Structuring a Comprehensive Software Security Course Around the OWASP Application Security Verification Standard

Sarah E. Elder
Department of Computer Science)
North Carolina State University
Raleigh, USA
seelder@ncsu.edu

Nusrat Zahan
Department of Computer Science)
North Carolina State University
Raleigh, USA
nzahan@ncsu.edu

Val Kozarev
Department of Computer Science
North Carolina State University

Rui Shu
Department of Computer Science
North Carolina State University
Raleigh, USA
rshu@ncsu.edu

Tim Menzies
Department of Computer Science
North Carolina State University
Raleigh, USA
timm@ieee.org

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, USA
laurie_williams@ncsu.edu

***Abstract*—Lack of security expertise among software practitioners is a problem with many implications. First, there is a deficit of security professionals to meet current needs. Additionally, even practitioners who do not plan to work in security may benefit from increased understanding of security. *The goal of this paper is to aid software engineering educators in designing a comprehensive software security course by sharing an experience running a software security course for the eleventh time.* Through all the eleven years of running the software security course, the course objectives have been comprehensive – ranging from security testing, to secure design and coding, to security requirements to security risk management. For the first time in this eleventh year, a theme of the course assignments was to map vulnerability discovery to the security controls of the Open Web Application Security Project (OWASP) Application Security Verification Standard (ASVS). Based upon student performance on a final exploratory penetration testing project, this mapping may have increased students' depth of understanding of a wider range of security topics. The students efficiently detected 191 unique and verified vulnerabilities of 28 different Common Weakness Enumeration (CWE) types during a three-hour period in the OpenMRS project, an electronic health record application in active use.**

***Index Terms*—Security and Protection, Computer and Information Science Education, Industry-Standards**

I. Introduction

Throughout the world, demand exceeds supply for trained cybersecurity professionals. In the United States, the cybersecurity workforce has been deemed, in a Presidential Executive Order [1], a "strategic asset that protects the American people, the homeland, and the American way of life." The Cyberseek[1] project, sponsored by the US National Institute of Standards and Technology (NIST), highlights a dangerous shortage of cybersecurity workers that puts the country's digital privacy and infrastructure at risk. To help address this problem, NIST has created the National Initiative for Cybersecurity Education (NICE) Cybersecurity Workforce Framework [2]. The mission of the NICE Framework is to "energize and promote a robust network and an ecosystem of cybersecurity education, training, and workforce development." The NICE Framework establishes a taxonomy and common lexicon for describing the cybersecurity workforce including tasks, skills, training, and personnel; as well as a matrix of specialty areas for K-12 through university degree programs. Similarly, the United Kingdom National Cyber Security Programme supported the development of the Cybersecurity Body of Knowledge (CyBoK)[2]. The CyBoK codifies literature, such as textbooks, academic research articles, technical reports, white papers and standards on 19 essential knowledge areas of cybersecurity.

Higher education plays a key role in filling the gaps in the cybersecurity workforce. In 2017, the two major computer science professional societies, the Association for Computing Machinery (ACM) and the IEEE Computer Society (IEEE-CS), published Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity (CSEC2017 v1.0) [3]. A key aspect of cybersecurity education is the area of *software security* [4] which is considered to be the intersection of software engineering and cybersecurity in which software engineers "build security in" to products.

The goal of this paper is to aid software engineering educators in designing a comprehensive software security course by sharing an experience running a software security

This material is based upon work supported by the National Science Foundation under Grant No. 1909516. 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.

[1]https://www.cyberseek.org

[2]https://www.cybok.org/

course for the eleventh time. When this course was first offered in 2009, few courses on software security existed. Through all the eleven years of running the software security course, the course objectives have been comprehensive – ranging from security testing, to secure design and coding, to security requirements to security risk management.

The course has always been designed to be experiential, whereby students applied classroom topics on a system. Due to the rapid evolutionary nature of cybersecurity, each year the course was updated to include new tools, new techniques, new standards, new regulations, and new research results. In Spring 2020, a theme of the course assignments was to map vulnerability discovery to the security requirements of the Open Web Application Security Project (OWASP) OWASP Application Security Verification Standard (ASVS). *Of all the changes made to the course offering over the eleven years, the change toward structuring the course around the ASVS seemed to be the most beneficial from a student learning perspective*, and we share our experiences in this paper.

With this paper, we make the following contributions:

- A software security course structure (topics and assignments) that has matured over an eleven year period; and
- Our experience structuring a software security course around the OWASP Application Security Verification Standard (ASVS).

The rest of this paper is organized as follows. In Section II and Section III, we present related work and key concepts, respectively. In Section IV we provide background on the ASVS standard which was the main innovative aspect added to the class. We lay out the course topics in Section V and course assignments in Section VI. We present the results of the final exploratory penetration testing exercise in Section VII. Finally, we present our lessons learned in Section VIII.

II. Related Work

Previous work has been done on tools and techniques for security education. For example, work has been done in the use of serious games and the gamification of security research, particularly for the purpose of education including two summits on *Gaming, Games, and Gamification in Security Education* (3GSE)[3] and more recent works by Antonaci et al and Švábenskỳ et al [5], [6]. Other work has looked at the use of Massive Open Online Courses (MOOCs) to teach cybersecurity [5], [7], [8]. Many organizations provide online cybersecurity learning resources such as the Cyber Security Body of Knowledge (CYBOK)[4] and SAFECode[5]. Several researchers, teachers, and organizations provide lists of knowledge objectives or course outcomes such as work by Mead et al [9] and the ACM/IEEE/AIS SIGSEC/IFIP Cybersecurity Curricula Guideline [10] which map knowledge objectives and learning outcomes to Bloom's Taxonomy.

