ABSTRACT

METRO, MONICA GRACE. An Empirical Case Study to Compare the Effectiveness and Efficiency of Vulnerability Detection Techniques. (Under the direction of Dr. Laurie Williams.)

Multiple studies and surveys have highlighted the impact of security breaches on financial markets or an organization's reputation. As cyber-attacker activity rises and the threat of security breaches continues, practitioners could benefit from increasing detection practices to detect and remove vulnerabilities. There are numerous techniques available for vulnerability detection, each with their own advantages and disadvantages. However, practitioners have limited time and resources. Unfortunately, empirical results on the effectiveness and efficiency of vulnerability detection techniques that could help practitioners with technique selection are sparse.

The goal of this thesis is to aid software practitioners in selecting vulnerability detection techniques through empirical comparison of the effectiveness and efficiency of vulnerability detection techniques.

We conducted an empirical case study using an open-source electronic health record system to compare detection techniques: systematic manual penetration testing, automated penetration testing, and automated static analysis. The results show testing with a single technique is insufficient for detecting all vulnerability types as no technique detected every vulnerability type and the subsets of vulnerability types each technique discovered were largely orthogonal to each other. Automated penetration testing was the most efficient technique in terms of true positive vulnerabilities detected per hour. For maximum vulnerability type coverage, perform automated static analysis for detecting implementation bugs and systematic manual penetration testing for design flaws. Efficiency and effectiveness per vulnerability type varies.

An Empirical Case Study to Compare the Effectiveness and Efficiency of Vulnerability Detection Techniques

by
Monica Grace Metro

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Dr. Alexandros Kapravelos	Dr. William Enck

Dr. Laurie Williams
Chair of Advisory Committee

DEDICATION

To all of my friends and family that have supported me along the way.

BIOGRAPHY

Monica Metro was raised in Pennsylvania. She finished her high school education in North Carolina and went on to obtained her Bachelor of Science in Biochemistry from North Carolina State University in 2015. She started her Master's in Computer Science in 2018 in order to pursue her passion for software engineering.

ACKNOWLEDGEMENTS

I would like to thank Dr. Williams for granting me the opportunity to join her research group to learn about software security. Being able to learn under a researcher as distinguished as her has been such a rewarding experience. I would also like to thank Sarah and Val for all of their time and contributions. I don't know how I would have completed this thesis without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Recent studies, surveys, and reports documenting the impacts of security breaches on financial markets or an organization's reputation [9, 20, 37, 38] portray the importance of application security inspection and testing efforts. With cyber-attacker activity on the rise [20, 39], organizations are increasingly adopting security practices. However, response practices are being adopted at a higher rate than detection practices [42]. Practitioners could benefit from increasing detection efforts to detect and remove vulnerabilities. A software vulnerability is defined as an occurrence of a defect or weakness within software that causes unintended negative impacts to application data when exploited or triggered by a threat source. [14, 15, 29]

Organizations as well as individual developers have multiple methods at their disposal to test an application for vulnerabilities throughout the software development life cycle, each with advantages

and disadvantages. Whether practitioners choose to allocate time to test internally or hire external security professionals, one has a limited amount of time and resources to detect and remove vulnerabilities within an application. Unfortunately, empirical results on the effectiveness and efficiency of vulnerability detection techniques that could help practitioners with technique selection are sparse. More information is needed to guide resource-constrained practitioners on what techniques or tools to use to detect vulnerabilities to improve a practitioner's ability to find and remove an increased number of vulnerabilities in a cost-effective manner.

The goal of this thesis is to aid software practitioners in selecting vulnerability detection techniques through empirical comparison of the effectiveness and efficiency of vulnerability detection techniques.

In previous works [6–8, 36], case studies were conducted on three electronic health record (EHR) systems, Tolven Electronic Clinician Health Record (eCHR), OpenEMR, and PatientOS, to compare four vulnerability detection techniques: systematic penetration testing, exploratory manual penetration testing, automated penetration testing, and automated static analysis. Empirical evidence determined no single technique detected every type of vulnerability. The specific set of vulnerabilities identified by one tool was largely orthogonal to that of other tools. Static analysis detected both the most vulnerabilities per hour and the largest variety of implementation bugs. However, static analysis contained the highest false positive rate, making it less efficient. Automated penetration testing was the most productive in finding implementation bugs in terms of vulnerabilities per hour. Systematic manual penetration testing found the most design flaws.

In past years, techniques and tools have improved. We replicated this work to provide effectiveness and efficiency information on current vulnerability detection techniques using an open-source EHR subject similar to the previous subjects: Open Medical Record System (OpenMRS).

We state the following research questions:

RQ1 (Effectiveness): How does the effectiveness of vulnerability detection techniques compare in terms of the number and type of true positive vulnerabilities detected?

RQ2 (Efficiency): How does the efficiency of vulnerability detection techniques compare in terms of the number of true positive vulnerabilities detected per hour?

In summary, this research makes the following contributions:

- A comparison of the number and type of vulnerabilities detected with systematic penetration testing, automated penetration testing, and automated static analysis
- Empirical evidence indicating which vulnerability detection technique should be used to find both implementation bugs and design flaw vulnerabilities
- An evaluation of the efficiency of each vulnerability detection technique based on the metrics: vulnerabilities detected per hour and false positive rate.

1.1 Thesis organization

The rest of this paper is organized as follows:

- Section 2 explains important terminology, our detection techniques, and detected vulnerabilities discussed in our analysis and discussion.
- Section 3 presents related work relevant to vulnerability detection and the detection techniques.
- Section 4 describes the methodology of our empirical case study of vulnerability detection techniques.
- Section 5 provides our results.
- Section 6 contains our analysis and discussion of the results.
- Section 7 discusses limitations to our case study.

- Section 8 lists our conclusions.
- Section 9 suggests possibly future work.

CHAPTER

2

BACKGROUND

This section defines the terminology used throughout the paper, describes the vulnerability detection techniques being evaluated, and provides a reference for the types of vulnerabilities detected during our analysis.

2.1 Implementation Bugs and Design Flaws

Vulnerabilities can be classified as implementation bugs and design flaws.[25] Implementation bug vulnerabilities are software security defects that occur at the code level and are corrected by fixing a small number of lines of code. Design flaw vulnerabilities are software security defects that occur at the higher-level architecture phase of a software system's life cycle and can only be corrected with redesign. An example of implementation bugs vs. design flaws is unauthorized access

of data or a resource. Given a vulnerability where a resource (e.g. a file) is able to be accessed by an unauthorized actor because the code incorrectly checks the conditions that grant authorization, the vulnerability is an implementation bug. If the file is able to be accessed because the application does not implement access control at all, the vulnerability is a design flaw.[40] While software testing efforts generally focus on implementation bugs [25, 40], design flaws are considered to appear in equal proportions of security issues. [25]

2.2 Vulnerability Terminology

In this document, we use the term *potential vulnerabilities* to refer to the vulnerabilities reported by automated detection techniques. A potential vulnerability that is correctly labeled as a vulnerability is a true positive. Since no perfect technique exists, a potential vulnerability may be mislabeled. Mislabels are defined as false positives. Developers must manually examine each potential vulnerability and determine if it is a true or false positive. These manual reviews are time-consuming and reduce the productivity of developers. A vulnerability that is known to exist but was not detected is a false negative.

2.3 Vulnerability Detection Techniques

The following sections describe the vulnerability detection techniques utilized in this case study. The two main techniques are dynamic analysis and static analysis. Automated tools in both categories commonly report false positives and false negatives.[2, 3] False positives require extra resources but false negatives prevent vulnerabilities from being removed from an application. Consequently, the performance of tools are frequently quantified and compared.

2.3.1 Dynamic Analysis and Penetration Testing

Dynamic analysis is the detection of software vulnerabilities by executing test cases on a running application and analyzing the application's runtime behavior.[15] [29] Since the application needs to be running, the process includes installing and configuring the application and all required database servers. This run-time overhead can cause dynamic analysis to be slow and difficult to setup.

Penetration testing is a type of dynamic analysis where known malicious attacks or inputs are performed against an application in an attempt to find security vulnerabilities. The attacks are performed from an attacker's perspective which can be beneficial because access to the source code is not needed as is with static analysis. [3] [27] [15] Penetration testing is used to detect vulnerabilities before they can be exploited by a threat, allowing a mitigation or mitigation plan to be developed before the attack happens. This allows individuals and organizations to reduce or even eliminate the impact of a security breach. The two main types of penetration testing are automated penetration testing and manual penetration testing.

2.3.2 Automated Penetration Testing

Because vulnerabilities can be successful in more than one way but only need to be successful once to be exploitable, a large number of test cases need to be covered to ensure application security. Automated tools are often used to reduce time, labour, and costs (e.g. testing knowledge) required for conducting large numbers of test cases, increasing the amount of test cases that can be executed against the application.[34] Facilitating execution of test cases may increase the number of test cases executed against an application enabling an increased testing coverage [10] as repeatedly performing manual penetration tests for each vulnerability type would be extremely time consuming [3]. However, web application penetration testing techniques are known to be error prone when automated [34] as the tool can only examine the application's output and external behavior to determine if a vulnerability is present [2, 3]. If testing does not execute every code path within a

running application, vulnerabilities that exist in code may be missed.

Fuzzing, or fuzz testing, is an automated testing technique that can be implemented to perform automated penetration testing. Invalid, semi-random, or irregular data is sent as input to locations in the application that accept structured data (e.g. numbers, files, strings) across a trust boundary. [15] [11] [24] The application's behavior is then monitored for erroneous behavior, such as crashes or sensitive information disclosure that would indicate the presence of a defect. Fuzzers are the tools that implement automated fuzzing.[15] [24] Automated penetration tools can utilize fuzzers to generate and execute a large amount of test cases against an application. While fuzzers can speed up the testing process, the defects detected by the fuzzer have to be validated which can be time-consuming if done manually. [11]

Automated penetration tools can also implement a web proxy server to act as a man-in-the-middle between the web application and the browser. The ability to intercept, inspect, and modify requests and responses combined with a fuzzer or other defect-causing methods allows for an effective testing tool.

2.3.3 Manual Penetration Testing

Manual penetration testing is penetration testing without the aid of an automated tool. Non-automated tools may still be utilized, e.g. a web proxy server. With manual penetration testing, an auditor does not have to worry about undetected false negatives or the time required to review false positives, as is needed with automated tools. The primary disadvantage of manual penetration testing is the required resources needed to both manually define and execute attacks on the application such as time, expertise, and additional security requirements.[2] A tester with little penetration security expertise would need to take extensive time to educate themselves on the types of vulnerabilities to look for, as well as examples of attacks that would detect the vulnerability in the target application. An experienced tester has to take time to stay up-to-date with common vulnerabilities and attacks that expose those vulnerabilities as technology evolves. There are two

different types of manual penetration testing: systematic and exploratory.

Systematic Manual Penetration Testing follows a predefined test plan. For example, a collection test cases containing numbered procedures that can be executed. Although creating test cases requires time and penetration testing expertise, the test cases can be executed by a tester regardless of their penetration testing skill level or expertise. Manually inspecting software systematically offers the highest level of security assurance, but is typically time and cost extensive.[3]

Exploratory Manual Penetration Testing does not follow a predefined test plan. Instead, a tester "explores" the application with prior security and application knowledge and experience, testing the application opportunistically. The results of exploratory analysis are significantly affected by background knowledge [19], experience [19, 21, 41], and system knowledge [21]. Exploratory testing may be more effective and efficient than test-case-based systematic testing because of the way testers apply their experience and knowledge to test design and defect recognition [21] as well as by supporting tester creativity.[32].

2.3.4 Static Analysis

Static analysis detects software vulnerabilities in the internal structure of an application without execution. [15] Typically, the analysis is performed on source code, object code, or binaries. The reported potential vulnerabilities are behaviors that the analysis predicts to occur at run-time. Static analysis is thought to have a larger security coverage due to scanning all of the code, assuming the automated tool supports the programming language. An exception is in vulnerability types that typically require interaction with a running application to detect [2], such as server configuration vulnerabilities. It can also be cost effective as it does not include the run-time overhead of dynamic analysis. Since the application does not have to be running to perform static analysis, it can be used at an earlier stage in the software development life cycle. This, in addition to automated tools that facilitate repeatability, allow for an efficient testing workflow with metrics that can be evaluated between application releases.

However, static analysis has several disadvantages. Tools perform lexical analysis to look for a fixed set of common syntax and semantics patterns in the code. Patterns have to be added to the tool before the vulnerability can be detected. Code complexity that disrupts patterns can cause false negatives.[2] Historically, static analysis has also produced larger false positive rates than automated penetration testing.[2] The resources required to manually review reported vulnerabilities and the exhausting result of high false positives rates cause practitioners to ignore reported vulnerabilities or to avoid static analysis completely. To reduce the number of false positives created from static vulnerability detection, techniques have been developed to combine static and dynamic analysis by detecting vulnerabilities statically and verifying the vulnerabilities dynamically.[22]

2.4 Vulnerability Types

This section contains a reference for all of the vulnerability types detected by the techniques. The Common Weakness Enumeration (CWE)[1] provides common categorization, identifiers, and terminology for software security weaknesses, including web application vulnerabilities. CWE is owned by MITRE Corporation, a non-profit research and development organization that is federally sponsored by the U.S. Department of Homeland Security (DHS), the Cybersecurity Infrastructure Security Agency (CISA), as well as public-private partnerships for a variety of fields such healthcare, defense and intelligence, and cybersecurity. The detection tools used in this research often map a detected potential vulnerability to a CWE-defined unique identifier and vulnerability type. When available, these mappings are used to compare vulnerabilities between detection tools. For example, CWE-89 is "Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')" where 89 is the unique identifier, and SQL Injection is the vulnerability type.

