain of events that lead to a vulnerability can be reconstructed. To facilitate such analysis, the creation of a semantic template can be viewed as a systematic process of untangling the CWE categories and their descriptions into different bins that

correspond to the four questions. We first describe the preparation and collection phase of building a semantic template.

Preparation and Collection Phase

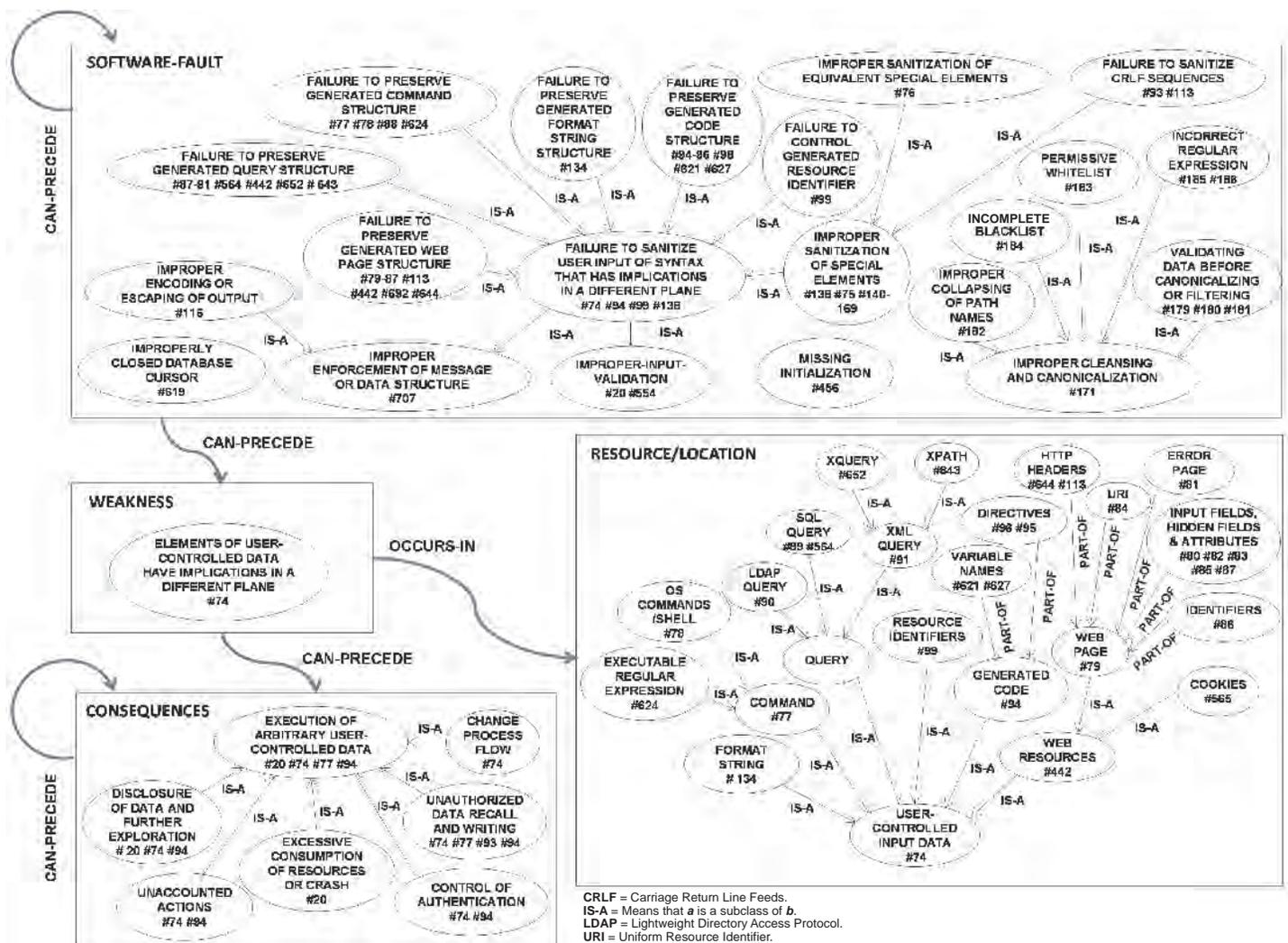
Selection of Content

Since the CWE is continuously evolving, it is important to note that our template is based on Version 1.6 [7]. The CWE uses views to integrate multiple categorizations of weaknesses that share several CWE categories. We use the two most prominent views of the CWE: the development view (CWE-699) of CWE categories, suited for practitioners in the SDLC, and the research view (CWE-1000), suited for research purposes (as it has a deep and abstract hierarchical structure).

