Vulnerability of the Day: Concrete Demonstrations for Software Engineering Undergraduates

Andrew Meneely
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY 14623, USA
andy@se.rit.edu

Samuel Lucidi
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY 14623, USA
sxl6435@ rit.edu

***Abstract*— Software security is a tough reality that affects the many facets of our modern, digital world. The pressure to produce secure software is felt particularly strongly by software engineers. Today's software engineering students will need to deal with software security in their profession. However, these students will also not be security experts, rather, they need to balance security concerns with the myriad of other draws of their attention, such as reliability, performance, and delivering the product on-time and on-budget. At the Department of Software Engineering at the Rochester Institute of Technology, we developed a course called Engineering Secure Software, designed for applying security principles to each stage of the software development lifecycle. As a part of this course, we developed a component called Vulnerability of the Day, which is a set of selected example software vulnerabilities. We selected these vulnerabilities to be simple, demonstrable, and relevant so that the vulnerability could be demonstrated in the first 10 minutes of each class session. For each vulnerability demonstration, we provide historical examples, realistic scenarios, and mitigations. With student reaction being overwhelmingly positive, we have created an open source project for our Vulnerabilities of the Day, and have defined guiding principles for developing and contributing effective examples.**

***Index Terms*—security, design, vulnerability, Common Weakness Enumeration, historicla**

I. Introduction

Software security is a tough reality that affects the many facets of our modern, digital world. The software that we depend upon must be secure, or we are at risk not only as customers, but as citizens.

The pressure to produce secure software is felt particularly strongly by today's software engineers. Even without regarding security, software engineers face a heavy load in their everyday jobs: understanding customer requirements, collaborating in large teams, learning new technologies, fixing bugs, and delivering new features on time. All of these activities involve a mindset of "building" software, yet security is about "breaking" software. To a customer, software is supposed to transparently improve their lives. To a malicious hacker, software is an opportunity to abuse functionality for malicious gain. As a result, software engineers must maintain both the "builder" mindset as well as the "breaker" mindset throughout the software development lifecycle.

Educating future software engineers the about the "breaker" mindset is no simple task. Undergraduate computing curricula are often steered toward teaching students to build software, not necessarily break software. Typically, educating software engineers begins with training them in programming (even "Hello, World!" on the first day of class is builder-oriented). Topics like performance, quality, and security are often pushed to the more advanced classes, long after students have developed the builder mindset.

Historically, many software security courses cover a variety of highly technical topics that require a specialized background, such as cryptography and networking [1], [2]. Or, security courses emphasize attack techniques over fortification techniques [2]. But software engineering students will not be security experts, they will be developing and maintaining applications with security as a non-functional requirement. Software engineering students must understand how the fundamentals of security apply to their software, but from the standpoint of developing secure software, not security software. To a software engineer, developing secure software is about preventing and mitigating *vulnerabilities*, or faults that "violate an [implicit or explicit] security policy" [3]–[5].

The objective of this work is to educate future software engineers about software security by providing simple, demonstrable, and relevant vulnerability examples. At the Department of Software Engineering at the Rochester Institute of Technology, we developed a course designed for applying security principles to each stage of the software development lifecycle. We developed a series of 19 vulnerability examples, called "Vulnerability of the Day" (VotD) that are designed to be demonstrated in 10 minutes at the beginning of each class period. A single VotD contains a plain-English description of the problem, the mitigation, a brief executable code sample set in a hypothetical realistic scenario, and links to real-world historical examples. From these examples, we have formed an open source project for other instructors to use these VotD and to contribute new examples.

II. Engineering Secure Software

The first author developed and taught a course titled *Engineering Secure Software* (ESS) in the Department of Software Engineering (SE) at the Rochester Institute of Technology. This 300-level course will be required for all SE

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Software Engineering in Education

undergraduates in the coming years. The one prerequisite for the course is our 200-level *Introduction to Software Engineering*, which in turn pre-requires some introductory programming courses.

The goal of ESS is to introduce software security as it applies to the discipline of SE. This emphasis differs from traditional security courses in that the intent of ESS is to educate future software engineers, not future security experts. This change in focus means that the entire course is structured and balanced around the software development lifecycle: requirements, design, implementation, test, and deployment.