CWE currently claims 808 total software weaknesses documented. The weaknesses are grouped into general concepts and then further into categories. Weaknesses can be linked by relationships,

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

such as child, parent, or peer. Closely related CWE vulnerabilities can therefore be combined into groups, discussed in more detail in our analysis and discussion in Section 6. During analysis, we grouped CWE-200, CWE-209, and CWE-210 into CWE-200 (information disclosure), CWE-284 and CWE-285 into CWE-284 (improper access control), and CWE-400 and CWE-404 into CWE-400 (uncontrolled resource consumption). The vulnerability types were grouped just enough to eliminate some bias between the specificity of the CWE mappings given by each detection tool. Types were not grouped into the most general structure in CWE possible.

Using improper access control as an example of CWE categorization, CWE-284 and -285 are both members of the Authorize Actors category, a member of the overarching Architectural concept. CWEs -276, -419, -639, -668, -827 are also all members of the Authorize Actors category. CWEs -284 and -285 were specifically grouped together without the others due to the detection tools defining -285 and -284 in the same way such that we could not differentiate between the two vulnerability types. The CWE page for improper access control admits that terms "access control" and "authorization" are often used interchangeably because the community cannot come to a consensus on the proper definitions of the terms, however, CWE also determines that authorization is typically more narrowly defined access control. As such, -284 is the CWE defined parent of -285.

Likewise, CWE-400 and -404 are both members of the Resource Management Errors category, a member of the Development concept. CWE-400 was determined to the the most general because it contains the definition of -404 in the weakness description.

CWE-200 is the parent of -209 which is the parent of -210. CWE-200 is also linked to other vulnerability types detected in this work, however, only -200, -209, and -210 were grouped because all refer to the same concept defined by CWE and refer to the same context in the application (information disclosure via displayed error pages with sensitive information). CWE-532 (sensitive information in log files) is also a child of CWE-200, but is not linked to -209 or -210 by CWE in addition to being a different context within the application (a non-error page information disclosure within the log).

Our work detected the following 32 vulnerability types, in numerical order by CWE identifier:

- **Configuration (CWE-16)**: a weakness introduced during the configuration of the software.
- **Improper input validation (CWE-20)**: an incorrect or lack of input validation can affect the data flow of the application.
- **Path Traversal (CWE-22)**: an attacker can manipulate a path allowing the ability to access, modify, or corrupt application files.
- **Cross-site scripting (CWE-79)**: the application does not neutralize or incorrectly neutralizes user-controllable input such that JavaScript code is executed or stored in the database.
- **SQL injection (CWE-89)**: input from users is directly used in constructing SQL queries, causing the database to be accessed in unintended ways.
- **Code injection (CWE-94)**: code is dynamically influenced by incorrectly validated external input from an upstream component allowing modification of code syntax or behavior.
- **External format string (CWE-134)**: external string inputs are not sufficiently validated, causing application crashes and abnormal session closures.
- **Information exposure (CWE-200,-209,-210)**: sensitive information is exposed such as user data or information about the system environment.
- **Parameter pollution (CWE-235)**: the application does not handle or incorrectly handles when the number of parameters, fields, or arguments with the same name exceeds the expected amount.
- **Call to an inherently dangerous function (CWE-242)**: the application calls a function that can never be guaranteed to work safely.

- **Incorrect default permissions (CWE-276)**: new user accounts do not start with minimal permissions such that the users receive access to features before being assigned.
- **Improper access control (CWE-284, -285)**: the application did not perform or incorrectly performed an authorization check allowing an unauthorized actor to access a resource or perform an action.
- **Cleartext sensitive data in a database (CWE-313)**: sensitive data is stored in the database.
- **Risky cryptographic hashing function (CWE-328)**: a weak hashing algorithm was detected that may be susceptible to collisions.
- **Use of insufficiently random values (CWE-330)**: random numbers or values used are insufficient for a security context.
- **Origin validation error (CWE-346)**: the event handler does not check the origin of the received message event.
- **Cross-site request forgery (CSRF) (CWE-352)**: an unauthorized request or action from cannot be distinguished as a legitimate request from the user.
- **Uncontrolled resource consumption (CWE-400, -404)**: the allocation and maintenance of a limited resource is not properly controlled such that a malicious actor can influence the amount of resources consumed, eventually leading to the exhaustion of available resources.
- **Unprotected primary channel (CWE-419)**: an administrative interface is not protected against unauthorized use.
- **Unsafe reflection (CWE-470)**: external input with the ability to create dynamic data is passed to a reflection API .
- **Parameter tampering (CWE-472)**: externally controlled inputs are not sufficiently verified.

- **Trust boundary violation (CWE-501)**: untrusted data flows to a data structure or context that is often assumed to be trustworthy.
- **Deserialization of untrusted data (CWE-502)**: untrusted JSON data is deserialized without sufficiently verifying that the resulting data will be valid.
- **Weak password requirements (CWE-521)**: strong passwords are not required making it easier for attackers to compromise user accounts.
- **Inclusion of sensitive information in log files (CWE-532)**: sensitive information is written to a log file.
- **Thread unsafe modification in singleton (CWE-543)**: the application uses a singleton pattern when creating a resource within a multi-threaded environment.
- **Unsynchronized access to shared data in a multithreaded context (CWE-567)**: Data is not properly synchronized outside of a locked context.
- **Open redirect (CWE-601)**: user-controlled input that specifies a link to an external site is accepted, and used in a redirect.
- **Authorization bypass through user-controlled key (CWE-639)**: sensitive data and services are not protected against directed object attacks.
- **Exposure of resource (CWE-668)**: a cross-origin window message can be sent without restricting the origin that can receive it, allowing unintended access to the resource.
- **Uncontrolled recursion (CWE-674)**: recursion is not controlled such that the loop will execute forever, or until system resources are exhausted.
- **Unrestricted document type definition (CWE-827)**: an untrusted document type definition (e.g. an XML document) can be used to specify an external URL.

CHAPTER

3

RELATED WORK

This section describes related research to the topics presented in this thesis, specifically, papers pertaining to healthcare IT and EHR security as well as vulnerability detection techniques.

3.1 Healthcare and EHRs

The subject of this thesis work, OpenMRS, is an open-source web-based EHR system. An EHR is an electronic version of a patients medical history and can also include administrative data such as demographics.[17] EHRs can improve the quality of healthcare by improving medical data accuracy and clarity as well as enabling providers and patients to make better decisions by making health information more accessible.

Cyber activity research has been an increasingly active in the healthcare domain in recent years

with growing public interest. [5] Historically, healthcare has been a vulnerable sector due to a lack of investments in cybersecurity. Organizations and governing bodies have employed regulations or incentives in efforts to boost healthcare security by establishing standards. For example, the United States requires EHR systems to meet the certification standards and regulations in the Office of the National Coordinator for Health Information Technology (ONC) Health IT Certification[1] in order to qualify for merit-based incentive payment systems. Federal policy like the U.S. Health Insurance Portability and Accountability Act (HIPAA)[2] omnibus rules may contribute to increased security by significantly reducing the frequency of privacy breaches.[43]

Security breaches have numerous impacts on healthcare entities. Financial losses can occur due to fines or costs associated with response and mitigation efforts. Patient trust can be lost as valuable healthcare data is stolen and sold or ransomed. [5, 16] A recent study has even attributed security breaches to decreasing the quality of care and increasing mortality rates during 3-year windows following a breach.[13] This thesis work primarily studies the possibility of a security breach through software vulnerabilities, instead of the impacts of security breaches as a whole.

Farhadi at al [18] performed static analysis via open-source tool RIPS on one of the open-source web applications studied in the previous works, OpenEMR. Even though the application is written in PHP instead of Java, the analysis reported similar vulnerabilities to the results in this thesis work e.g. cross-site scripting, uncontrolled resources, and access control. They highlighted application design characteristics that caused the vulnerabilities by mapping the vulnerabilities to unmet HIPAA security requirements. They plan future work to analyze other open-source EHR applications and the design characteristics that threaten EHRs.

To secure EHRs, Rezaeibagha et al recommends defining and mandating access control policies through proper standards to increase patient privacy.[33] While this thesis work does not study privacy explicitly, vulnerability type access control is detected by multiple techniques. EHRs could

[1] https://www.healthit.gov/topic/certification-ehrs/certification-health-it
[2] https://www.hhs.gov/hipaa/index.html

also benefit from adopting an easy-to-use multi-factor authentication practices (MFA) to decrease the number of compromised accounts[16].

3.2 Vulnerability Detection Techniques

This thesis work empirically measures effectiveness and efficiency of detection techniques systematic manual penetration testing, automated penetration testing, and automated static analysis against an open-source web application. Our metrics include vulnerability count, true and false positive counts, false positive rate, the number and variety of vulnerability types detected, and vulnerabilities detected per hour. Other recent works have use different metrics, techniques, methodology, and test subjects.

Cyber agility, the ability for a system to adapt dynamically as security threats or system functionality changes, is a dynamic security metric recently adopted.[12] All of the metrics in this thesis work are static and the target application is evaluated at one release version, not based on changes over time.

In 2018, Seng et all [34] conducted a systematic review of the methodology and metrics behind quantifying web application security scanners. The study concluded both the methodologies and metrics used in research are extremely diverse and recommends future work to produce a methodology and metric system to quantify web application security scanners. For example, the study documented 42 different metrics. The most popular metrics were number of vulnerabilities, the number of false positives, and the number of false negatives. Per category, the most frequent metrics were:

- test coverage: number of web pages visited.
- attack coverage: number of test cases generated.
- vulnerability detection: number of vulnerabilities.

- efficiency: scanning time (time required to "complete a vulnerability scanning")

The study also discovered that SQL injection and cross-site scripting are the most common web application vulnerabilities quantified (with 32.6% and 22.4% frequencies).

Alsaleh et al [1] performed a comparison of the performance of open-source web vulnerability scanners to examine detection capability and agreement between the scanners. For detection capability four scanners evaluated vulnerable applications IBM's AppScan test suite and Web application Vulnerability Scanner Evaluation Project (WAVSEP). For scanner agreement, 140 URLs were collected and scanned by two of the tools. The study noted significant variance in type and numbers of detected web vulnerabilities between the tools, high levels of disagreement, and a lack of significance in performance metrics e.g. speed, crawler coverage.

Antunes and Vieira [4] and Nunes et al [28] have analyzed metric selection for evaluation of software vulnerability detection tools per a specific "scenario". Scenarios are general descriptions of real-world applications:

- applications where removing as many vulnerabilities as possible is the primary goal and the resources required is not important (high detection).
- applications that require high detection, but also consider resources available.
- applications where vulnerability detection and resource utilization are equal priorities.
- applications where the primary goal is to reduce the amount of resources required (low false positive rate).

This thesis performs a case study on a real-world application instead of a vulnerable application that is seeded with known vulnerabilities. We cannot use the same metrics evaluated by Antunes, Vieira, and Nunes et al such as recall, precision, and f-measure that require a finite and defined vulnerability count. However, this thesis work is similar in concept as we also consider resources available to a practitioner when providing recommendations on technique selection, e.g. time-constraints.

Nunes also concluded that static analysis tool selection depends on the class of vulnerabilities being detected in addition to resource-constraint scenarios. This thesis work made a similar conclusion that technique selection depends on the type of vulnerability being detected in addition to resource-constraints.

Morrison et al [26] determined vulnerabilities are most often found by testing involving a high number of input variations while comparing vulnerability detection to the detection of non-security defects. Vulnerability detection may therefore be able to be significantly reduced upon further advancement of automated testing and fuzzers.

CHAPTER

4

METHODOLOGY

We conducted an empirical study to aid software practitioners in understanding the effectiveness and efficiency of vulnerability detection techniques. We based our approach on previous works [6–8, 36]. First, we collected reports of potential vulnerabilities generated by each detection technique. Next, we classified each of the potential vulnerabilities as true or false positive. Then, we analyzed each vulnerability type classified to determine if it could be detected at least once by each distinct technique. We also classified each vulnerability type as implementation bug or design flaw. Finally, we compared the vulnerability counts, false positive rates, effectiveness at detecting vulnerability types, and efficiency as vulnerabilities detected per hour. The following subsections explain this procedure in more detail.

4.1 Subject Selection

This section describes the subject evaluated in this case study as well as the subjects evaluated in prior works.

4.1.1 Prior Works

The prior works this thesis is replicating [6–8] evaluated three open-source EHR web applications, Tolven Electronic Clinician Health Record (eCHR)[1] and OpenEMR[2], as well as one open-source client-server EHR, PatientOS[3]. Tolven eCHR and OpenEMR were being used within the United States when they were evaluated in 2011. PatientOS was being used within the United States when it was evaluated in 2013.