Extraction of Relevant Weaknesses

The next step is to identify the CWE category that identifies the weakness of interest at the most abstract level. For the *Injection* weakness, CWE-74 is such a category [3]. Referred to as the root category,

Figure 1: Injection Semantic Template



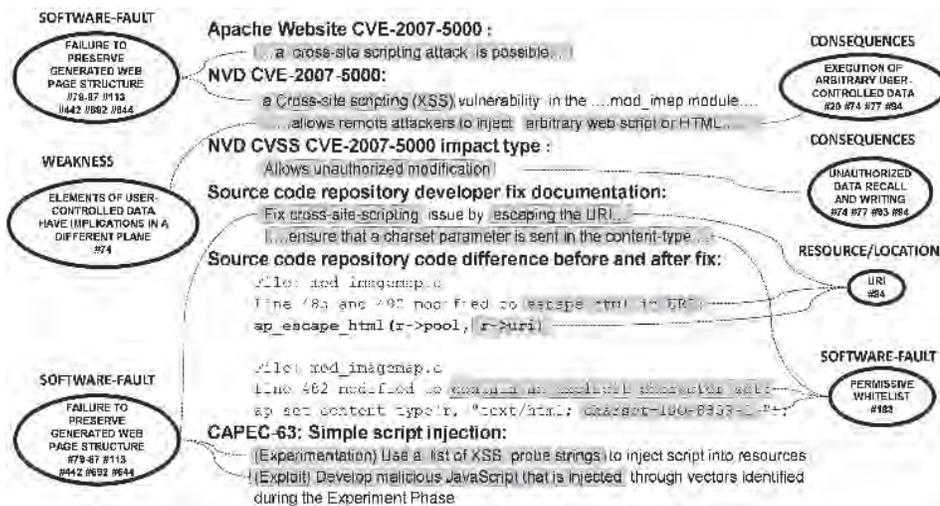


Figure 2: Annotation of Information Pieces for Vulnerability CVE-2007-5000 with Concepts of the Injection Semantic Template

we start here and adopt four strategies to gather weaknesses related to it in the CWE development and research views:

1. Navigate hierarchical relationships of the root category (*Parent* and *Child Of*).
2. Navigate non-taxonomical relationships such as *Can Precede*, *Can Follow*, *Peer-of* in the CWE hyperlinked document [7].
3. Keyword search on the CWE document [7] for weaknesses that have the injection weakness described in their primary or extended description. Keyword search is followed by exploration of *Parent*, *Sibling*, and *Child* categories of the discovered CWE category, for relevance to the root category.
4. Visualization [8] of the root category and its related weaknesses identified by automatically parsing the CWE specification available in XML [1].

While applying each strategy, use of heuristics and some degree of judgment is required on part of the subject matter expert to include a CWE category into the pool of relevant weaknesses. Details about the CWE categories—discovered by applying our strategies to gather weakness related to the root category CWE-74—can be found at <<http://faculty.ist.unomaha.edu/rgandhi/st/injectioncwe.pdf>>. Table 1 gives some summary statistics roughly describing the scale of the

work involved. It speaks volumes about the complexity of the *mental model* that developers need to be aware of to understand the consequences of their coding and design decisions, such that injection weakness can be avoided. Although hyperlinked, navigating the CWE documentation and various graphical representations is tedious and non-intuitive. While different CWE views help to accommodate multiple perspectives, it adds an additional layer of complexity.