Two studies [11], [12] from the University of Maryland (UMD) are particularly useful in understanding our own experiences in Software Security education. The studies from UMD examine how students perform in a *Build It, Break It, Fix It* competition where teams of students attempt to build the most secure software (the "Build It" phase), find the most vulnerabilities in each other's software (the "Break It" phase), and fix the vulnerabilities in their own software that have been found (the "Fix It" phase). The studies found that the reason most teams implemented security vulnerabilities was lack of knowledge. However, the impact of education on the vulnerabilities introduced by students was mixed. The first study, published in 2016 by Ruef et al. [11], found that students who had taken an online course on cybersecurity performed better than students who had not taken the course. The second study found that while knowledge, or lack thereof, appeared to be the main reason vulnerabilities were introduced in the "Build It" phase of the competition, participation in the online course on cybersecurity did not seem to influence whether students introduced vulnerabilities. The authors suggest that experience with a variety of different types of vulnerabilities may help in reducing the number and types of vulnerabilities introduced (in the "Build it" phase), as well as increasing the number and types of vulnerabilities identified when looking for vulnerabilities in software (in the "Break it" phase). The results of these studies indicate the need for more strategies for security education. Using a comprehensive standard, such as ASVS, may be a helpful strategy, since students who utilize the ASVS framework should be more familiar with a wide variety of vulnerability types.

Other studies have examined the role of knowledge in software verification and validation. Votipka et al's study on *Hackers vs Testers* [13] examines how white-hat hackers approach security testing as compared to software testing experts. The authors performed a series of interviews with white-hat hackers and with testers. Based on the interviews, the authors developed a model for the vulnerability discovery process. They found that although both hackers and testers follow similar processes for finding vulnerabilities, different experience and knowledge between the two groups led to different results. In a non-security-specific setting, Itkonen et al [14] found that domain, system, and general software engineering knowledge were all useful in performing exploratory testing; and the ability to apply that knowledge "on the fly" may contribute to exploratory testing being more efficient than systematic, scripted testing.

This paper adds to existing work by providing our experience and lessons learned when using a standard, such as ASVS, to structure assignments and lectures in a graduate-level software security course at a public research university. As can be seen in Section V, ASVS was incorporated into existing course objectives and topics. The course website[6] references many additional materials that were also used to help students.

[3]https://www.usenix.org/conferences/byname/885

[4]https://www.cybok.org/

[5]https://safecode.org/training/

[6]https://sites.google.com/a/ncsu.edu/csc515-software-security/Schedule-of-Subjects/Prelim

III. Key Concepts

In this section, we define terminology needed to understand the rest of this paper.

A. Vulnerability

In the course, students were introduced to several definitions of the term *security vulnerability*. While we do not think that the differences between the definitions would impact the student results, we provide these definitions for the reader. Two informal definitions used at the beginning of the course were that a security vulnerability is "a bug with security consequences" or "a design flaw or poor coding that may allow an attacker to exploit software for a malicious purpose". More formally, students were given the definition of a vulnerability from the U.S. National Institute of Standards and Technology (NIST) Security and Privacy Controls for Federal Information Systems and Organizations Special Publication (SP) 800-53 [15]. NIST SP 800-53 defines a vulnerability as "a weakness in ... an implementation that could be exploited by a threat source". The vulnerability counts in Section VII-D abide by the definition of vulnerability from the Common Vulnerabilities and Exposures (CVE)[7] list that is also used by the U.S. National vulnerability Database[8]. Namely, that a vulnerability is a "flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components"[9].

B. Common Weakness Enumeration (CWE)

Per the CWE website, "CWE is a community-developed list of software and hardware weakness types."[10]. Many security tools, such as the OWASP Application Security Verification Standard (ASVS) and most vulnerability detection tools, use CWEs to identify the types of vulnerabilities relevant to a given security requirement, test case, or tool alert. Each CWE type header describes the vulnerability, and each type has a unique numeric value which we will refer to as the CWE number. We use the vulnerability types in the CWE list to better understand and describe the diversity of vulnerabilities students found during their exploratory testing exercise in section VII

IV. OWASP Application Security Verification Standard (ASVS)

The Open Web Application Security Project (OWASP)[11] is a non-profit organzation that promotes software security through educational resources, open-source software, and other open-source projects.

The OWASP Application Security Verification Standard (ASVS) is an open standard for performing Web application security verification. The ASVS provides a high-level set of "requirements or tests that can be used by architects, developers, testers, security professionals, tool vendors, and consumers to define, build, test and verify secure applications" [16]. The ASVS Project[12] intends for the standard to be "commercially workable", and to provide adequate coverage so that ASVS can be used for a wide range of purposes including as an internal metric for security, guidance for developers when implementing new security features, or as a security baseline when evaluating third-party software or development contracts. The current version of OWASP ASVS is Version 4.0.1 released in March 2019[13].