4.1.2 Case Study: OpenMRS

Our case study was conducted on Open Medical Records System (OpenMRS) which is in the electronic health records domain, similar to the prior works discussed in Section 4.1.1. OpenMRS[4] is an open-source, EHR web application with Java as its main language. OpenMRS was created in 2004 as a non-profit collaborative open-source application for use in developing countries. The platform claims non-commercial support from "international and government aid groups, NGO's, as well as for-profit and non-profit corporations". The absolute number of contributors is unknown, however, the community's discussion forum currently lists 4,382 members of varying activity.

In addition to being comparable to the subjects selected in the prior work, OpenMRS is comparable to modern web applications that incorporate a client-server model, wrapper API calls to the database, and add-on modules that extend core application functionality. Table 4.1 compares the

[1] https://sourceforge.net/projects/tolven/
[2] https://www.open-emr.org/
[3] https://sourceforge.net/projects/patientos/
[4] https://openmrs.org/

specifications of the previous subjects collected by the prior researchers with the specifications of OpenMRS.

Table 4.1 Specifications of subjects from previous works compared to OpenMRS

	Tolven eCHR	**OpenEMR**	**PatientOS**	**OpenMRS**
Case Study	prior work	prior work	prior work	current study
Language	Java	PHP	Java	Java
Version evaluated	RC1 (5/28/2010)	3.1.0 (8/29/2009)	0.99 (1/17/2010)	RA 2.8.0 (6/27/2018)
Lines of code (counted by CLOC[5])	466,538[1]	277,702[1]	487,437[1]	871,838[2]

[1] counted by CLOC version 1.08.
[2] counted by CLOC version 1.84.

4.2 Detection Technique and Tools

The following subsections describes the methodology used to evaluate each detection technique and describes the tools chosen to represent those techniques. The methodology for systematically classifying vulnerability types as implementation bug or design flaw in discussed in Section 5.1.

4.2.1 Systematic Manual Penetration Testing

The prior work by Smith and Williams [36] created a systematic manual penetration test plan from a software system's functional requirement specification. The specification was based on the certification criteria for ambulatory (outpatient) and inpatient EHR systems defined by the Certification Commission for Health Information Technology (CCHIT)[6]. The work took the English written requirements and broke them down into phrase types then used those types and the patterns

[6]https://www.cchit.org/

discovered to systematically generate 137 black box test cases, using the process defined in [35, 36].

The systematic manual penetration test plan used in this thesis work was based upon the Application Security Verification Standard (ASVS) version 4.0[7] web application security requirements guide. ASVS was adopted because it was created by the OWASP[8] community, a well-known and trusted non-profit security organization. Version 4.0 incorporated the NIST 800-63-3 Digital Identity Guidelines[9] as well as a mapping from the test outline to a CWE weakness identifier. The ASVS guide is a general-purpose standard for the entire software development life cycle with three defined levels of security with increasing depth. Level 1 is the lowest, designed to be penetration testable. Level 2 requires additional protection for applications that contain sensitive data and is the level ASVS recommended for most applications. Level 3 is designed for applications that contain high value sensitive information and requires the most in-depth protection.

The ASVS guide is written into individual conditions a system is required to pass in order to be security at the assurance level the condition applies. Level 3 requires a system pass all ASVS defined conditions, level 2 requires a specified subset of all conditions, and level 1 requires an even smallest specified subset. Fig. 4.1 is an example of a condition written for cross-site scripting attacks required by all three levels of security assurance.

Figure 4.1 Example of a condition from OWASP's ASVS Guide

#	**Description**	**L1**	**L2**	**L3**	**CWE**
5.3.3	Verify that context-aware, preferably automated - or at worst, manual - output escaping protects against reflected, stored, and DOM based XSS.	X	X	X	79

ASVS conveniently lists a single mapped CWE identifier to every individual condition as a reference. Systematic manual penetration test cases were based on the 127 level 1 ASVS conditions

[7] https://github.com/OWASP/ASVS/tree/master/4.0
[8] https://www.owasp.org/index.php/Main_Page
[9] https://www.nist.gov/publications/digital-identity-guidelines

by turning the conditions into expected results which the target application could pass or fail. Section 4.2.1.1 describes the structure of test cases in more detail.

We created 81 test cases some of which contained multiple expected results, allowing for 139 "pass" or "fail" test results. However, only 77 test cases were executed on the target application after discarding 4 that could not be executed, allowing for 133 expected results. The discarded test cases are catalogued in Appendix A.4. One (Test Case 5-L14 Legacy Register a Patient) was missing application knowledge that was necessary for the tester to execute the test case. A note has been added to the test case in the appendix to include the necessary information. The other three (test cases 2.2.3-2, 2.5.4-1, and 2.5.3-1) were discarded for not being relevant to the target application. For example, test cases 2.5.3-1 tests authentication recovery, but the target application does not employ a recovery mechanism.

The target application was too large (more than 870 thousand lines of code, as shown in Table 4.1) to analyze or test every page and function in full, and our research goal could be accomplished with partial analysis/testing. Test locations were first determined by relevance to the test cases. For example, test cases regarding authentication were performed on the pages that contained application authentication. If multiple locations were viable, we selected locations based on application functionality. The functionality critical to the application was targeted for analysis and testing. The test functionality locations are listed in each test case, viewable in Appendix A.2.

We created test cases from the following ASVS defined sections: Authentication (Section 2), Access Control (Section 4), Validation, Sanitization, and Encoding (Section 5), and Logging (Section 7). Discussed further in Section 7, a limitation of utilizing ASVS as a guide for systematic manual penetration testing is that applying the depth of the guide to a system of production size like the target application by a few individuals over a short period of time is impractical. Sections were prioritized by relevance to the functionality critical to the application. The following offers a brief summary of why a given section was not tested:

- Section 1 (Architecture) does not contain and level 1 conditions.
- Section 3 (Session Management) was determined to be non-essential due to overlapping conditions and vulnerability types with every section that was tested.
- Section 6 (Cryptography) was determined to be non-essential for critical application function.
- Section 8 (Data Protection) was determined to be non-essential due to addressing data security policies. For example, the target application has no functionality for patients to login and view their data. Sensitive information exposure is also covered in other sections that were tested.
- Section 9 (Communications) conditions regarding configuration of communication paths, e.g. TLS, was determined to be non-essential.
- Section 10 (Malicious Code) conditions covering integrity controls on automatic updates and third-party sources such as plugins was determined to be non-essential.
- Section 11 (Business Logic) conditions covering anti-automation controls like actions being performed in realistic human time or order were determined to be non-essential.
- Section 12 (Files and Resources) tests the security regarding file transfer, execution, and storage, none of which are critical functions of the application.
- Section 13 (API and Web Services) was determined to be non-essential due to overlapping conditions and vulnerability types with every section that was tested.
- Section 14 (Configuration) was determined to be non-essential for testing server configurations as well as for containing overlap with other sections that were tested.

The same researcher developed every test case. Two separate researchers executed the test cases on the target application. The use of common non-automated penetration tools was allowed as long as the tests were executed manually. Specifically, we used the web proxy server feature of OWASP

ZAP to act as a man-in-the-middle between the web application and the browser, the Requester feature of OWASP ZAP to send direct REST requests, and Postman[10] - another free API client that can send REST requests.

4.2.1.1 Test Case Form and Structure

Every created executable test case has the following form:

- **Unique pattern ID**: the unique pattern ID was in numeric form and was based on the original ASVS organization convention. For example, the first executable test case created from the ASVS guide at test outline 2.1.1 would be given ID 2.1.1-1. Test cases that utilized a legacy functionality within the target application were labeled with an "L" before the test case number, e.g. 2.1.1-L1.
- **Test location**: a function in the target application reachable via direct URL or instructions when a URL cannot be given.
- **Assumptions**: a list of necessary actions that need to be completed before the test case can be completed or necessary background information, if any.
- **Test Procedure**: the numbered test procedure.
- **Expected Results**: The requirements for the target application to pass the test procedure defined from using the ASVS guide as a template. Expected results were numbered.

The ASVS guide already organizes each test outline by a target vulnerability type and maps each outline to a CWE weakness. To make the tests easier to both write and execute, test patterns were implemented for each ASVS section. Test patterns are test case templates that ensure each test case within a section are as similar in assumption, procedure, and expected result as possible. The patterns are defined in Appendix A.

[10]https://www.getpostman.com

4.2.2 Automated Penetration Testing

The prior work performed automated penetration testing with IBM Rational AppScan 8.0. The researchers gave AppScan the authentication credentials to the systems and left scanning options on default settings.

In this study, automated penetration testing was performed by OWASP's Zed Attack Proxy (ZAP).[31] ZAP is a free, open-source automated penetration tool that is designed to be used by testers of all security experience. ZAP can run a full automated scan against an application as well as execute individual tasks, such as fuzzing.

For full, in-depth testing, OWASP recommends first walking through the application manually with a browser and the ZAP browser's proxy, followed by both an automated web-crawler (the spider) and an AJAX web-crawler, then the automated active scan. The active scan uses known attacks against the target application to find potential vulnerabilities. We followed all of OWASP's recommendations listed above with default settings set in ZAP with the exception of configuring ZAP with the target application's admin authentication to allow ZAP to access the full functionality of the application while scanning. ZAP returned a report of potential vulnerabilities when scanning was complete.

A full active scan of the target application was too large for ZAP to run successfully. The active scan was therefore broken up into parts and reported separately from each other such that the entire application was scanned with one vulnerability type at a time. Some similar vulnerability types were scanned together, e.g. the persistent and reflective cross-site scripting scans ran in succession within the same active scan session. Virtual machine snapshots enabled all the scans to be initiated with the same seed of sites saved by the ZAP proxy.

4.2.3 Static Analysis

The prior work performed static analysis with Fortify 360 v.2.6. The researchers chose configuration options "Show me all issues that have security implications" and "No I don't want to see code quality issues".

Static analysis in this case study was performed by an automated proprietary tool. Due to our licensing restrictions, we are unable to disclose the name of the tool. The tool supports 19 languages and over 70 different frameworks used by those languages including Java, JavaScript, JSP, HTML, and Ruby which the CLOC scan discovered in the target application's code base. Code scanner CLOC[11], used to determine the specifications of the target application listed in Section 4.1.2, calculated the languages listed above as 81% of the target application's code. To evaluate these languages, the automated static analysis tool was configured to scan for all web application security defects. A report of potential vulnerabilities was generated by the static analysis tool when scanning was complete.

[11] https://github.com/AlDanial/cloc

CHAPTER

5

RESULTS

This section contains the results of the analysis completed by the three detection techniques. To compare the effectiveness of each technique, we counted the number of reported potential vulnerabilities, true positive vulnerabilities, false positives, the number of true positive vulnerability types, and the number of implementation bugs and design flaws detected. To compare efficiency, we computed vulnerabilities discovered per hour and false positive rate. Vulnerabilities discovered per hour only includes true positive vulnerabilities, not false positives.

5.1 Implementation Bugs and Design Flaws

OWASP's Top 10 [30] Critical Web Application Risks and IEEE Center's Top 10 Design Flaws [40] provide a few examples of implementation bugs and design flaws, however, not every vulnerability

type detected in this thesis work is mentioned. There is no standard classification of every vulnerability type into implementation bugs and design flaws to reference within existing literature. Instead, we first checked the CWE[1] phase descriptions of the vulnerability type where CWE often defines in which phases in the software development life cycle the type occurs, e.g. architecture, implementation, operation. If multiple phases were documented, we compared the description of the vulnerability type given by the detection technique to the description of the CWE phase, if available, to determine if the type was occurring as implementation that could be easily patched or architecture that required redesign.

Table 5.1 contains the vulnerability types we classified as implementation bugs, Table 5.2 contains the design flaws. Section 2.4 provides a reference for each vulnerability type.

5.2 Systematic Manual Penetration Testing

The following subsections contain the results for systematic manual penetration testing.

5.2.1 Metrics

The ASVS v4.0 guide contains 127 level 1 required conditions that are mapped to CWE-identifiers. Using this guide, we implemented 81 test cases some of which contained multiple expected results, allowing for 139 "pass" or "fail" test results. In total, 55 vulnerabilities were detected when the expected results resulted in "fail". The detected vulnerability types for each ASVS section are discussed in future subsections. The test cases were created over approximately 27.8 hours and were executed over 8.8 hours. The vulnerabilities per hour metric is $55/36.6 = 1.5$. There were 11 different

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

vulnerability types: 4 implementation bugs and 7 design flaws. Since all vulnerabilities discovered during manual testing are true positives, false positive rate is 0. These finds are summarized in Table 5.3.

5.2.2 Vulnerability Types

The following subsections discuss the vulnerability types detected in each ASVS section. Table 5.4 summarizes the results by vulnerability type, ordered by the CWE identifier.