Template Structuring Phase Separation of Tangled CWE Descriptions

In this phase, the descriptions of the set of CWE categories from the previous phase are carefully analyzed for their correspondence to either a *Software Fault* that leads to injection; defining characteristic of the injection *Weakness*; *Resource/Location* where injection weaknesses occur; or *Consequences* that follow from a injection weakness. After these parts have been separated and placed in appropriate bins, well-formed and succinct concepts for the injection semantic template are identified in each bin. For example, by analyzing the descriptions for CWE-74 in [1], the following concepts (shown in quotes) can be systematically identified for each of the

semantic template conceptual units (shown in bold).

- **Software Fault:** “Failure to sanitize user input of syntax that has implications in a different plane.”
- **Weakness:** “Elements of user-controlled data have implications in a different plane.”
- **Resource/Location:** “User controlled input data.”
- **Consequences:** “Execution of arbitrary user-controlled data,” “Disclosure of data and further exploration,” “Unaccounted actions,” “Control of authentication,” “Unauthorized data recall and writing,” and “Change process flow.”

While some of these concepts overlap with the CWE-79, this category identifies the following unique and more specific concepts:

- **Software Fault:** “Failure to preserve generated Web page structure,” derived from CWE-79, is a more specific software flaw than a “Failure to sanitize user input of syntax that has implications in a different plane,” which is derived from CWE-74.
- **Resource/Location:** “Web page” (output that is served to other users), which is a “User controlled input data” that is addressed in CWE-74.

Filtering Concepts and Introducing Abstractions

The CWE categories are class, base, or variant weakness, with class being the most general. Class weaknesses are described in a very abstract fashion, typically independent of any specific language or technology. Base weakness is also described in an abstract fashion, but with sufficient details to infer specific methods for detection and prevention of the weakness. On the other hand, variant weaknesses are described at a very low level of detail, typically limited to a specific language or technology.

With the original intent of the semantic template to make weakness more understandable, we derive the primary concepts for software faults and weakness characteristics from the more general class and base CWE categories—while preserving traceability to the CWE categories (with more specific variants) using their identifiers. This design decision was taken primarily to avoid missing the forest for the trees. We expect it to be easier for developers to remember a more generic model of the weakness rather than a detailed one. However, in the case of the Resource/Location conceptual unit, it is not uncommon to extract concepts in the

Table 1: Measures Related to the Collection of Injection Related CWEs Measures

Measures	Value
Total number of CWEs relevant to injection	46
Total number of relationships among CWEs relevant to injection	37
Average number of relationships (inward and outward) per CWE	1.6
Highest depth of the hierarchy among CWEs relevant to injection	4 (including root)
Total number of pages in the CWE document relevant to injection	83

template from variant weaknesses. For the Consequences conceptual unit, we have discovered that the concepts extracted from consequences listed for class and base CWE categories provide comprehensive coverage of consequences identified from more specific-variant CWE categories.

Template Structuring and Representation

In this sub-task, the identified concepts for the template are structured and related to each other based on the relationships between their corresponding CWE categories. From this effort, a highly structured collection of interdependent concepts emerge (as shown in Figure 1 on page 17). Each concept in the semantic template of Figure 1 includes numbers that identify relevant CWE categories. The semantic template reduces duplication of content across related CWE categories while putting them in the context of each other.

Template Refinement and Tailoring

The template can be easily used to study vulnerability information gathered from multiple sources or reconstruct a successful software exploit. Related to both CWEs and CVEs, the Common Attack Pattern Enumeration and Classification (CAPEC) [9] provides a standard way to capture and communicate the manner in which software weaknesses can be exploited. They are stepwise operationalizations of attacks against software systems. By mapping specific vulnerabilities (CVEs) and attack patterns (CAPECs) to the semantic template, it is further refined and checked for obvious omissions. In the following section, we describe such mapping in the context of the XSS vulnerability from CVE-2007-5000. We also expect the semantic templates to be tailored for a specific project, product, or organization.