ASVS requirements are phrased in terms of what would need to be verified. The requirements must be adapted to a specific system under test to be effective. The requirements are grouped into higher-level sections and sub-sections. Additionally, each ASVS requirement is mapped to a CWE type using the CWE number. Where applicable, ASVS requirements are also mapped to applicable standards from the U.S. National Institute of Standards and Technology (NIST). An example ASVS requirement, ASVS 2.2.1, is shown in Figure 1. ASVS 2.2.1 is part of ASVS section *V2: Authentication Verification Requirements*, subsection *V2.2 General Authenticator Requirements*. ASVS requirements from section V2 are mapped to *NIST Special Publication (SP) 800-63: Digital Identity Guidelines*. A violation of ASVS 2.2.1 would result in a vulnerability of the type *CWE-307: Improper Restriction of Excessive Authentication Attempts*. Additionally, ASVS 2.2.1 maps to sections 5.2.2, 5.1.1.2, 5.1.4.2, and 5.1.5.2 of NIST SP 800-63.

The OWASP ASVS has three levels of requirements. If a requirement falls within a level, it also falls within higher levels. For example a requirement that is part of Level 2 is also part of Level 3 of the standard. ASVS describes Level 1 as "the bare minimum that any application should strive for" [16]. As of Version 4 of ASVS, Level 1 requirements can all be verified manually. Level 2 should be addressed by any applications handling sensitive information and is appropriate for most applications. Addressing Level 3 requirements may require more resources than organizations are willing or able to spend on some systems. However, Level 3 requirements should be met for critical systems. In the example in Figure 1, ASVS 2.2.1 is an L1 requirement, which means it must also be verified to meet L2 and L3 requirements.

V. Course Topics

The graduate-level Software Security course introduces students to the discipline of designing, developing, and testing secure and dependable software-based systems. The course website and resources can be accessed at https://sites.google.com/a/ncsu.edu/csc515-software-security/

The four course learning objectives and topics taught to achieve these objectives are as follows:

1) **Security testing.** *Objective 1: Students will be able to perform all types of security testing.*

[7]https://cve.mitre.org
[8]https://nvd.nist.gov/vuln
[9]https://cve.mitre.org/about/terminology.html
[10]https://cwe.mitre.org/
[11]https://owasp.org/
[12]https://owasp.org/www-project-application-security-verification-standard/
[13]https://github.com/OWASP/ASVS/tree/v4.0.1

Fig. 1. ASVS requirement 2.2.1

#	Description	L1	L2	L3	CWE	NIST§
2.2.1	Verify that anti-automation controls are effective at mitigating breached credential testing, brute force, and account lockout attacks. Such controls include blocking the most common breached passwords, soft lockouts, rate limiting, CAPTCHA, ever increasing delays between attempts, IP address restrictions, or risk-based restrictions such as location, first login on a device, recent attempts to unlock the account, or similar. Verify that no more than 100 failed attempts per hour is possible on a single account.	✓	✓	✓	307	5.2.2/ 5.1.1.2/ 5.1.4.2/ 5.1.5.2

- **Penetration Testing (PT)**: Penetration testing is a method for "gaining assurance in the security of an IT system by attempting to breach some or all of that system's security, using the same tools and techniques as an adversary might." [17] In the class, students performed exploratory penetration testing in which the tester "spontaneously designs and executes tests based on the tester's existing relevant knowledge" [18]. The students also performed systematic penetration testing in which the tester methodically develops and documents a test plan in which the test cases comprehensively address the security objectives of the system. The test plan is then manually executed against the System Under Test (SUT) [19]–[22] in academic literature and in practice, penetration testing may refer to any dynamic security testing, automated or manual [23]–[25]. However, in practice testers may use a combination of exploratory and systematic testing [13]. Students also learned how to augment their testing with the use of a proxy server, OWASP ZAP[14], that allows the user to manipulate traffic outside the browser. Penetration testing can also be referred to as *pentesting* and *ethical hacking*.
- **Dynamic Application Security Testing (DAST)**: DAST tools automatically generate and run a test cases against the SUT, without any access to the SUT source code[15]. DAST can also be be referred to as *fuzzing*.
- **Static Application Security Testing (SAST)**: SAST tools automatically scan application source code for defects [23]–[25]. SAST can also be referred to as *static analysis*.
- **Interactive Application Security Testing (IAST)**. IAST tools perform dynamic analysis but also have access to source code. IAST tools require a tester to interact with the application while the IAST tool monitors what code is exercised during the interactions[16].

2) **Secure design and coding practices.** *Objective 2: Students will understand secure design and coding practices to prevent common vulnerabilities from being injected into software.*
   - **Design**. Approximately half of the vulnerabilities in a software product are due to design flaws [4] that can cause architectural-level security problems and need to be fixed via redesigning a portion of a product. The students learned about avoiding the "Top 10 Software Security Design Flaws" [26] according to the IEEE Center for Secure Design.
   - **Coding**. Approximately half of the vulnerabilities in a software system are due to implementation bugs that are caused by code-level security problems. Implementation bugs are usually fixed via altering some lines of code. The students learned about avoiding common implementation bugs such as OWASP Top 10 Web Application Security Risks[17] and CWE Top 25 Most Dangerous Security Weaknesses[18].
   - **Process**. Students learned about organizing their secure design and coding practices around a Secure Development Lifecycle such as the Microsoft Secure Development Lifecycle (SDLC)[19]. Students also learned about assessing the SDLC and/or the security practices used by an organization via the Building Security In Maturity Model (BSIMM)[20] and the OWASP Software Assurance Maturity Model (SAMM)[21].
3) **Security requirements**. *Objective 3: Students will be able to write security and privacy requirements.* Security and privacy requirements include compliance with necessary standards and regulations (such as GDPR[22] or HIPAA[23]). These security and privacy requirements must proactively include functionality to thwart attacker's attempts to exploit system functionality. In the

[14]https://www.zaproxy.org/
[15]https://insights.sei.cmu.edu/sei_blog/2018/07/10-types-of-application-security-testing-tools-when-and-how-to-use-them.html
[16]https://www.synopsys.com/glossary/what-is-iast.html
[17]https://owasp.org/www-project-top-ten/
[18]https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html
[19]https://www.microsoft.com/en-us/securityengineering/sdl
[20]https://www.bsimm.com/
[21]https://www.opensamm.org/
[22]https://gdpr-info.eu/
[23]https://www.hhs.gov/hipaa/index.html

class, students learned about formal security requirements, such as those that are specified in NIST 800-53 Security and Privacy Controls[24]. Additionally, students used the following four techniques to aid in their ability to "think like an attacker" and to specify this defensive security functionality.