5.2.2.1 Authentication

We created 40 test cases with 53 total expected results from the 27 conditions defined in the ASVS Authentication level 1 guide (2.1, 2.2, 2.3, 2.5) over approximately 8.2 hours. Three test cases were discarded due to not being compatible with the target application. For example, ASVS level 1 Section 2.5.6 tests authentication recovery, however, the target application does not contain any authentication recovery mechanism. The target application failed 16 / 49 expected results over approximately 3 hours of testing. Including creation time, this equates to 16 vulnerabilities /11.2 hours = 1.4 vulnerabilities per hour. The detected vulnerabilities were mapped to the following types by ASVS, in order of occurrence:

- **Weak password requirements (14)** (CWE-521): strong passwords are not required making it easier for attackers to compromise user accounts.
- **Use of insufficiently random values (2)** (CWE-330): initial passwords generated by the admin at user creation are not random and do not expire after a short period of time.

5.2.2.2 Logging

We created 14 test cases with 15 total expected results from three ASVS conditions, 3.1.1, 7.1.1, and 7.1.2, defined in the ASVS level 1 guide over approximately 3.2 hours. The target application failed 2 / 28 expected results over approximately 0.8 hours of testing. Including creation time, this equates to 2 vulnerabilities / 4.0 hours = 0.5 vulnerabilities per hour. The only vulnerability type detected was **inclusion of sensitive information in log files (CWE-532)**: the logs could potentially contain sensitive information, such as sensitive patient data, because the application appends user input to the log.

5.2.2.3 Access Control

We created 11 test cases with 13 total expected results from 7 ASVS conditions, 4.1.3, 4.1.4, 4.1.5, 4.2.1, 4.2.2, 4.3.1, and 13.1.2, defined in the ASVS level 1 guide. The target application failed 10 / 13 expected results over approximately 0.75 hours of testing. In total, 26 vulnerabilities were detected as some test cases resulted in multiple possible vulnerabilities. Vulnerabilities per hour cannot be defined for this individual section because these test cases were created at the same time as validation, sanitization, and encoding. The detected vulnerabilities were mapped to the following types by ASVS, in order of frequency of occurrence:

- **Improper authorization (8)** (CWE-285): the application did not perform or incorrectly performed an authorization check allowing an unauthorized actor to access a resource or perform an action.
- **Incorrect default permissions (8)** (CWE-276): new user accounts do not start with minimal permissions such that the users receive access to features before being assigned.
- **Authorization bypass through user-controlled key (8)** (CWE-639): sensitive data and services are not protected against directed object attacks.

- **Unprotected primary channel (2)** (CWE-419): an administrative interface is not protected against unauthorized use. Specifically, ASVS checks for multi-factor authentication.

5.2.2.4 Validation, Sanitization, and Encoding

We created 16 test cases with 45 total expected results from the following 19 sections of the ASVS level 1 guide: 5.1.1, 5.1.3, 5.1.4, 5.1.5, 5.2.2, 5.2.4, 5.3.3, 5.3.4, 5.3.5, 5.3.6, 5.3.7, 5.3.8, 5.3.10, 5.5.1, 5.5.2, 5.5.3, 5.5.4, 7.4.1, and 13.2.2. The target application failed 10 / 43 expected results over approximately 4.25 hours of testing. One test case was not executed (5-L14 Legacy Register a Patient) because application knowledge necessary for the tester to execute the test case was absent. A note has been added to the test case in the appendix to include the necessary information. The injection test cases all resulted in the storage of the attack string, but never execution, and were classified as improper input validation (CWE-20) instead of classifying each vulnerability by vulnerability type (e.g. cross-site scripting whenever an cross-site scripting attack is stored) because we cannot prove that the application will never call and execute the executable attack strings. We also did not want to skew the number of implementation bugs detected by multiplying the number of locations where attack strings were being stored by the number of vulnerability types that could be represented as attack strings. Vulnerabilities per hour cannot be defined for this individual section because these test cases were created at the same time as access control. The detected vulnerabilities were mapped to the following types by ASVS, in order of occurrence:

- **Improper input validation (7)** (CWE-20): an incorrect or lack of input validation can affect the data flow of the application. Stored attacks that could be called later were included. Addresses that were not validated (city, zip code pairs) were also included.
- **Information exposure through self-generated error message (2)** (CWE-210): The application identifies an error condition and creates its own error message that contains sensitive information.

- **Deserialization of untrusted data (1)** (CWE-502): untrusted JSON data is deserialized without sufficiently verifying that the resulting data will be valid.
- **Parameter pollution (1)**(CWE-235): the application does not handle or incorrectly handles when the number of parameters, fields, or arguments with the same name exceeds the expected amount. The target application generally chooses one.

5.3 Automated Penetration Testing

The following subsections contain the results for automated penetration testing performed with OWASP's ZAP.

5.3.1 Metrics

ZAP detected 20,781 potential vulnerabilities. We were able to filter the list down to 5,170 by removing duplicate attacks which is explained further in Section 5.3.2. The output generated by ZAP was organized by vulnerability type and reported information about each potential vulnerability. For instance, the cross-site scripting output reported URI, HTTP verb, parameter being attacked, and attack string. The number of detected potential vulnerabilities per type ranged from 1 to 6,565.

Table 5.5 contains an overview of the metrics calculated for ZAP. There were 4,178 true positive vulnerabilities and 998 false positives evaluated over approximately 28.8 hours. Out of the true positive vulnerabilities, 6 were counted for two separate types: once as cross-site scripting and once as information exposure because they were successful cross-site scripting attacks that also caused the application to crash and expose sensitive information about the system's environment. The vulnerabilities per hour metric is $4,178/28.8 = 144.9$. There were 10 different vulnerability types: 8 implementation bugs and 2 design flaws. The false positive rate is 19.3%. Table **??** displays the results by vulnerability type where "LBR" stands for Labeled-by-Researcher. The process for evaluating potential vulnerabilities as true or false positive is described for each vulnerability type in following

subsections.

5.3.2 Vulnerability Locations

ZAP does not report the location of code where the vulnerability is detected. Instead, it often lists a URI, HTML method (e.g.: GET, POST), and parameter used to attack the target application. However, the location of the code where the vulnerability is located can sometimes be approximated when URIs are formatted as `../module/<module-name>/...` by the application.

ZAP would sometimes mistake URIs that contained different combinations of parameters as multiple separate URIs. Vulnerability locations were determined to be the same if the given URIs differed only by combinations of parameters. The exception to this was Application Error Disclosure. Since the application would react differently to different combinations of parameters. For example, if a specific parameter was missing, the application might crash and disclose sensitive information.

5.3.3 Vulnerability Types

There were 10 CWE-mapped detected vulnerability types reported by ZAP. Some of the CWE-identifiers were repeated, allowing for 8 implementation bugs and 2 design flaws. Table 5.6 summarizes the results by vulnerability type. The vulnerabilities were mapped into the following types, appearing in order of frequency of occurrence:

- **Application error disclosure (2725)** (CWE-200): An application error was detected, such as HTTP 500 (internal server error) responses. Since the automated penetration tool mapped the vulnerability type to CWE-200 (Information Exposure), we also included all sensitive information exposures regardless of the presence of a HTTP 500 error.

- **Information exposure through an error message (1347)** (CWE-209): Labeled by the researcher, not by the automated penetration tool, when an attack caused sensitive information exposure

through an error message. Information could be related to the application environment or application data.

- **Improper input validation (51)** (CWE-20): Labeled by the researcher, not by the automated penetration tool, when an attack caused the application to crash, but did not cause information exposure.

- **Parameter tampering (37)** (CWE-472): a known parameter value is sent to the application and the HTTP response is checked for server error status codes and error content commonly found in Java, Microsoft VBScript, OLE DB, JET, PHP, and Tomcat. Mapped to CWE-472: the application does not sufficiently verify externally controlled inputs.

- **Cross-site scripting (10)** (CWE-79): externally controlled input is not sufficiently validated, allowing malicious code to be injected into a web browser..

- **Format string error (3)** (CWE-134): external string inputs are not sufficiently validated, causing application crashes and abnormal session closures.

- **Private IP Disclosure (2)** (CWE-200): sensitive information in the form of a private or internal IP address was detected in an HTTP response. Mapped to CWE-200 information exposure.

- **Cookie No HttpOnly Flag (1)** (CWE-16): a cookie is not flagged to HttpOnly, allowing the cookie to be used by client side scripting. Mapped to CWE-16 configuration.

- **Improper Access Control (1)** (CWE-284): labeled by the researcher, not by the automated penetration tool, when a lack of access control allowed the execution of a service by an unauthorized actor.

There are two design flaws: improper access control and parameter tampering. The other 8 vulnerability types are implementation bugs. The following sub-sections explain in more detail how these vulnerabilities were detected and counted.

5.3.3.1 Labeled-by-Researcher

The methods the automated penetration tool utilized to detect vulnerabilities were not always obvious and documentation was sparse. However, based on community discussion forums as well as the personal observations of the researcher, it seemed that the tool often detected potential vulnerabilities simply when noticing a change in the HTML given by the HTTP response. For instance, the tool would try an SQL injection attack and would notice an HTML change and label the SQL injection attack as successful, when it actually returned an error page because the SQL injection failed.

The target application does not handle exceptions or validate unexpected input well. This caused a large number of information exposure vulnerabilities to be manually detected and labeled by the researcher while auditing other vulnerability types for true/false positives. If an attack caused a page to crash, but did not display sensitive information, it was labeled as improper input validation. The sensitive information was always important system environment information, such as a Java stack trace. The improper access control was detected from a potential vulnerability that was originally labeled as a successful SQL injection attack.

The format string error and parameter tampering true positive vulnerabilities also displayed information exposure or input validation vulnerabilities. However, those were not added to the researcher labeled types because they were considered the successful result of those individual vulnerability types.

5.3.3.2 Application Error Disclosure

The detection tool reported 3390 instances of the application disclosing sensitive information through an error message. The locations of these attacks were checked against all other vulnerability types so as to not include duplicates with the labeled-by-researcher information exposure vulnerability type. Only 2 duplicates were found from within the detection tool application error

disclosure report itself. No duplicates among the other vulnerability types were found. Therefore, there were 3388 instances of potential vulnerabilities.

The tool claims to report potential vulnerabilities when returned evidence of HTTP 500 - Internal Server Error responses, however that was often not the case when checked by the researcher. False positives were often generated from error messages produced when a required parameter was missing but no unexpected application behavior or information exposure was observed.

5.3.3.3 Parameter Tampering

There were 2 instances of duplicate attacks where the same parameter was deleted from the same URI and HTTP method location, as discussed in Section 5.3.2, therefore there were 37 total instances. All 37 instances of the attack caused the target application to crash and display an error page exposing sensitive application environment information. All 37 vulnerabilities were therefore counted as true positives for only parameter tampering.

5.3.3.4 Cross-Site Scripting

The detection tool reported 76 potential vulnerabilities over three separate risk levels of cross-site scripting: reflected in response low risk, reflected in response medium risk, and reflected in JSON response. Only the 10 instances in the reflected medium risk were true positives for cross-site scripting. There was only one false positive at the reflected in JSON response risk level; the rest were false positives for low risk cross-site scripting. All of the false positives cross-site scripting vulnerabilities and 6 of the true positives were positive for information exposure as the application would crash and display a error page with sensitive application environment information. The false positives were labeled by the research as information exposure through an error message vulnerabilities. The 6 cross-site scripting vulnerabilities were also labeled as true positive for information exposure through an error message in addition to being labeled true positive for cross-site scripting.

5.3.3.5 Format String Error

The detection tool reported 252 potential format string vulnerabilities. The tool defined a format string vulnerability with CWE-134, "the software uses a function that accepts a format string as an argument, but the format string originates from an external source." The detection tool repeated some attacks on the same location, as discussed in Section 5.3.2. After filtering through duplicates, the report is reduced to 196 potential vulnerabilities, 2 of which were false positives. The other 194 were vulnerabilities due to crashing the application and resulting is an error page that disclosed sensitive environment information. Of these 196 attacks, 192 were of the same "lang" parameter. This parameter is used by the application to change locale settings. Since the functionality is the same whenever it is used, it was counted as one vulnerability. The end result is 3 vulnerabilities and 2 false positives.

5.3.3.6 Private IP Disclosure

The tool reports an information exposure of a private IP address when it discovers RFC 1918 Ipv4 addresses as well as Amazon EC2 private hostnames in HTML responses. The tool was successful at finding these private IP addresses, however, the false positives were example IP addresses left in text messages from the developer to the user. The true positive vulnerabilities were instances of the database IP address that was hosted by a local Docker container.

5.3.3.7 SQL Injection

The detection tool reported 1005 potential SQL injection vulnerabilities. The tool repeated the same attack on four URI locations, as discussed in Section 5.3.2. Therefore, there were 1001 total potential vulnerabilities. 983/1001 or 98% of the potential vulnerabilities were located in the webservices module and could quickly be filtered by the "/ws/rest/" in the given URL. All of these potential vulnerabilities from the webservices module displayed an error page with sensitive application

environment information and were labeled by the research as information exposure through an error message. The error message was for not being able to find a matching object ID in the database. Out of the 18 potential vulnerabilities not from the webservices module, 14 were also labeled by the research as information exposure through an error message. There were 4 false positives that did not display an error page or result in a successful SQL injection attack. As explained in Section 5.3.3.1, the tool detected these 4 false positives because the attack resulted in a change in HTML in the HTTP response. The SQL injection potential vulnerability detected via MySQL injection was a false positive for SQL injection, but a true positive for improper access control because the application would generate a JSON with a random user ID regardless of any attacks on parameters. The final count of true positives is 997 vulnerabilities.