The course covers a variety of software security principles and practices, such as: misuse and abuse cases, threat modeling, domain analysis, risk assessment, security design patterns, static analysis tools, defensive coding, deployment of cryptographic libraries, and insider threat.

Students also work together in teams on a term-long case study of an open source product. Case studies last term included large projects such as Android, Firefox, Ruby on Rails, and Chrome. Students had to examine the vulnerability history of the project, assess the architecture, enumerate and prioritize the threats, and conduct code inspections of the product.

The class was conducted in a studio laboratory format with two two-hour sessions each week. Each session was structured as follows: the first 10 minutes was VotD (see Section 3), then approximately 30 minutes for a lecture, then approximately 60 minutes for a class activity.

III. Vulnerability of The Day

One critical part of the SE discipline is understanding software design at multiple levels of abstraction. Security follows this abstraction principle in that security affects a product from the architecture level down to the coding level. For the portion of the class that covers coding level vulnerabilities, we developed VotD.

A. Vulnerabilities to a Software Engineer

A vulnerability is "an instance of a fault that violates an [implicit or explicit] security policy" [3], [5], [4]. Informally, a vulnerability is a special type of software bug that involves a user abusing functionality for malicious gains. Vulnerabilities can be small coding mistakes, or large design issues.

Today, numerous types of vulnerability exist. The Common Weakness Enumeration (CWE[1]) is an open taxonomy of vulnerability types that has hundreds of entries. These vulnerability types range from broad categories (e.g. Encryption Issues) to technology-specific issues (e.g. Explicit call to finalize() in Java).

For us, VotD was a way to incorporate a broad assortment of vulnerability types into a class of diverse principles. Most of the vulnerabilities we wished to demonstrate were small, and self-contained, had little to do with each other, yet demonstrated major security principles like Least Privilege and Defense in Depth. Initially, we developed three 30-minute lectures where all of these vulnerabilities would be demonstrated, but we found that trudging through so many assorted vulnerability types in a few class days would be tedious and difficult to follow. Furthermore, other class days had more of abstract view of security, by examining large designs and principles (e.g. threats and architectures). Thus, to make our class periods a balance between principles and practice, we developed VotD.

A single VotD contains the following elements: a description of the vulnerability, the mitigation, a brief executable code sample set in a hypothetical, realistic scenario, and links to real-world historical examples. An example VotD can be found in Section 3.3.

B. Guiding Principles

We initially developed these VotD examples while teaching the ESS course, so that the feedback from one VotD could be used in developing the next VotD. As we progressed through the quarter, we began codifying the principles of successful VotDs. The three guiding principles we developed were: simple, demonstrable, and relevant.

Simple. The instructor must be able to describe the VotD in only a few minutes, using a brief coding example. We have found that the best size for a code example is one that can fit on a standard projector screen readable from the back of the classroom without scrolling. For us, that limits the essence of the code sample to be 90 characters wide and 48 lines long. For all of our VotDs, we use these metrics as guidelines to simplify the example down to its most essential pieces.

Furthermore, an essential part of keeping each VotD simple, we employed a "plain English" philosophy to writing about each VotD. Examining large vulnerability taxonomies like the Common Weakness Enumeration (CWE), provides a considerably large amount of information where the essence of the vulnerability can get lost in the technical minutia. Thus, we describe each vulnerability in as concise and simple terms as possible, then link to more detailed descriptions like those in the CWE.

Demonstrable. We have found that demonstrability is critical to VotD because **s**tudents become more convinced that a vulnerability is a real threat when it can be demonstrated. In our experience, simply talking about possibilities and potentials is often not enough to convince the students that a simple coding mistake is a real threat. Thus, every VotD is must be a demonstrable coding example.

We also emphasized demonstrating the vulnerability over demonstrating complex exploitation of the vulnerability. For example, our SQL injection VotD shows how one can execute arbitrary SQL on the database, but the example does not give a crafted SQL statement that drops tables or tampers with data. We believe that, due to having SE students as the audience, VotD ought to be defense-focused, not offense-focused. Furthermore, the effort of turning a vulnerability into a true exploit adds complexity to the example that can obscure the root problem. Our philosophy is to demonstrate that the vulnerability exists, then show how to mitigate it.