Using the Semantic Template to Study Vulnerabilities

We use the injection semantic template to study the vulnerability information available from multiple project specific sources for the reported XSS vulnerability CVE-2007-5000 in the Apache HTTP server. These sources include the CVE vulnerability descriptions; media reports about the vulnerability on the Apache HTTP server project public Web site; change history in the open source code repository; source code versions (before and after the fix); and related CAPECs as test cases. The semantic template allows us to anno-

Software Defense Application

As the government and defense sector adopts standards for tracking and detecting specific vulnerabilities, there is an urgent need for developers to build software artifacts to avoid weaknesses that cause vulnerabilities in the first place. Semantic templates have multiple usage scenarios in software assurance, such as to study past vulnerabilities in source code repositories, suggest test cases for a identified software resource, elicit requirements for avoiding weakness, and provide intuitive explanation-based guidance to developers when conditions that lead to weaknesses are detected.

tate the natural language vulnerability descriptions in order to understand and reconstruct the way the injection weaknesses occur. The semantic template also allows extrapolating or identifying missing information (if any).

The semantic template provides intuitive visualization capabilities for the collected vulnerability information. In Figure 2, the vulnerability artifacts related to CVE-2007-5000 are filled into the template. A larger visualization can be found at <http://faculty.ist.unomaha.edu/rgandhi/st/injectioncve.pdf>. Figure 2 provides an integrated view that shows how developers can effectively reason about why the vulnerability occurred; brainstorm possible attack vectors (CAPECs); and discuss the adequacy of performed fixes. Stakeholders in the SDLC can consume technical details with relative ease and guided explanation.

We expect that over a collection of CVE vulnerabilities in a particular project, their mappings to specific weakness categories will reveal recurring error patterns and provide project-specific measures for identifying the most prominent CWE weaknesses for which developers need awareness and training.

Synergy with Other Security Standardization Efforts

The semantic template provides a unified view of software weaknesses (CWE), actual vulnerabilities (CVE), and relevant attack patterns (CAPEC) that can be used to develop and prioritize risk-based test cases for the most exploited software flaws. Many source code static analysis tool reports now provide explicit mappings from their error reports to CWE and CVE identifiers. However, exploring a CWE category and its related weaknesses (with currently available textual and limited visualization formats) poses a significant burden to the tool users. To this end, the concepts in the semantic template maintain explicit traceability to CWE identifiers and hence can be used to provide an intuitive, visual, and layered explanation to the tool user in the context of the discovered flaw. The tool user can also

examine the fix information from past vulnerabilities to determine the course of action to take. In addition, mapping of attack patterns (CAPECs) to software faults in the semantic template provides concrete scenarios to test and justify the fix adequacy.

With the availability of the Malware Attribute Enumeration and Characterization [10] standardization effort and its mappings to the CWE, we expect to use the semantic template to study what software flaws most often contribute to successful malware behaviors and CAPECs. For example, the flaws that precede the injection weakness would most likely contribute to the success of malware behavior for delivering a malicious payload.

Currently, the process of encoding the known vulnerabilities and attack patterns into the template is manually performed. While manual population of templates is scalable for recording of new vulnerabilities as they are detected, relating past vulnerabilities with the templates requires automation. An empirical study with the Apache repository will be conducted to assess the accuracy of this automated process.

As part of our future work, we also expect to build associations of the semantic template with the Knowledge Discovery Metamodel (KDM) [11]. The KDM defines an ontology for software assets and their relationships; this could be leveraged to describe the software faults and resources in the semantic template using a language-independent semantic representation. In turn, the semantic templates could provide abstractions and visualizations to enhance the explanation of KDM-based software mining results.

Conclusion

The CVE grows by roughly 15 to 20 vulnerabilities every day. Each discovered vulnerability produces several information pieces extending from its discovery to its fix. With over 600 entries and more than 20 different views, the CWE provides a significant body of knowledge for classifying and categorizing software weaknesses. However, it is a difficult task

to use the CWE for conducting a systematic study of observed vulnerabilities.

This article describes a process to systematically study software vulnerabilities using several software assurance community standards. A semantic template enables us to systematically assimilate the information pieces related to a vulnerability. This integrated information allows fundamental questions to be answered:

- How do software flaws lead to a vulnerability?
- What are the consequences of exploiting the vulnerability?
- How were they exploited?
- What resources were involved?
- How were they fixed?
- Are the applied fixes sufficient?
- What project specific measures can be produced for the CWE weakness categories that the vulnerability is related to?
- How do the discovered vulnerability and its fix revise our confidence in the software system?
- What other weaknesses still remain?
- What steps should be taken to prevent the vulnerabilities in general?
- Can tools be optimized to look for the discovered patterns?