- **Adversarial thinking.** Students were exposed to the tactics, techniques of procedures (TTP) of attackers through the use of the MITRE ATT&CK®[25], a globally-accessible knowledge base of real-world observations and analysis of the actions of adversaries.
- **Threat modeling.** Through the use of threat modelling [27], students considered, documented and discussed the security implications of design in the context of their planned operational environment and in a structured fashion. Threats could be enumerated using a systematic approach of considering each system component relative to the STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) Model [28]. Threat models could also be collaboratively developed via the Elevation of Privilege game[26].
- **Attack trees.** Students created conceptual diagrams of threats on systems and possible attacks to reach those threats [29].
- **Abuse cases.** Students created abuse cases [30], which describe the system's behavior when under attack by a malicious actor. To develop abuse cases, students enumerated the types of malicious actors who would be motivated to attack the system.

4) **Security risk management.** *Objective 4: Students will be able to assess the security risk of a system under development.* Students learned to assess security risk in and a standardized and an informal way.

- **Formal risk assessment.** Students learned to assess security risk using processes outlined in the NIST Cybersecurity Framework[27] and NIST 800-30 Guide for Conducting Risk Assessments[28].
- **Informal, collaborative risk assessment.** Students played Protection Poker [31], [32], an informal game for threat modeling, misuse case development, and relative security risk quantification.

VI. Course Assignments

In this section, we provide information on the four-part project the students completed. We first provide information on the system the students worked on, followed by the structure of the assignments.

[24]https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final

[25]https://attack.mitre.org/

[26]https://www.usenix.org/conference/3gse14/summit-program/presentation/shostack

[27]https://www.nist.gov/cyberframework

[28]https://www.nist.gov/privacy-framework/nist-sp-800-30

A. System Under Test (SUT)

The SUT is OpenMRS, an open-source medical records system designed to be flexible so it could be adapted to many contexts where other medical records systems were not available[29]. OpenMRS has over 900,000 lines of code. OpenMRS, as described by the developer's manual is a "Java-based web application capable of running on laptops in small clinics or large servers for nation-wide use"[30]. OpenMRS uses both java and javascript. The OpenMRS architecture is modular.

B. Equipment

All student tasks were performed on a virtual machine using the school's virtual computing lab[31]. The use of a virtual computing lab allowed researchers to create a system image including the SUT (OpenMRS) and necessary testing tools. Each student could then checkout their own copy of the SUT and tools with minimal installation effort on the part of the student. For client-server tools, the server was setup in a separate VCL instance in advance by members of the research team with assistance from the teaching staff. All students accessed the same server instance through different accounts. Student images were assigned 4 cores, 8G RAM, and 40G disk space. One server instance had 4 cores, 8G RAM, and 60G disk space. The other server instance had 8 cores, 16G RAM, and 80G disk space. These specifications were based on the minimum requirements needed for students to complete the tasks in their assignments. Full testing of the entire OpenMRS system was outside the scope of student assignments.

C. Assignment Structure

Through the semester, the students completed a four-part project in teams of 3-4 students. The four parts are as follows.

1) Project Part 1: This assignment had two sub-parts, both involving the ASVS.

- **Systematic Penetration Testing (SPT).** The students planned and executed 15 penetration test cases. Each test case had to map to a unique ASVS Level 1 or Level 2 control. Student reported whether the test cases passed or failed, and the CWE of the vulnerability that was being tested for. Most teams had failing test cases, indicating they found vulnerabilities in OpenMRS.
- **SAST.** Students ran two commercially-available SAST tools on OpenMRS: Fortify[32] and Synposys Coverity[33]. For each tool, students randomly chose 10 alerts produced by the tool and determined if the alert was a true or false positive. If the alert was a true positive, the student had to explain how to fix the vulnerability, map the vulnerability

[29]https://atlas.openmrs.org/

[30]http://devmanual.openmrs.org/en/Kick_off/solving_the_health_it_challenges_our_responsemd.html

[31]http://vcl.apache.org/

[32]https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[33]https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html

to the CWE, and map the vulnerability to an ASVS control.

2) Project Part 2: This assignment had two sub-parts, the first of which involved the ASVS.

- **DAST.** The students ran the OWASP ZAP DAST and a commercially-available DAST, Synopsys Defensics[34]. The students chose five true positive alerts produced by each tool. For each of these, they wrote a structured black box text to replicate the discovered vulnerability. Each of these test cases were mapped to the associated ASVS control and CWE.
- **Vulnerable Dependencies.** Modern software uses many third-party libraries and frameworks as dependencies. Known vulnerabilities in dependencies is a major security risk. Students ran five tools (OWASP-Dependency-Check[35], RedHat Victims[36], GitHub Security scanners[37], Sonatype DepShield[38], and Snyk[39]) to identify the vulnerable dependencies in OpenMRS. Students compared the output of the five tools.