[1] http://cwe.mitre.org

Relevant. A VotD must be a real threat to modern coding practices. This criterion steers us away from the "what if?" kinds of vulnerabilities to the "what has been happening?" vulnerabilities.

One way we maintain relevance is by framing the code example in a hypothetical scenario. For example, for Integer Overflow (see Section C), our example is a healthcare system that tracks patients, and the vulnerability is a HIPAA violation. An unexpected arithmetic mistake then becomes a vulnerability in the domain-specific situation. By tying the threat of the vulnerability to one domain, students can begin to see how the vulnerability can arise in other domains.

Another way we keep each VotD relevant is by finding historical, disclosed vulnerabilities. The Common Vulnerabilities and Exposures (CVE) database is a large collection of vulnerabilities in specific products. In many of these vulnerabilities, we have traced the CVE identifier for a product to their bug tracking system so that students can view the fix. By seeing the fix and the discussion around the fix, students can see how relevant a vulnerability is immediately.

In looking for relevant VotDs, we found that several examples we initially developed had been largely mitigated in recent years. For example, HTTP Response Splitting (CWE-113) used to be a major concern for application developers, but it has been largely solved by improved web application servers. Thus, in striving for relevance we avoided teaching vulnerabilities considered to be mostly obsolete.

Finally, getting a relevant VotD also implies that we need to choose a set of examples that are mutually exclusive. For example, Buffer Overflow (CWE-119) has a multitude of variants, all with slightly different situations but with similar mitigations. To keep each VotD relevant, if we teach one type of Buffer Overflow, then similar vulnerabilities become less relevant. Striving for relevance keeps our selections of VotDs diverse.

We found that developing a relevant VotD is the most challenging part of developing these examples. A VotD can be simple and demonstrable, but can be highly situational and not relevant to most applications.

C. Example: Integer Overflow

The Integer Overflow vulnerability is an example of an arithmetic calculation gone wrong. In languages such as C and Java, integer calculations are computed without attempting to check against the boundaries of the integer type. Thus, when two large numbers are added to each other, or when a large number is cast down to a smaller number, the system will silently change the number in risky ways.

Consider the Java example in Figure 2. The system is pulling a patient record by first looking up the unique identifier for the patient "Chuck Norris". But, the identifier for Chuck Norris is larger than Java's Integer.MAX_VALUE. So, when the number gets cast down, the wrong patient's records are retrieved. In the healthcare domain in the United States, this can lead to a violation of the Health Insurance Portability and Accountability Act of 1996, where a patient's records are unnecessarily exposed.

Historically, integer overflows have arisen in a variety of ways. For example, in the Firefox vulnerability shown in Figure 3, the parser made an integer casting error and the size of a memory buffer was computed to be negative number. In many C implementations, malloc() will not check for negative numbers, and return an empty buffer – allowing the user to overwrite arbitrary memory locations. This particular historical example led to arbitrary code execution of user's browsers.

Mitigations for integer overflows can depend on the situation. The mitigation for our example is simply not casting to an int and always using a long. But, in other situations (such as adding large numbers for, say, bank accounts), you might want to check that the number has not overflowed. In situations where you are doing many calculations, consider using special overflow-resistant libraries, like Java's BigInteger and

```
public class GetPatient {
        public static void main(String[] args) {
                int patientID = (int) getPatient("Chuck Norris");
                //         oops ^^^^^
                System.out.println("Patient ID should be 4294967314L, but is: " + patientID);
        }
        private static long getPatient(String string) {
                return 4294967314L;
        }
}
```

Fig 1. Code example for Integer Overflow VotD

```
--- a/layout/xul/base/src/tree/src/nsTreeSelection.cpp
+++ a/layout/xul/base/src/tree/src/nsTreeSelection.cpp
@@ -698,17 +698,22 @@ NS_IMETHODIMP nsTreeSelection::SetCurren
-    nsTreeRange* macro_new_range = new nsTreeRange(macro_selection, (macro_start), (macro_end));
+    PRInt32 start = macro_start;
+    PRInt32 end = macro_end;
+    if (start > end) {
+      end = start;
+    }
+    nsTreeRange* macro_new_range = new nsTreeRange(macro_selection, start,end);
```

Fig 2. Historical fix of an Integer Overflow from Firefox (Bugzilla #571106)

BigDecimal.