Answering these questions is essential for an organization to measure the effectiveness of its secure software develop-

ment activities and justify the corresponding assurance given to customers. ♦

Acknowledgement

This research is funded in part by DoD/Air Force Office of Scientific Research, National Science Foundation Award Number FA9550-07-1-0499, under the title “High Assurance Software.”

References

1. The MITRE Corporation. *CWE—Common Weakness Enumeration*. 10 Apr. 2010 <<http://cwe.mitre.org/>>.
2. The MITRE Corporation. *CVE—Common Vulnerabilities and Exposures*. 10 Apr. 2010 <www.cve.mitre.org>.
3. The MITRE Corporation. *CWE—Common Weakness Enumeration*. “CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component.” 5 Apr. 2010 <<http://cwe.mitre.org/data/definitions/74.html>>.
4. The MITRE Corporation. *CWE—Common Weakness Enumeration*. “CWE-79: Improper Neutralization of Input During Web Page Generation.” 5 Apr. 2010 <<http://cwe.mitre.org/data/definitions/79.html>>.
5. The MITRE Corporation. *CVE—Common Vulnerabilities and Exposures*.

“CVE-2007-5000 (under review).” 9 Sept. 2007 <<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5000>>.

6. Wu, Yan, Robin A. Gandhi, and Harvey Siy. *Using Semantic Templates to Study Vulnerabilities Recorded in Large Software Repositories*. Proc. of the 6th International Workshop on Software Engineering for Secure Systems (SESS ’10) at the 32nd International Conference on Software Engineering (ICSE 2010), South Africa, Cape Town. 2010.
7. Martin, Robert A. *CWE Version 1.6*. The MITRE Corporation. 29 Oct. 2009 <http://cwe.mitre.org/data/published/cwe_v1.6.pdf>.
8. Siy, Harvey. “Injection-Related CWEs – Graph-Viz Visualization.” <www.cs.unomaha.edu/~hsiy/research/zgrviev/injectionCWEs.html>.
9. The MITRE Corporation. *CAPEC—Common Attack Pattern Enumeration and Classification*. 18 May 2010 <<http://capec.mitre.org>>.
10. The MITRE Corporation. *MAEC – Malware Attribute Enumeration and Characterization*. 10 Apr. 2010 <<http://maec.mitre.org>>.
11. “KDM 1.1.” *Object Management Group*. 10 Apr. 2010 <www.omg.org/spec/KDM/1.1>.

About the Authors



Robin A. Gandhi, Ph.D., is an assistant professor of information assurance in the College of Information Science and Technology at the University of Nebraska, Omaha

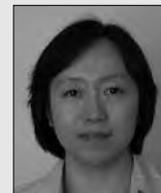
(UNO). He received his doctorate from The University of North Carolina at Charlotte. The goal of Gandhi’s research is to develop theories and tools for designing dependable software systems that address both quality and assurance needs. Gandhi is a member of the DHS’s Software Assurance Workforce Education and Training Working Group.

**Nebraska University Center for Information Assurance
College of Information Science and Technology (IS&T)
6001 Dodge ST
PKI 177 A
Omaha, NE 68182-0500
Phone: (402) 554-3363
E-mail: rgandhi@unomaha.edu**



Harvey Siy, Ph.D., is an assistant professor in the Department of Computer Science at the UNO. He received his doctorate in computer science from the University of Maryland at College Park. He conducts empirical research in software engineering to understand and improve technologies that support the development and evolution of reliable software-intensive systems. Siy has previously held positions at Lucent Technologies and its research division, Bell Laboratories.

**Department of Computer Science
College of IS&T
6001 Dodge ST
PKI 281 B
Omaha, NE 68182-0500
Phone: (402) 554-2834
E-mail: hsiy@unomaha.edu**



Yan Wu is currently pursuing her doctorate in information technology at the UNO, and is expecting to receive her degree in Spring 2011. The goal of her research

is to conduct empirical study on analyzing software engineering knowledge in order to support the development and maintenance of reliable software-intensive systems.

**Department of Computer Science
College of IS&T
6001 Dodge ST
Omaha, NE 68182-0500
E-mail: ywu@unomaha.edu**