5.3.3.8 Buffer Overflow

The detection tool reported 4762 potential buffer overflow vulnerabilities. The tool did not report every vulnerability in the same systematic way. Some vulnerabilities were missing the parameter that was attacked and/or the attack used to generate the buffer overflow error. A script was used to filter the attacks by URI location, HTML method, and attack parameter, if it was listed, as discussed in Section 5.3.2 to produce 1862 attacks on different parameters. Since this application was not written in any language that is susceptible to buffer overflow attacks, all of the potential vulnerabilities are false positives.

The detection tool looked for buffer overflows by sending out large strings to cause crashes or abnormal session closure. Since the application does poorly with validation and unexpected exceptions (discussed in Section 5.3.2, many of the potential vulnerabilities labeled for buffer overflow where positive for information exposure through an error message and were labeled as such by the researcher. Due to the non-systematic way the tool was reporting the buffer overflow attacks, pruning the 1862 different parameters was tedious and time-consuming. Two researchers collectively spent 18.3 hours to review a random subset of 655, or 36%, of the 1862 attacks. Five

different patterns of information exposure error messages the application was known to produce were identified before hand to ensure both researchers would acquire the same results. Out of the 655 potential vulnerabilities reviewed, there were 279 information exposure through an error message vulnerabilities labeled by the researcher, 51 improper input validation vulnerabilities labeled by the researcher where the application would crash, but no information exposure would occur, and 335 false positives.

5.3.3.9 Other

Web Browser XSS Protection Not Enabled: The potential vulnerabilities detected for the X-XSS-Protection header not being enabled were ignored because the CWE page referenced by the detection tool is marked as "obsolete" and modern web browsers are either retiring or never implementing the feature.

Cookie No HttpOnly Flag: The tool detected 1696 potential vulnerabilities for cookies not being set to HTTP only. This issue was counted as one vulnerability because only one cookie was detected with this issue. This issue was counted as a vulnerability even though the HTTP only flag can be set by the server because it was not clear if the developers of the target application intended the cookie to be accessed through client side scripting.

X-Frame-Options Header Not Set: The 2979 detected potential vulnerabilities for the lack of the X-FrameOptions header to prevent clickjacking attacks were also ignored. While this header can be set in the application's code, it could also be configured by the server. Since the researcher configured the server, all server-based vulnerabilities were ignored.

X-Content-Type-Options Header Missing: The 4193 detected potential vulnerabilities for the lack of the X-Content-Type-Options header being set to "nosniff" to prevent MIME type sniffing were also ignored. While this header can be set in the application's code, it is typically set by the server. Since the researcher configured the server, all server-based vulnerabilities were ignored.

Content-Type-Options Header Missing: The 2 detected potential vulnerabilities for the lack of

the Content-Type header were both false positives. They were counted as one false positive because the attacks were determined to be on the same as discussed in section 5.3.2.

5.4 Automated Static Analysis

The following subsections contain the results for automated static analysis.

5.4.1 Metrics

The static analysis tool reported 295 potential vulnerabilities after selection for security defects only.

Some of the potential vulnerabilities reported by the static analysis tool were grouped into one individual vulnerability. Example 1: the static analysis tool detects a vulnerability in a code segment multiple times. Each time, a different path in the code was taken because of if-statement branching or multiple functions calling the function that contained the code segment. Example 2: multiple similar code segments are detected and grouped as one vulnerability. Often, the code segments were in the same function. Since each individual potential vulnerability was validated as true or false positive, grouped potential vulnerabilities were added to the total number of potential vulnerabilities. One potential vulnerability was subtracted from the total because the detection tool reported the vulnerability twice in two separate modules. The final number of potential vulnerabilities was 332: 312 vulnerabilities and 0 false positives evaluated over approximately 21.7 hours. The vulnerabilities per hour metric is $312/21.7 = 14.4$. There were 19 different vulnerability types: 13 implementation bugs and 6 design flaws. False positive rate is 6.0%.

While evaluating the classifications, we noticed some false positives were grouped in files contained in a testing directory. All of the potential vulnerabilities were then checked against this testing directory. Potential vulnerabilities from the testing directory were evaluated because the test directory was discovered late into classification, however, the metrics were re-calculated after discarding potential vulnerabilities from the test directory because neither the manual nor automated pene-

tration testing included application tests. The result was a list of 280 potential vulnerabilities: 260 true positive and 20 false positives after discarding 52 potential vulnerabilities. We calculated the adjusted vulnerabilities per hour metric as 260 true positives /((280/332) potential vulnerabilities $*$21.7 hours) = 14.2 vulnerabilities per hour. There were 18 different vulnerability types: 12 implementation bugs and 6 design flaws. False positive rate of 7.1%. Table 5.7 summarizes the metric calculations.

5.4.2 Vulnerability Types

The vulnerabilities were spread over 25 vulnerability types defined by the static analysis tool. The tool mapped the 25 types to 23 individual CWE identifiers. For example, types resource leak and resource leak on an exceptional path were mapped to the same CWE-404 identifier. There were 6 true positive design flaw vulnerability types: cleartext sensitive data in a database, risky cryptographic hashing function, unsafe reflection, trust boundary violation, unsafe deserilization, and open redirect. The other 16 true positive detected vulnerability types were implementation bugs. Table 5.8 summarizes the results by vulnerability type not including the test files.

The following vulnerabilities types were detected and mapped to a CWE weakness by the static analysis tool. One type, use of a hard-coded password (CWE-259), was discarded for being detected only within the test directory. Types appear in order of frequency of occurrence:

- **Resource leak (43)**: the application does not release or incorrectly releases a resource before it is made available for re-use (CWE-404).
- **Trust boundary violation (39)** (CWE-501): untrusted data flows to a data structure or context that is often assumed to be trustworthy.
- **Unsafe reflection (37)** (CWE-470): external input with the ability to create dynamic data is

passed to a reflection API .

- **Thread unsafe modification in singleton (19)** (CWE-543): the application uses a singleton pattern when creating a resource within a multi-threaded environment.
- **Resource leak on an exceptional path (18)** (CWE-404): a resource leak detected after taking an exceptional path.
- **Cross-site request forgery (17)** (CWE-352): an unauthorized request or action from cannot be distinguished as a legitimate request from the user.
- **DOM-based cross-site scripting (17)** (CWE-79): cross-site-scripting where JavaScript code executed potentially allows session hijacking, disclosing of sensitive data in the DOM, or viewing of keyboard and mouse events.
- **Risky cryptographic hashing function (14)** (CWE-328): a weak hashing algorithm was detected that may be susceptible to collisions.
- **Open redirect (10)** (CWE-601): the application accepts user-controlled input that specifies a link to an external site, and uses that link in a Redirect.
- **Path Traversal (9)** (CWE-22): an attacker can manipulate a path allowing the ability to access, modify, or corrupt application files.
- **Unsafe deserialization (8)** (CWE-502): the application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.
- **Unchecked origin of message event (7)** (CWE-346): the event handler does not check the origin of the received message event.
- **Value not atomically updated (5)** (CWE-567): data is not properly synchronized outside of a locked context. Otherwise known an unsynchronized access to shared data in a multithreaded

context.

- **SQL injection (4)** (CWE-89): input from users is directly used in constructing SQL queries, causing the database to be accessed in unintended ways.
- **Unchecked Target for Cross-Origin Message (4)** (CWE-668): a cross-origin window message can be sent without restricting the origin that can receive it, allowing unintended access to the resource.
- **Cross-site scripting (4)** (CWE-79): the application does not neutralize or incorrectly neutralizes user-controllable input such that JavaScript code is executed or stored in the database
- **Regular Expression Injection (3)** (CWE-94): uncontrolled dynamic data is used as part of a regular expression.
- **Call to an inherently dangerous function (2)** (CWE-242): the application calls a function that can never be guaranteed to work safely.
- **IL: infinite loop (2)** (CWE-674): recursion is not controlled such that the loop will execute forever, or until system resources are exhausted.
- **Cleartext sensitive data in a database (1)** (CWE-313): sensitive data is stored in the database.
- **Socket accepts connections from all origins (1)** (CWE-400): denial of service and sensitive data leakage may occur from the server being open to all connections.
- **Unrestricted document type definition (1)** (CWE-827): an untrusted document type definition can be used to specify an external URL. In this case, it was an XML form being accepted by the application.

The following potential vulnerabilities were detected by the static analysis tool, but were found to be false positives in every occurrence or only occurred in the test files. All detected potential

vulnerabilities are included in the summary of results by vulnerability type, Table 5.8. An overview of the calculated metrics for the automated static analysis tool, including false positives, is in Table 5.7.

- **Use of hard-coded cryptographic key** (CWE-321): users with access to source code can use this key to access encrypted data.
- **Use of hard-coded URI password** (CWE-259): users with access to source code can use this password to access protected services or data.
- **Hashing a password with a weak salt** (CWE-760): passwords can be computed with modest computational effort.

Table 5.1 Detected vulnerability types classified as implementation bugs

Vulnerability Type	**CWE identifier**
Configuration	16
Improper input validation	20
Path Traversal	22
Cross-site scripting	79
SQL injection	89
Code injection	94
Format string error	134
Information exposure	200
Information exposure through an error message	209
Information exposure through self-generated error message	210
Parameter pollution	235
Call to an inherently dangerous function	242
Incorrect default permissions	276
Use of insufficiently random values	330
Insufficient verification of data authenticity	345
Origin validation error	346
Cross-site request forgery (CSRF)	352
Uncontrolled resource consumption	400
Improper resource shutdown or release	404
Thread unsafe modification in singleton	543
Unsynchronized access to shared data in a multi-threaded environment	567
Exposure of resource to wrong sphere	668
Uncontrolled recursion	674
Unrestricted document type definition	827

Table 5.2 Detected vulnerability types classified as design flaws

Vulnerability Type	**CWE identifier**
Improper access control	284
Improper authorization	285
Cleartext sensitive data in a database	313
Risky cryptographic hashing function	328
Unprotected primary channel	419
Unsafe reflection	470
Parameter tampering	472
Trust boundary violation	501
Deserialization of untrusted data	502
Weak password requirements	521
Inclusion of sensitive information in log files	532
Open redirect	601
Authorization bypass through user-controlled key	639

Table 5.3 Systematic manual penetration calculated metrics overview

Metric	**Value**
Total Vulnerabilities	55
Variety of Vulnerability Types	11
Vulnerabilities / Hour	1.5
Implementation Bug Types	4
Design Flaw Types	7

Table 5.4 Systematic manual penetration results by vulnerability type

Vulnerability Type	**CWE Identifier**	**True Positives**
Improper Input Validation	20	7
Information Exposure Through Self-generated Error Message	210	2
Improper Handling of Extra Parameters	235	1
Incorrect Default Permissions	276	8
Improper Authorization	285	8
Use of Insufficiently Random Values	330	2
Unprotected Primary Channel	419	2
Deserialization of Untrusted Data	502	1
Weak Password Requirements	521	14
Inclusion of Sensitive Information in Log Files	532	2
Authorization Bypass Through User-Controlled Key	639	8
Total		**55**

Table 5.5 Automated penetration calculated metrics overview

Metric	**Value**
Total Potential Vulnerabilities	5,182
Variety of Vulnerability Types	10
True Positives	4,178
False Positives	1,010
False Positive Rate	19.5%
Vulnerabilities / Hour	144.9
Implementation Bugs	8
Design Flaws	2

Table 5.6 Automated Penetration results by vulnerability type

Vulnerability Type	CWE Identifier	True Positives	False Positives	False Positive Rate
Application Error Disclosure	200	2725	663	20%
(LBR) Information Exposure Through an Error Message	209	1348	-	-
(LBR) Improper Input Validation	20	51	-	-
Parameter Tampering	472	37	0	0%
Cross-Site Scripting (Reflected)	79	10	0	0.0%
Format String Error	134	3	2	40%
Private IP Disclosure	200	2	5	71%
Cookie No HttpOnly Flag	16	1	0	0%
(LBR) Improper Access Control	284	1	-	-
SQL Injection	89	0	4	100%
Buffer Overflow	120	0	335	100%
Content-Type-Options Header Missing	345	0	1	100%
Total		**4178**	**1010**	**19.5%**

Table 5.7 Automated static analysis calculated metrics overview not including test files

Metric	**Value**
Total Potential Vulnerabilities	280
Variety of Vulnerability Types	22
True Positives	260
False Positives	20
False Positive Rate	7.1%
Vulnerabilities / Hour	14.2
Implementation Bugs	16
Design Flaws	6

Table 5.8 Static analysis results not including test files by vulnerability type

Vulnerability Type	CWE Identifier	True Positives	False Positives	False Positive Rate
Resource leak	404	43	1	2.3%
Trust Boundary Violation	501	39	2	4.9%
Unsafe Reflection	470	33	4	10.8%
Thread unsafe modification in singleton	543	19	0	0%
Resource leak on an exceptional path	404	18	0	0%
Cross-Site Request Forgery (CSRF)	352	17	0	0%
DOM-based Cross-Site Scripting	79	16	1	5.9%
Risky Cryptographic Hashing Function	328	14	0	0%
Open Redirect	601	10	0	0%
Path Traversal	22	9	0	0%
Unsafe Deserialization	502	8	0	0%
Unchecked Origin of Message Event	346	7	3	30.0%
Value not atomcially updated	567	5	0	0%
Cross-Site Scripting	79	4	0	0%
SQL Injection	89	4	0	0%
Unchecked Target for Cross-Origin Message	668	4	0	0%
Code Injection	94	3	0	0%
Call to an inherently dangerous function	242	2	0	0%
IL: Infinite Loop	674	2	0	0%
Cleartext Sensitive Data in a database	313	1	0	0%
Socket Accepts Connections from All Origins	400	1	0	0%
Unrestricted Document Type Definition	827	1	0	0%
Use of Hard-Coded URI Password	259	0	0	0%
Use of Hard-Coded Cryptographic Key	321	0	2	100%
Hashing a password with a weak salt	760	0	7	100%
Total		**260**	**20**	**7.1%**

CHAPTER

6

ANALYSIS & DISCUSSION

The first subsection discusses and analyzes the vulnerabilities discovered by each detection technique. The second subsection discusses the efficiency of each detection technique in terms of vulnerabilities detected per hour. The third subsection highlights the weaknesses in each detection technique through the vulnerability types each technique failed to discover.