3) Project Part 3: This assignment had four sub-parts, the first three of which involved the ASVS directly.

- **Logging.** Students wrote 10 black box test cases for ASVS Section *V7: Error Handling and Logging Verification Requirements* requirements for Levels 1 and 2.
- **IAST.** Students ran the Synopsys Seeker tool [40] using five failing black box test cases from their earlier work to seed the Seeker run. For five of the true positive vulnerabilities identified by the tool, the students had to write a black box test case to replicate each discovered vulnerability. Each of these black box test cases were mapped to the associated ASVS control and CWE.
- **Test coverage.** ASVS has 14 sections. Students computed their test coverage for each of these sections for all the test cases they had written during Parts 1-3 of the course. The students then wrote 5 more test cases to increase the coverage of ASVS controls that they did not have a test case for.
- **Vulnerability discovery comparison.** Students compared the effectiveness and efficiency of the four detection techniques they had used (systematic penetration testing, SAST, DAST, and IAST). They computed an efficiency measure based upon true positive vulnerabilities detected per hour. They also recorded all the CWEs detected by each tool. Students were asked to reflect upon their experience with these techniques, comparing their ability of each technique to efficiently and effectively detect a wide range of types of exploitable vulnerabilities.

[34]https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html

[35]https://owasp.org/www-project-dependency-check/

[36]https://github.com/victims/maven-security-versions

[37]https://github.com/features/security

[38]https://depshield.github.io/

[39]https://snyk.io/

[40]https://www.synopsys.com/software-integrity/security-testing/interactive-application-security-testing/demo.html

4) Project Part 4: This assignment had three sub-parts, the first two of which involved the ASVS directly.

- **Protection Poker.** The students wrote 5 new functional requirements for OpenMRS to add functionality that is not in the system yet. The played Protection Poker, using the OpenMRS database tables on these requirements and reflected how to reduce the security risk.
- **Vulnerability fix.** Students submitted a fix for a vulnerability that had been detected earlier in the class.
- **Exploratory Penetration Testing (EPT).** Students were assigned to individually spend three hours performing exploratory testing. Students produced a video recorded of their three-hour session, noted any vulnerabilities found, and created black box test cases (labeled with ASVS control) based on the vulnerabilities found to document their results in a replicable way.

VII. Exploratory Penetration Testing Exercise: A View into Student Learning

While not a formal experiment and without a baseline comparison to a prior semester, we consider the student's performance on the final exploratory penetration testing exercise to be indicative of the skills they had learned during the semester. We share the verbatim assignment and the student results in this section.

A. Verbatim Assignment

The text below is the exact text students were given on how to perform the final exploratory penetration testing exercise.

Each team member is to perform 3 hours of exploratory penetration testing on OpenMRS. This testing is to be done opportunistically, based upon your general knowledge of OpenMRS but without a test plan, as is done by professional penetration testers. DO NOT USE YOUR OLD BLACK BOX TESTS FROM PRIOR MODULES. Use a screen video/voice screen recorder to record your penetration testing actions. Speak aloud as you work to describe your actions, such as, "I see the input field for logging in. I'm going to see if 1=1 works for a password." or "I see a parameter in the URL, I'm going to see what happens if I change the URL." You should be speaking around once/minute to narrate what you are attempting. You don't have to do all 3 hours in one session, but you should have 3 hours of annotated video to document your penetration testing. There's lots of screen recorders available – if you know of a free one and can suggest it to your classmates, please post on Piazza.

Pause the recording every time you have a true positive vulnerability. Note how long you have been working so a log of your work and the time between vulnerability discovery is created (For example, Vulnerability #1 was found at 1 hour and 12 minutes, Vulnerability #2 was found at 1 hour and 30 minutes, etc.) If you work in multiple sessions, the elapsed time will pick up where you left off the prior session – like if you do one session for 1 hour 15 minutes, the second session begins at 1 hour 16 minutes. Take a screen shot and number

each true positive vulnerability. Record your actions such that this vulnerability could be replicated by someone else via a black box test case. Record the CWE for your true positive vulnerability. Record your work as in the following table. The reference info for video traceability is to aid a reviewer in watching you find the vulnerability. If you have one video, the "time" should aid in finding the appropriate part of the video. If you have multiple videos, please specify which video and what time on that video.

While students were graded on their video and results, only the results were studied further. We discuss the results in the next section.

B. Data Collection

The exploratory testing results were collected, alongside other data as part of a separate study on vulnerability detection techniques. Hence the data we discuss focuses on unique vulnerabilities rather than student responses. For the same reason, we did not retain the videos students created as they were not necessary and not easily anonymized. Student data was collected following North Carolina State University's Institutional Review Board Protocol 20569. The protocol was amended to verify that the EPT vulnerabilities could be discussed in this study. Sixty-three of seventy students allowed their data to be used for the study by signing an informed consent form. The data was collected during the Spring Semester of 2020, the first semester the course was structured around ASVS. As part of the original vulnerability detection technique comparison, three researchers who are authors on this paper reviewed the student results remove erroneous vulnerability reports and duplicate vulnerabilities.

C. Students' Previous Experience

At the beginning of the course, students were asked to fill out a survey about their experience relevant to the course. The four survey questions were as follows:

- Q1: How much time have you spent working at a professional software organization – including internships – in terms of the # of years and the # of months?
- Q2: On a scale from 1 (none) to 5 (fully), how much of the time has your work at a professional software organization involved cybersecurity?
- Q3: Which of the follow classes have you already completed?
- Q4: Which of the following classes are you currently taking?