IV. The VotD Project

In the interest of seeing this project be used more widely, we have released this project under the Educational Community License 2.0 license. For each of our 19 VotD examples, we have compiled the code example, Makefiles, instructor notes, and an HTML description. Our description includes links to historical examples, the CWE, and other sources of information. The project is posted on GitHub[2] and we welcome usage, feedback, and contributions.

Since educators may seek to use only a few vulnerabilities from the VotD project, we have a script to build a smaller subset of VotDs from the original project. This script builds the website for posting on the web, along with a zip file of the VotDs for the students to download and try out.

In the future, we envision the VotD project containing a much bigger database of vulnerability examples. Currently, our examples span a wide variety of languages and technologies, including Java, PHP, Ruby, C, and configuration files. Our goal is to present vulnerabilities as language-independent, so we welcome coding existing VotDs in other languages (where appropriate). We also currently maintain a list of platforms that we have tested our examples on, so compatibility testing is also needed.

V. Student Feedback

By way of their course evaluations and in-class discussions, the students made their opinions clear that VotD was the most effective part of the class. In a survey given at the end of the Spring 2012 class, students were asked what their favorite part of the course was. Of the 21 students, 100% responded and 16 said that VotD was their favorite part of the class. In particular, students mentioned that they appreciated the wide selection of vulnerabilities, the level of detail with which the code examples were provided, and the fact that these were simple, running examples.

VI. Related Work

To our knowledge, we do not know of any other education-focused, software engineering project for teaching undergraduate students about vulnerabilities. In terms of source material, the most relevant work to ours is the massive CWE taxonomy and recent vulnerability disclosures in the software industry.

Discussing how security ought to be incorporated into computing curricula is common today. Taylor and Azadegan [7] propose a way of introducing security into a wide variety of courses throughout a computer science and information technology curriculum. Rowe, et al. [8] discuss applying security to an information technology curriculum based on the security principle "Prepare, Defend, Act". Rubin and Misra [9] discuss the challenges of introducing a security curriculum at the graduate level that emphasizes software engineering. Finally, Firesmith [10] discusses the teaching of requirements engineering for topics likes safety and security.

VII. Summary and Future Work

The objective of this work is to educate future software engineers about software security by providing simple, demonstrable, and relevant vulnerability examples. We developed a series of vulnerability examples, called Vulnerability of the Day, that are designed to be demonstrated in 10 minutes at the beginning of class. These VotD examples contain brief descriptions as well as links to historical patches to demonstrate relevance. From our current set, we have started an open source project under the Educational Community License v2.0 so that other instructors may use these examples. We welcome anyone to contribute, provide feedback, test, or use our examples.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0837656. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Jones and T. J. Stallings, "Network security in two-year colleges," *J. Comput. Sci. Coll.*, vol. 25, no. 5, pp. 83–88, May 2010.

[2] V. Pothamsetty, "Where security education is lacking," in *Information security curriculum development*, New York, NY, USA, 2005, pp. 54–58.

[3] J. Allen, S. Barnum, R. Ellison, G. McGraw, and N. Mead, *Software Security Engineering*, 1st ed. Addison-Wesley Professional.

[4] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[5] I. V. Krsual, "Software Vulnerability Analysis." PhD Dissertation, Purdue University, 1998.

[6] "Agile Manifesto." [Online]. Available: http://agilemanifesto.org. [Accessed: 01-Nov-2012].

[7] B. Taylor and S. Azadegan, "Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum," in *Proceedings of the 3rd annual conference on Information security curriculum development*, New York, NY, USA, 2006, pp. 24–29.

[8] D. C. Rowe, B. M. Lunt, and J. J. Ekstrom, "The role of cyber-security in information technology education," in *2011 conference on Information technology education*, New York, NY, USA, 2011, pp. 113–122.

[9] B. S. Rubin and B. S. Misra, "Creating a Computer Security Curriculum in a Software Engineering Program," in *29th international conference on Software Engineering*, Washington, DC, USA, 2007, pp. 732–735.

[10] D. G. Firesmith, "Engineering safety- and security-related requirements for software-intensive systems: tutorial summary," in *International Conference on Software Engineering - Volume 2*, New York, NY, USA, 2010, pp. 489–490.

[2] https://github.com/votd/vulnerability-of-the-day