6.1 Comparing Vulnerabilities Discovered

In this section, we provide results that aggregate the specific vulnerability types discovered within the target application, OpenMRS. To gain a better understanding of when to use each detection tool, we compare how effective one tool was at detecting the specific vulnerability types found with other tools.

Due to the detection tools having distinct systems of mapping vulnerability types to CWE values, the results contain 37 individual CWE-mapped vulnerability types. The automation penetration tool used more general CWE weaknesses whereas the static analysis tool and the ASVS defined CWE mappings were more specific. The types were condensed such that similar types were grouped into the most general type to compare the techniques without bias towards the techniques that use a larger number of CWE identifiers through more specific mappings. First, we recognized similar vulnerability types based off their CWE identifiers and CWE weakness descriptions mapped by the detection techniques. We then determined which type in a similar set was the most general according to CWE weakness structure and definitions. An explanation on the process of grouping and the internal CWE weakness structure is given in Section 2.4. Groups were only utilized in the following obvious cases:

- CWE-200, -209, -210 were grouped into CWE-200. All three cover the exposure of sensitive information via error pages displayed by the system.
- CWE-400 and -404 were grouped into CWE-400. Both cover the uncontrolled handling of resources.
- CWE-284 and -285 were grouped into CWE-284. Both cover unauthorized actors getting access to a resource due to improper access control.

After condensing the vulnerability types listed above, we reassessed each technique's ability to detect implementation bugs and design flaws, summarized in Table 6.1. Systematic manual penetration detected the most design flaws, but static analysis discovered only one less. Static analysis discovered the most implementation bugs. Based off this analysis, automated static analysis is the most effective at detecting implementation bugs and systematic manual penetration is the most effective at detecting design flaws. A resource-constrained practitioner may detect both implementation bugs and design flaws with static analysis. Fig. 6.1 displays the comparison of number of

implementation bug and design flaw vulnerability types. Without over-counting of the vulnerability type information disclosure, implementation bugs and design flaws occur at almost equal amounts.

We compared the detection techniques all together by vulnerability types. Table 6.2 lists the detected types for all the techniques and Fig. 6.2 offers an easier to see display. The details of Table 6.2 and Fig. 6.2 is discussed in the following subsections.

Table 6.1 Analysis of each technique's ability to detect implementation bugs and design flaws

Detection Technique	**Implementation Bugs**	**Design Flaws**
Systematic Manual Penetration Testing	4	7
Automated Penetration Testing	6	2
Automated Static Analysis	13	6

6.1.1 Automated vs. Systematic Manual Penetration Testing

Fig. 6.3 displays the detected vulnerabilities by type for the two techniques without type information exposure where systematic detected 2 and automated detected 4075.

Automated penetration testing detected 4178 total vulnerabilities. Most of the automated penetration vulnerabilities detected were vulnerability type information disclosure. Ignoring all information disclosure vulnerabilities, automated penetration only discovered 103 vulnerabilities. In total, the 8 detected vulnerability types consisted of 6 implementation bugs and 2 design flaws. Systematic manual penetration testing was able to detect 11 types: 7 design flaws and 4 implementation bugs. The only two types that were detected by both techniques were improper access control and improper input validation. Systematic manual penetration testing was better at improper access control and worse in improper input validation.

Figure 6.1 Comparison of implementation bug and design flaw vulnerability type counts

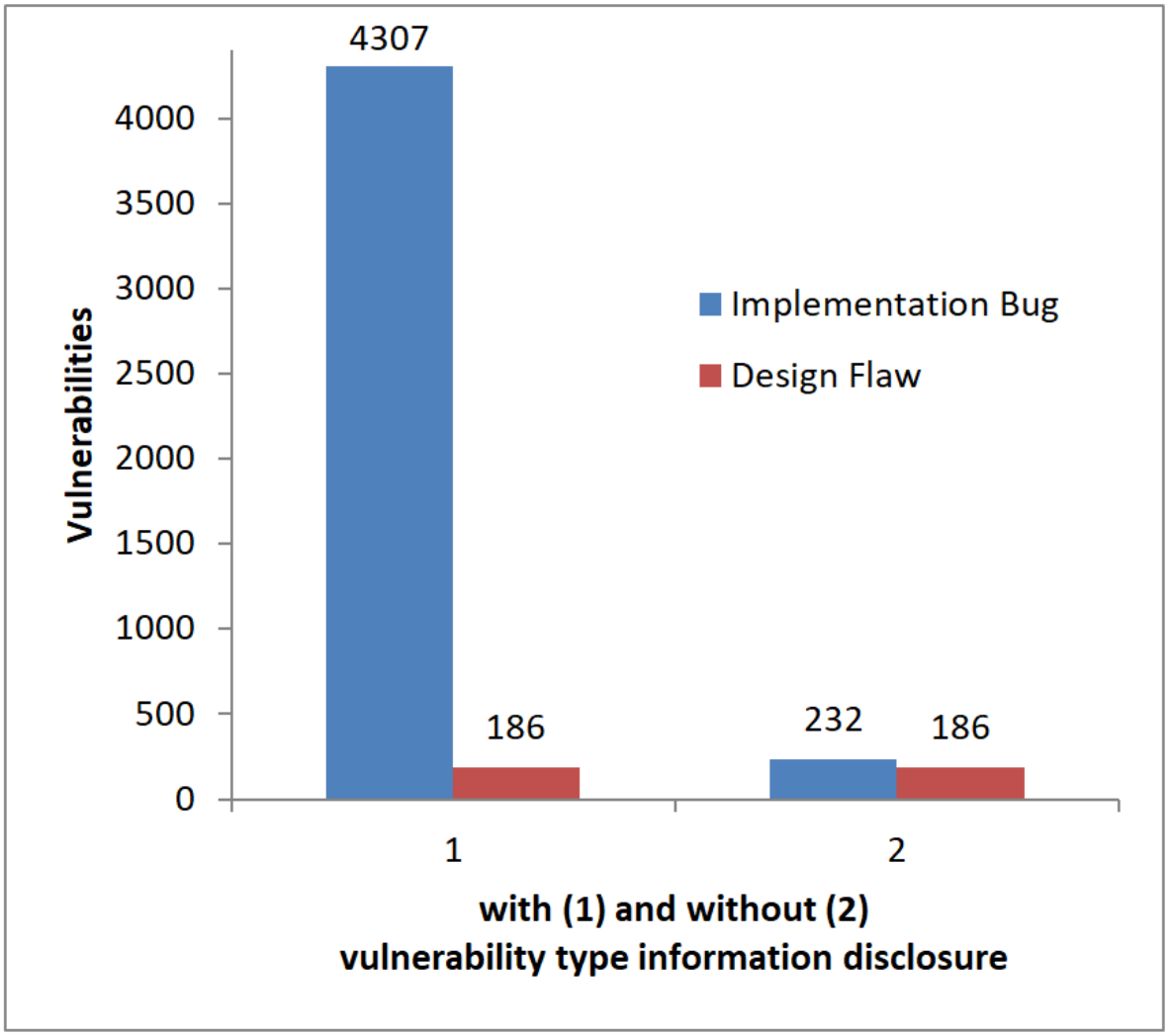
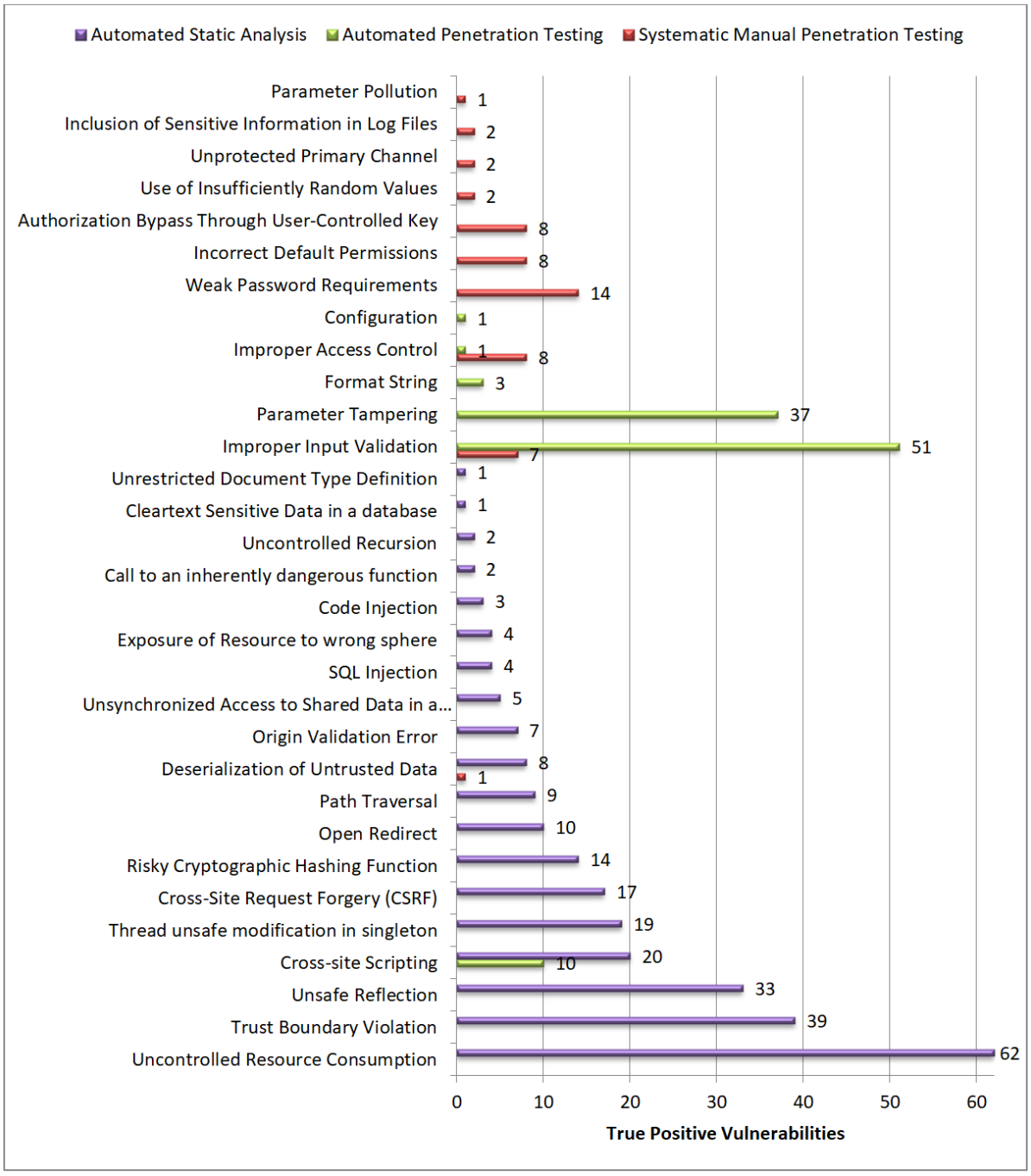


Table 6.2 Results of all detection techniques by vulnerability types

Vulnerability Type	CWE Identifier	Systematic Manual Penetration Testing	Automated Penetration Testing	Automated Static Analysis
Configuration	16	0	1	0
Improper Input Validation	20	7	51	0
Path Traversal	22	0	0	9
Cross-site Scripting	79	0	10	20
SQL Injection	89	0	0	4
Code Injection	94	0	0	3
Format String	134	0	3	0
Information Exposure	200, 209, 210	2	4075	0
Parameter Pollution	235	1	0	0
Call to an inherently dangerous function	242	0	0	2
Incorrect Default Permissions	276	8	0	0
Improper Access Control	284, 285	8	1	0
Cleartext Sensitive Data in a database	313	0	0	1
Risky Cryptographic Hashing Function	328	0	0	14
Use of Insufficiently Random Values	330	2	0	0
Origin Validation Error	346	0	0	7
Cross-Site Request Forgery (CSRF)	352	0	0	17
Uncontrolled Resource Consumption	400, 404	0	0	62
Unprotected Primary Channel	419	2	0	0
Unsafe Reflection	470	0	0	33
Parameter Tampering	472	0	37	0
Trust Boundary Violation	501	0	0	39
Deserialization of Untrusted Data	502	1	0	8
Weak Password Requirements	521	14	0	0
Inclusion of Sensitive Information in Log Files	532	2	0	0
Thread unsafe modification in singleton	543	0	0	19
Unsynchronized Access to Shared Data in a Multithreaded Context	567	0	0	5
Open Redirect	601	0	0	10
Authorization Bypass Through User-Controlled Key	639	8	0	0
Exposure of Resource to wrong sphere	668	0	0	4
Uncontrolled Recursion	674	0	0	2
Unrestricted Document Type Definition	827	0	0	1
Total		**55**	**4178**	**260**

Figure 6.2 Results of all detection techniques by vulnerability types



These results suggest that if one's goal is vulnerability type coverage or detecting design flaws, systematic penetration testing is the more effective choice. If one's goal is raw vulnerability count or finding input validation relevant vulnerability types, then automated penetration testing is the most effective choice.