Q1 was short answer. For Q2, students selected a single number between 1 and 5. For Q3, the students could select as many options as were appropriate from a list of five graduate-level security or privacy courses and one undergraduate-level security course offered at the institution. For Q4, the students selected from the two graduate-level security courses that were also offered in the Spring of 2020. These two courses had both been offered previously, and were also part of the list for Q3.

Fifty-nine of the sixty-three students who agreed to let their data be used for the study had responded to the survey. Four students' responses to Q1 provided a numeric value but did not specify whether the numeric value indicated years or months, rendering the value unusable. For example, if a student simply put "3", we did not know if the student had 3 years of industry experience or 3 months of industry experience. Consequently, industry experience from 55 participants was used to approximate the average industry experience of the student participants.

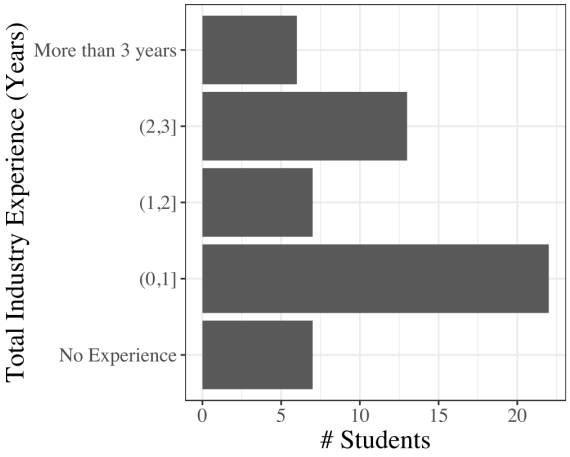

Fig. 2. Student Industry Experience (Q1)

Figure 2 shows the student's industry experience (Q1) in years. We use set notation where (indicates exclusive and] indicates inclusive. For example, in Figure 2, a student with exactly 2 years of experience would fall within the category $(1, 2]$ and not within the category $(2, 3]$. The median industry experience as indicated by answers to Q1 was 1 year. The average industry experience was 1 year 8 months. Of the 55 respondents whose answer for industry experience was clear, 7 had no industry experience at all.

Figure 3 shows that, among the 48 students who had industry experience, most students had some exposure to security. In other words, most students answered at least 2 to Q2. However, 20 of the 48 students who had any industry experience had no industry experience relevant to security. Only 10 students indicated a 3 or higher in answer to Q2. Students with more industry experience also had more security experience. Possible reasons for this include students having more time to gain security-related experience when they have more industry experience generally, and students with more industry experience in security being more likely to select a security-related course than students who intend to follow a career elsewhere within computer science.

Additionally, of the 59 students who responded to the survey, only 8 had previously taken a course in security or privacy (Q3). Nine students were currently taking a course in security or privacy in addition to the course from which the data was collected (Q4).

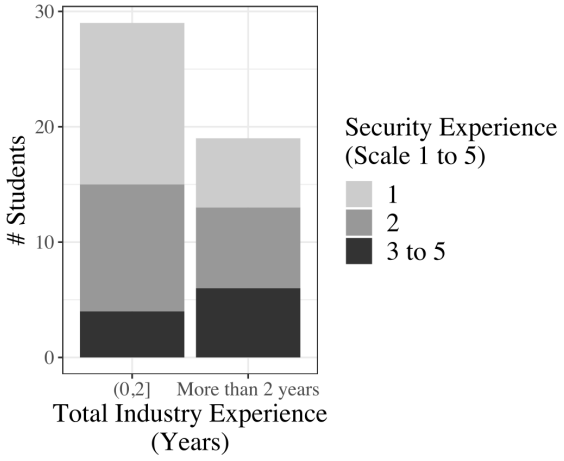

Fig. 3. Security-Related Industry Experience (Q2)

D. Student Results

Figure 4 shows student efficiencies as a boxplot. The data shown in Figure 4 has been trimmed [33], [34], using the median absolute deviation and median (MADN) to identify the most extreme outlier for each technique. The most extreme outlier for each technique was then removed from the dataset. Additionally, we use the abbreviations EPT and SPT to indicate exploratory penetration testing and systematic penetration testing respectively.

As shown in Figure 4, for the last assignment students were relatively efficient in using exploratory penetration testing to find vulnerabilities. Exploratory testing is highlighted in dark blue. The median, 2.47 vulnerabilities per hour in the case of exploratory testing, is indicated by the line in the middle of the boxplot. The average efficiency, 2.38 vulnerabilities per hour in the case of exploratory testing, is shown by the red x. Many factors clearly influenced student performance. Course information alone does not account for the difference since the vulnerability detection techniques used in Part 2 and Part 3 of the student project were less efficient than vulnerability detection techniques used in Part 1 as well as Part 4. However, the fact that the final exploratory testing assignment was the most efficient vulnerability detection assignment for the students is noteworthy. As discussed in Section II, knowledge is thought to be a key factor in exploratory testing. Furthermore, as described in Section VII-C, a majority of students had little or no security experience prior to this course. Nevertheless, the students were more efficient with exploratory testing than with tool-based techniques (SAST, DAST, and IAST), or with systematic penetration testing. The relatively high efficiency of the students when applying exploratory testing at the end of the course is promising.

In addition to high efficiency, students were able to find many different types of vulnerabilities during their exploratory testing exercise. Table I shows the unique vulnerabilities found by students using exploratory testing alone. Students recorded over 450 vulnerabilities. Since teams worked independently, many teams coincidentally found the same vulnerability. Additionally some reported vulnerabilities were not actually vulnerabilities, e.g. a student reported an information disclosure concern over the database URL being accessible. However, this access was by an admin, on a page designed to give admin users access to this sensitive information. In spite of the false positives and overlap, the student teams collectively found 191 unique vulnerabilities from 28 different CWE types. These 191 unique vulnerabilities are quantified in Table I.

TABLE I
VULNERABILITIES FOUND THROUGH EXPLORATORY PENETRATION TESTING

CWE	# Unique Vulns
16 - Configuration	2
20 - Improper Input Validation	13
79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	80
200 - Information Exposure	4
209 - Information Exposure Through an Error Message	10
269 - Improper Privilege Management	3
272 - Least Privilege Violation	1
280 - Improper Handling of Insufficient Permissions or Privileges	1
285 - Improper Authorization	23
308 - Use of Single-factor Authentication	1
319 - Cleartext Transmission of Sensitive Information	4
419 - Unprotected Primary Channel	3
434 - Unrestricted Upload of File with Dangerous Type	1
472 - External Control of Assumed-Immutable Web Parameter	1
509 - Replicating Malicious Code (Virus or Worm)	1
521 - Weak Password Requirements	7
532 - Information Exposure Through Log Files	3
544 - Missing Standardized Error Handling Mechanism	5
550 - Information Exposure Through Server Error Message	1
598 - Information Exposure Through Query Strings in GET Request	6
601 - URL Redirection to Untrusted Site ('Open Redirect')	2
602 - Client-Side Enforcement of Server-Side Security	1
613 - Insufficient Session Expiration	3
614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	1
620 - Unverified Password Change	1
639 - Authorization Bypass Through User-Controlled Key	5
770 - Allocation of Resources Without Limits or Throttling	2
778 - Insufficient Logging	11
Total	191

VIII. LESSONS LEARNED ABOUT THE USE OF ASVS

A. Improved Student Performance

Although we do not have a control group to formally quantify an improved performance on efficiency and effec-

Fig. 4. Exploratory Penetration Testing Efficiency[41]

tiveness in finding vulnerabilities, we were not expecting students to find as many unique vulnerabilities using exploratory penetration testing as they did, particularly in such a short timeframe. Student familiarity with the ASVS framework may have contributed their productivity.

B. Assisted in Identifying Knowledge Gaps

One advantage of ASVS when students erroneously identified or classified vulnerabilities was that the student references to ASVS could facilitate identifying knowledge gaps. For example, several ASVS requirements are about the entropy needed to secure authentication information and other secrets, such as ASVS 3.2.2 which states "Verify that session tokens possess at least 64 bits of entropy.". Many students wrote systematic test cases or logged exploratory tests vulnerabilities against this requirement. Although most of the tests and findings correctly identified the session token, most of the tests or findings incorrectly measured 64 bits of entropy. By referencing the original ASVS, we were sometimes able to better understand the student's intent. Entropy is outside the scope of the course, and would be anticipated as a gap in this case. Other gaps, such as a gap in what would be covered by a prerequisite course, might be of greater interest.

C. Motivated Self-Learning or Knowledge Sharing

In the projects Parts 1-3, students wrote each test case based on a single ASVS control, typically writing one test case per control. The course did not provide the technical knowledge for many of the ASVS controls. Additionally, the ASVS assumes that the reader either already has some knowledge of security concepts, or will obtain that knowledge elsewhere. As a result, students may have had to do some research in order to write a test case for a control, such as on entropy to continue the above example, or to share knowledge about security technology among team members.

D. Continued Need to Emphasize Repeatable Test Cases

In each project part, students were instructed to document their security test cases via "Detailed and repeatable (the same steps could be done by anyone who reads the instructions) instructions for how to execute the test case". This type of repeatable steps to replicate a vulnerability is necessary when reporting a discovered vulnerability to a product team. Since a Software Engineering course was a prerequisite to Software Security course, we assumed that students would be familiar with writing repeatable test cases. However, we found that students often did not specify their test cases to that level of detail. Since ASVS is high-level, to our knowledge ASVS alone cannot address this issue.

E. Awareness of Security Resources

Courses may instruct students on cybersecurity practices and technologies. However, government and practitioner organizations are creating valuable resources, such as the ASVS, and others mentioned in Section V. Students should learn to look for resources which can aid them in conducting their security work more systematically and based upon the input of security experts.

IX. Acknowledgments

We thank Jiaming Jiang for her support as teaching assistant for this class. We are grateful to the I/T staff at the university for their assistance in ensuring that we had sufficient computing power running course. We also thank the students in the software security class. Finally, we thank all the members of the Realsearch research group for their valuable feedback through this project.

This material is based upon work supported by the National Science Foundation under Grant No. 1909516. 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] D. Trump, "Executive Order on America's Cybersecurity Workforce," https://www.whitehouse.gov/presidential-actions/executive-order-americas-cybersecurity-workforce/, May 2, 2019, [Online; accessed 31-July-2019].

[2] National Institute of Standards and Technology (NIST), "National initiative for cybersecurity education (nice) cybersecurity workforce framework, nist special publication 800-181," https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-181.pdf, August 2017, [Online; accessed 31-Jan-2020].