6.1.2 Penetration Testing vs. Automated Static Analysis

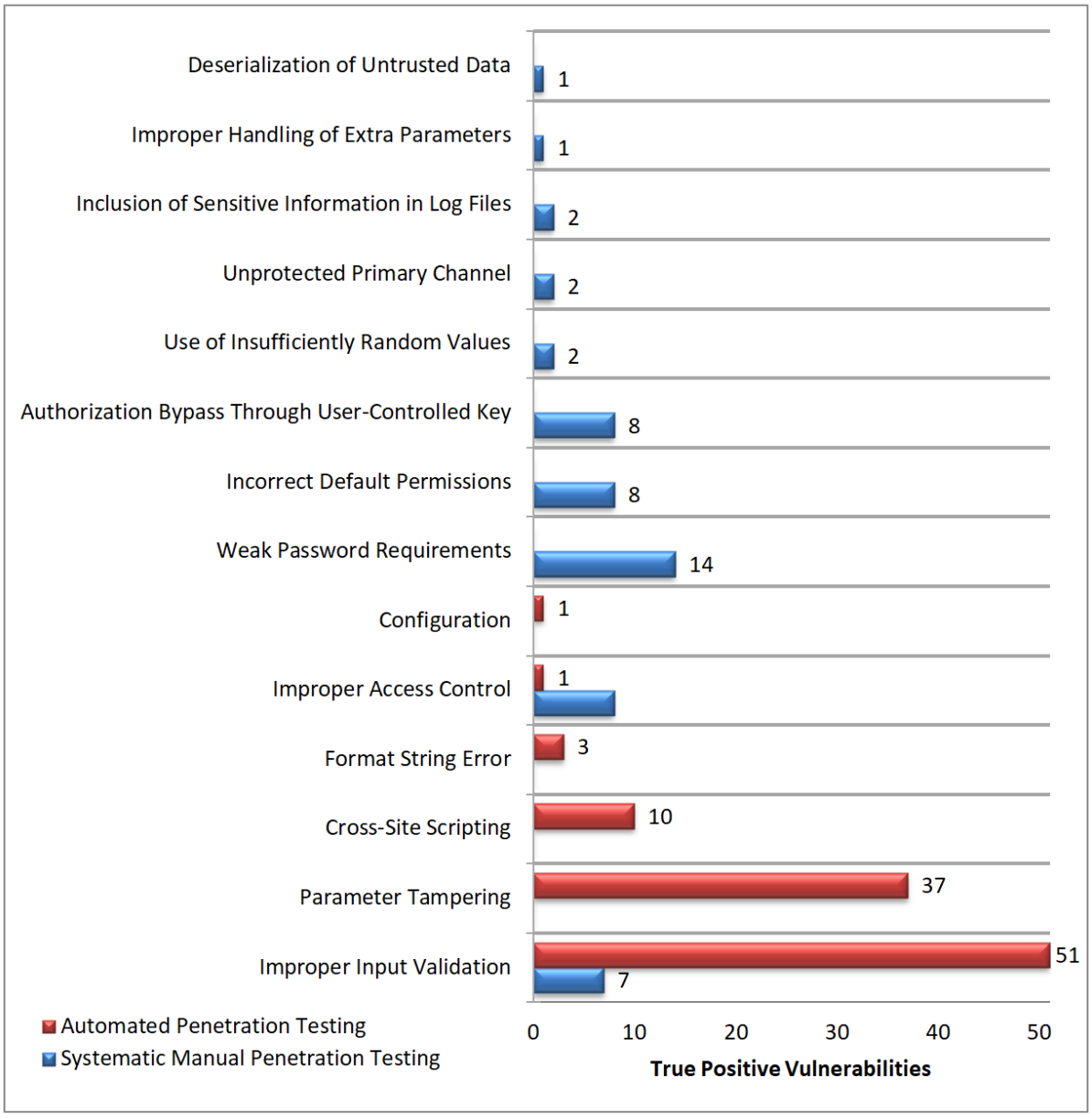
This section compares the results of automated static analysis to automated penetration testing and systematic manual penetration testing. Fig. 6.4 displays the detected vulnerabilities for the comparison with automated penetration testing, Fig. 6.5 with systematic manual. Vulnerability type information exposure was excluded from the later for easier viewing. Automated penetration detected 4075 and static analysis detected 0.

Automated tools can only detect errors in code by examining code segments or parsing output. Manual testing can detect functionality that is not present, enabling the detection of vulnerabilities neither automated static analysis or automated penetration testing could detect. One example is test case 2.3.1-1 (available in Appendix A.2.1), "the application should employ a method to generate secure, initial passwords...". The test case failed because the application does not provide password generation services. Since the application does not contain the functionality in code, an automates tool would not be able to report whether the functionality was secure or not, causing the detected vulnerability to be a false negative in both automated static analysis and automated penetration testing. The following subsections further discuss the differences between the individual penetration techniques and static analysis.

6.1.2.1 Systematic Manual Penetration

Static analysis and systematic manual penetration testing detected vulnerability types only directly overlapped with type deserialization of untrusted data. Both were also able to detect vulnerability types involving input validation, but the static analysis tool grouped the vulnerabilities by type (SQL

Figure 6.3 Comparison of vulnerability types detected by automated and systematic manual penetration testing



injection, cross-site scripting, code injection) whereas all the vulnerability types relevant to input validation for systematic manual penetration were grouped into type improper input validation due to the attack strings being stored instead of executed. The static analysis tool detected more vulnerabilities for both types.

Systematic manual penetration testing was able to discover vulnerability types static analysis could not regarding logging, password requirements, access control, and information disclosure - types that require interaction with a running application to detect that static analysis will not be able to find. Likewise, static analysis was able to discover vulnerability types unsafe reflection, uncontrolled resource consumption, cryptography, and calls to dangerous functions - vulnerabilities that would be difficult to find with manual penetration testing without rigorously examining each line of code.

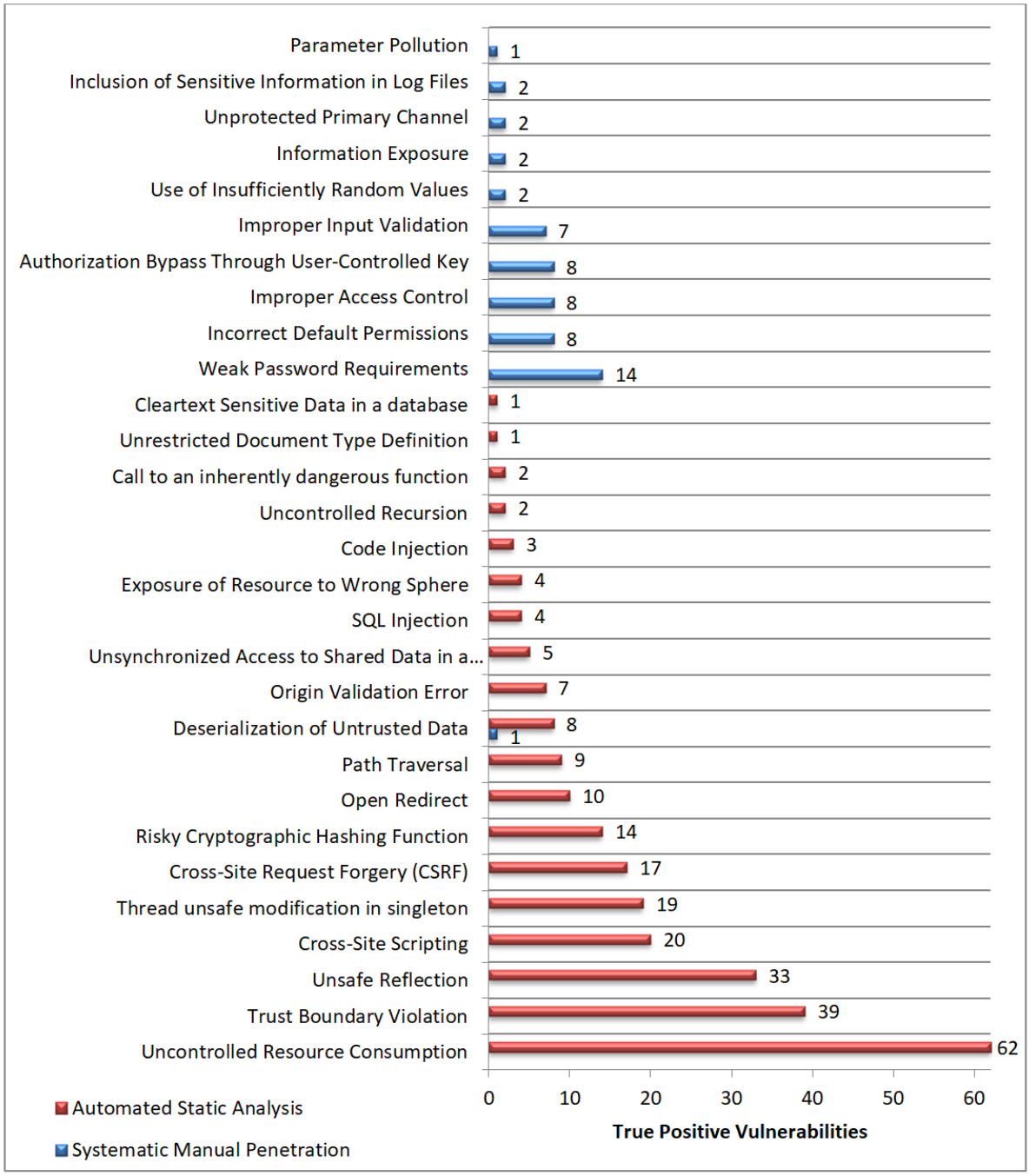
Static analysis performed better on raw vulnerability count (260) and type variety (19) over 55 and 11, respectively, for systematic manual penetration testing. Out of the 19 types detected by the static analysis tool, 6 were design flaws and 13 were implementation bugs. Out of the 11 types detected by systematic manual penetration, 7 were design flaws and 4 were implementation bugs.

These results suggest static analysis is more effect at raw vulnerability count and detecting a larger variety of vulnerability types. Systematic manual penetration testing was more effective at detecting design flaws, however, static analysis only found one less. Unfortunately, there was little overlap between the two techniques as far as vulnerability types. Therefore, if only one technique were to be used, many vulnerability types would go undetected.

6.1.2.2 Automated Penetration Testing

Automated static analysis and the automated penetration testing only overlapped in detected vulnerability types with type cross-site scripting. The automated penetration tool was able to detect vulnerability types: parameter tampering, insufficient verification of data authenticity, improper access control, improper configuration, and input validation vulnerabilities static analysis did not

Figure 6.4 Comparison of vulnerability types detected by systematic manual penetration testing and static analysis



detect (format string errors, generic lack of input validation). Parameter tampering, access control, configuration, and input validation all required interaction with a running application to detect, therefore static analysis would not be able to detect them. Likewise, the static analysis tool was able to discover vulnerabilities that would be difficult to discover with penetration testing, such as uncontrolled resource consumption, calls to dangerous functions, uncontrolled recursion, and the use of singleton code patterns in a multi-threaded environment.