[3] *Cybersecurity Curricula 2017: Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity*, Association for ComputingMachinery (ACM) and the IEEE Computer Society (IEEE-CS) Std., December 2017. [Online]. Available: https://cybered.hosting.acm.org/wp/wp-content/uploads/2018/02/csec2017_web.pdf

[4] G. McGraw, *Software Security*. Addison Wesley, 2006.

[5] A. Antonaci, R. Klemke, C. M. Stracke, M. Specht, M. Spatafora, and K. Stefanova, "Gamification to empower information security education," in *International GamiFIN Conference 2017*, 2017, pp. 32–38.

[6] V. Švábenskỳ, J. Vykopal, M. Cermak, and M. Laštovička, "Enhancing cybersecurity skills by creating serious games," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 2018, pp. 194–199.

[7] C. Theisen, T. Zhu, K. Oliver, and L. Williams, "Teaching secure software development through an online course." in *SecSE@ ESORICS*, 2017, pp. 19–33.

[8] S. Laato, A. Farooq, H. Tenhunen, T. Pitkamaki, A. Hakkala, and A. Airola, "Ai in cybersecurity education-a systematic literature review of studies on cybersecurity moocs," in *2020 IEEE 20th International Conference on Advanced Learning Technologies (ICALT)*. IEEE, 2020, pp. 6–10.

[9] N. R. Mead, J. H. Allen, M. Ardis, T. B. Hilburn, A. J. Kornecki, R. Linger, and J. McDonald, "Software assurance curriculum project volume 1: Master of software assurance reference curriculum," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2010.

[10] J. T. F. on Cybersecurity Education, "Cybersecurity curricula 2017: Curriculum guidelines for post-secondary degree programs in cybersecurity," New York, NY, USA, Tech. Rep., 2018.

[11] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build it, break it, fix it: Contesting secure development," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 690–703.

[12] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, "Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, to Appear.

[13] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 374–391.

[14] J. Itkonen, M. V. Mäntylä, and C. Lassenius, "The role of the tester's knowledge in exploratory software testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 707–724, 2013.

[15] *Security and Privacy Controls for Federal Information Systems and Organizations*, National Institute of Standards and Technology (NIST) Special Publication 800-53, Rev. 4, April 2013. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-53r4

[16] *Application Security Verification Standard*, OWASP Std., Rev. 4.0.1, March 2019.

[17] *Penetration Testing*, UK National Cyber Security Center Std., August 2017. [Online]. Available: https://www.ncsc.gov.uk/guidance/penetration-testing

[18] *Software and systems engineering — Software testing — Part 1: Concepts and definitions*, ISO/IEC/IEEE Std. 29 119-1, 09 2013.

[19] B. Smith and L. A. Williams, "Systematizing security test planning using functional requirements phrases," North Carolina State University. Dept. of Computer Science, Tech. Rep., 2011.

[20] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 97–106.

[21] B. Smith and L. Williams, "On the effective use of security test patterns," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 108–117.

[22] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Information and Software Technology*, vol. 55, no. 7, pp. 1279–1288, 2013.

[23] D. S. Cruzes, M. Felderer, T. D. Oyetoyan, M. Gander, and I. Pekaric, "How is security testing done in agile teams? a cross-case analysis of four software teams," in *International Conference on Agile Software Development*. Springer, Cham, 2017, pp. 201–216.

[24] R. Scandariato, J. Walden, and W. Joosen, "Static analysis versus penetration testing: A controlled experiment," in *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*. IEEE, 2013, pp. 451–460.

[25] M. Hafiz and M. Fang, "Game of detections: how are security vulnerabilities discovered in the wild?" *Empirical Software Engineering*, vol. 21, no. 5, pp. 1920–1959, 2016.

[26] IEEE Center for Secure Design, "Avoiding the top 10 software security design flaws," https://cybersecurity.ieee.org/blog/2015/11/13/avoiding-the-top-10-security-flaws, 2017. [Online]. Available: https://cybersecurity.ieee.org/blog/2015/11/13/avoiding-the-top-10-security-flaws

[27] K. Tuma, G. Calikli, and R. Scandariato, "Threat analysis of software systems: A systematic literature review," *Journal of Systems and Software*, vol. 144, pp. 275 – 294, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218301304

[28] M. Howard and D. E. Leblanc, *Writing Secure Code*. Microsoft Press, 2003.

[29] V. Saini, Q. Duan, and V. Paruchuri, "Threat modeling using attack trees," *J. Comput. Sci. Coll.*, vol. 23, no. 4, p. 124–131, Apr. 2008.

[30] P. Hope, G. McGraw, and A. I. Anton, "Misuse and abuse cases: getting past the positive," *IEEE Security Privacy*, vol. 2, no. 3, pp. 90–92, 2004.

[31] L. Williams, M. Gegick, and A. Meneely, "Protection poker: Structuring software security risk assessment and knowledge transfer," in *Massacci F., Redwine S.T., Zannone N. (eds) Engineering Secure Software and Systems (ESSoS) 2009*. Berlin, Heidelberg, Germany: Springer Lecture Notes in Computer Science, vol 5429, 2009.

[32] L. Williams, A. Meneely, and G. Shipley, "Protection poker: The new software security "game";," *IEEE Security Privacy*, vol. 8, no. 3, pp. 14–20, 2010.

[33] R. R. Wilcox and H. Keselman, "Modern robust data analysis methods: measures of central tendency." *Psychological methods*, vol. 8, no. 3, p. 254, 2003.

[34] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong, "Robust statistical methods for empirical software engineering," *Empirical Software Engineering*, vol. 22, no. 2, pp. 579–630, 2017.