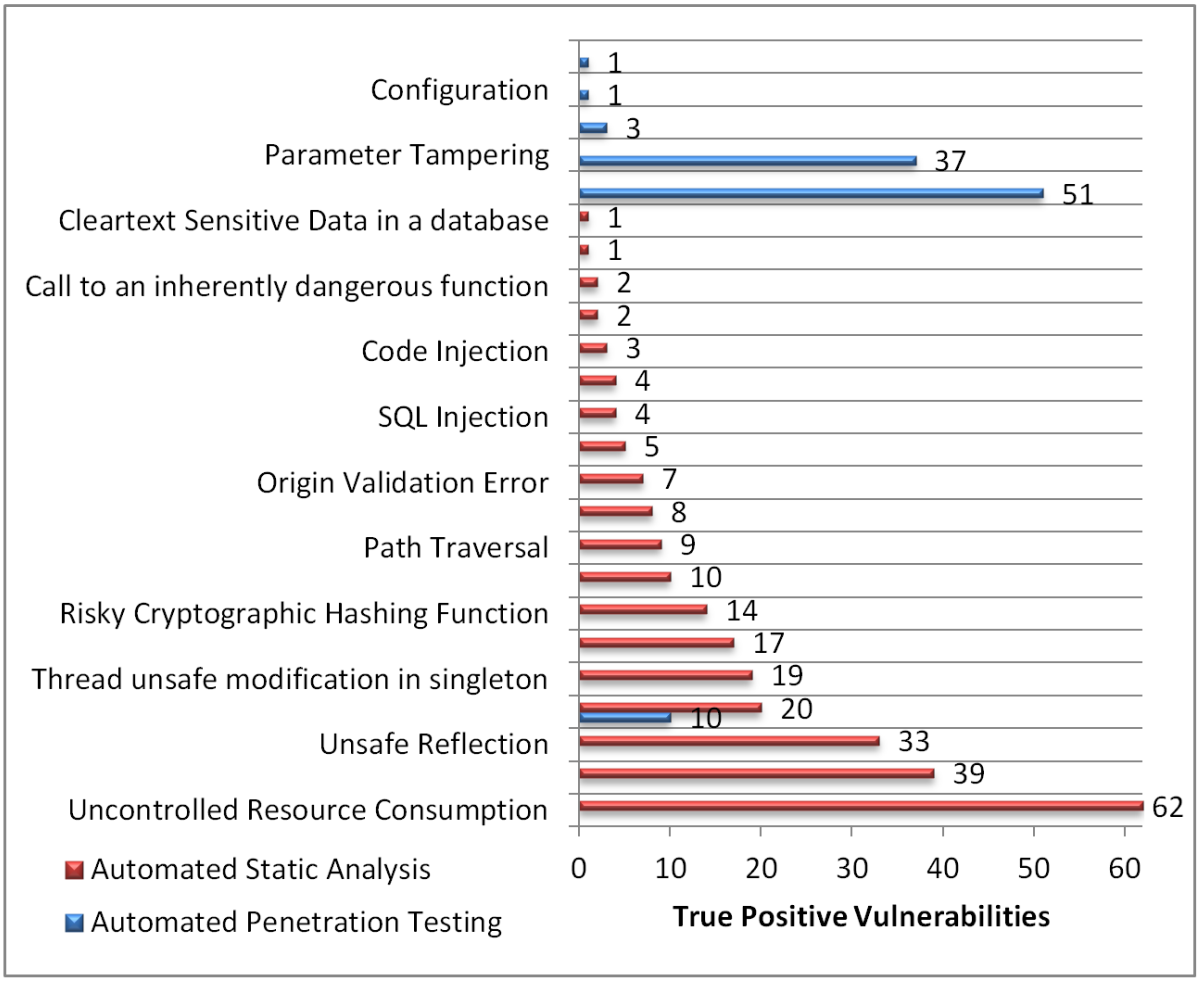
The automated penetration tool detected 4178 total vulnerabilities. Most of the vulnerabilities detected by the automated penetration tool were of vulnerability type information disclosure. Ignoring type information disclosure, the automated penetration tool only discovered 103 vulnerabilities. In total, the 8 detected types consisted of 6 implementation bugs and 2 design flaws. The static analysis tool discovered 260 vulnerabilities over 19 types: 13 implementation bugs and 6 design flaws.

Automated penetration testing detected the most vulnerabilities, but 97.5% of the vulnerabilities were instances of type information disclosure. If the information disclosure vulnerability type was ignored or counted in a way that prevented over-counting (discussed in Section 6.3.2), static analysis would be the better choice for raw vulnerability count. Static analysis also found more unique vulnerability types than automated penetration testing, offering a wider vulnerability type coverage. Both performed well in detecting input validation relevant vulnerability types, however, the types they detected only overlapped for cross-site scripting. Therefore, if performing only automated static analysis or only automated penetration testing would be insufficient as many vulnerabilities would go undetected.

6.2 Vulnerabilities Per Hour

Another metric to compare the vulnerability detection techniques is by efficiency. To measure efficiency, we calculated the time required to detect true positive vulnerabilities with each technique

Figure 6.5 Comparison of vulnerability types detected by automated penetration testing and static analysis



in terms of vulnerabilities per hour. The majority of the time required to perform systematic manual penetration testing was spent creating the test plan instead of testing the application. In automated penetration testing and automated static analysis, only the time needed to validate the reported potential vulnerabilities as true or false positive was recorded. Additional time needed to research vulnerabilities or vulnerability mitigation was not included. Time spent creating scripts used to organize or filter the results were also not included. Table 6.3 displays the efficiency metrics for all the detection techniques.

Static analysis was more efficient than systematic manual penetration testing. Automated penetration testing detected vulnerabilities the fastest. Most of the automated penetration vulnerabilities (4075/4178 or 97.5%) were information disclosure. Vulnerabilities per hour cannot be calculated without information disclosure in automated penetration testing because the time taken to validate all information disclosure type potential vulnerabilities individually was not recorded.

Static analysis had a lower false positive rate than automated penetration testing (discussed more in section 6.3), but still performed slower than automated penetration due to three factors. One, automated penetration testing discovered a large amount of information disclosure vulnerabilities very quickly. Two, auditing the reported potential vulnerabilities for false positives in automated penetration testing was faster than static analysis because the results could be validated by executing attacks quickly, instead of tediously examining code. Three, the automated penetration results contained patterns that facilitated fast validation of both individual and groups of similar vulnerabilities.

Automated penetration testing is therefore the most efficient as a whole. Since the subsets of vulnerability types detected per technique were largely orthogonal to each other, automated penetration testing may not be the most efficient for every vulnerability type.

Table 6.3 Efficiency of each vulnerability detection technique

Detection Technique	Vulnerabilities Per Hour	False Positive Rate
Systematic Manual Penetration Testing	1.5	N/A
Automated Penetration Testing	144.9	19.5%
Automated Static Analysis	14.2	7.1%

6.3 Comparing Detection Technique Weaknesses

This subsection discusses technique weaknesses and the inability for any of the detection tools to find every type of vulnerability. Based on these results, conducting security testing with one technique may leave many types of vulnerabilities undetected.

6.3.1 Systematic Manual Penetration Testing

The inability for systematic manual penetration testing to detect vulnerabilities lies on the limitations of execution. As discussed in Section 7, one of the limitations of this thesis work is the lack of resources available for testing the entire ASVS guide over every functionality in the target application. Vulnerability types that were accounted for in the test cases were generally detected successfully.

6.3.2 Automated Penetration Testing

Automated penetration testing struggled with detecting design flaws, only detecting parameter tampering. The other design flaw, improper access control, was labeled by the auditor. Automated penetration also demonstrated high false positive rates within the implementation bugs. The automated penetration tool discovered these bugs via fuzzing. Although fuzzing has been an active area in research with numerous algorithm improvements [11, 23, 24], the results may not always be trustworthy. Klees et al [23] observed many researchers over report number of bugs due to reporting "crashes" instead of "distinct bugs". This is a problem with our own reporting, as the automated penetration tool did not allow for easy analysis of where in code or application functionality the

bugs were located. The error pages discovered for Vulnerability type information disclosure may be over-counted and not distinct. There seemed to be 3 or 4 error page patterns that were occurring at individual code segments being detected hundreds of times.

Another source of over reporting is the overlap within the vulnerability types defined by the automated penetration tool. For example, the parameter tampering vulnerabilities were true positives because they caused the application to crash and expose sensitive information, which was also the case for types application disclosure and format string error. Further organization of the potential vulnerabilities into distinct groupings by similar resulting "crash" or output as well as functionality or code location in the application may produce a more representative implementation bug count.

Klees et al also observed substantial variance of run-to-run performance. Automated penetration testing was only performed once on our target application because of the extensive time required for both the setup and scans. The tool may produce different results upon re-scanning, especially after proper configuration. A default-setting scan was utilized, however, configuration of the tool to the application would be beneficial to removing multiple-attacks on the same location, as explained in section 5.3.2, that were caused by the application mistaking a single functionality as multiple distinct URIs. This was often a consequence of the presence or absence of parameters, e.g. UUIDs. In the case of UUIDs, configuring the application to attack one example object UUID per functionality would find the same vulnerabilities without the over-counting problem. The negative to this is of course the time and system knowledge required to configure the entire application.

6.3.3 Static Analysis

The static analysis tool produced a low false positive rate: only 7.1%. False positives were generally reported when the developers deviated from a pattern. For example, the developers passed a password to authentication methods after adding the salt to the password string. The static analysis tool determined salt was missing because it could not recognize the concatenated string being sent into the functions. Similarly, 3 of the vulnerabilities of type unchecked origin of message events were

false positive because the message was being checked by a value developers implemented that the tool failed to detect.

Otherwise, the static analysis tool performed as expected, failing to detect vulnerability types that require interaction with the running application, e.g. server configuration, information disclosure on error pages, and authentication.

CHAPTER

7

LIMITATIONS

The tools we selected to represent static analysis and automated penetration testing may not be representative of other similar tools. We also only used one tool to measure each discovery technique. Other tools may report different types of vulnerabilities.

The domain of open source electronic medical record domain of healthcare we selected as a case study subject may not be representative of software applications as a whole. Therefore, our results may not generalize not other subjects or other domains.

Manually classifying vulnerabilities as either true or false positive and classifying vulnerability types as implementation bug or design flaw may have introduced human errors.

Systematic manual penetration testing can only detect vulnerabilities for which test cases are written. The ASVS Level 1 guide consists of more vulnerabilities than we created test cases for.

In addition, only critical application functionality was tested by the test cases. By increasing the amount of test cases both by extending the vulnerability types considered and the functionality being tested, we could have increased detection. A subset of ASVS and of application functionality was implemented over critical application functionality because applying the entire guide over an application as large as the target application is impractical for a few individuals in a short period of time.

CHAPTER

8

CONCLUSIONS

Comparisons of the conclusions discovered in this case study and the prior works for effectiveness and efficiency of the detection techniques are is Table 8.1 and Table 8.2, respectively. In this case study, we found systematic manual penetration testing discovered the most design flaw vulnerability types. Automated static analysis discovered the most implementation bug vulnerability types. Automated penetration testing detected the most vulnerabilities if you include all the application crashes that causes information disclosure as individual vulnerabilities, otherwise, static analysis detected the most vulnerabilities. Automated static analysis produced a very low false positive rate while maintaining the the largest type coverage of vulnerability types, supporting that static analysis tools have greatly improved in vulnerability detection.

Systematic manual penetration testing, automated penetration testing, and automated static

analysis all detected subsets of vulnerability types that were predominantly orthogonal to each other such that almost no individual vulnerabilities were detected by multiple detection techniques. If one were to rely on just one technique, many vulnerabilities would go undetected. Because of vulnerability over-counting in automated penetration testing, it is not clear which tool was the most efficient for all vulnerabilities: automated penetration testing or static analysis. Efficiency for a specific vulnerability type depends on which tool is sufficient in detection of that type.

Table 8.1 Comparison of current and prior work conclusions for effectiveness of vulnerability detection techniques

	Tolven eCHR	**OpenEMR**	**PatientOS**	**OpenMRS**
Case Study	Prior Work	Prior Work	Prior Work	Current Work
Implementation Bug Types	Automated Static Analysis			Automated Static Analysis
Design Flaw Types	Systematic Manual Penetration Testing			Systematic Manual Penetration Testing
Most Efficient Implementation Bugs	Automated Penetration Testing			Automated Penetration Testing (?)

Table 8.2 Comparison of current and prior work conclusions for efficiency of vulnerability detection techniques

	Tolven eCHR	**OpenEMR**	**PatientOS**	**OpenMRS**
Case Study	Prior Work	Prior Work	Prior Work	Current Work
Vulnerabilities / Hour Systematic Manual Penetration Testing	0.9	0.6	0.6	1.5
Vulnerabilities / Hour Automated Penetration Testing	22	71	N/A	144.9 (3.6)
Vulnerabilities / Hour Automated Static Analysis	2.8	32.4	11.15	14.2
False Positive Rate Automated Penetration Testing	15%	3%	N/A	19.5% (6.8%)
False Positive Rate Automated Static Analysis	98%	74%	92%	7.10%

CHAPTER

9

FUTURE WORK

Future work could add the analysis of exploratory manual penetration testing while taking care of the significant affects of background knowledge, experience, and system knowledge.[19, 21, 32, 41]

The results of additional scans of automated penetration using the same tool and configuration may vary substantially from run to run.[23] Comparing multiple runs may be a better representation of automated penetration testing.

Comparing the results of automated penetration testing and automated static analysis with different tools than the once utilized in this case study may produce different conclusions than the ones we found.

BIBLIOGRAPHY

[1] Alsaleh, M. et al. “Performance-Based Comparative Assessment of Open Source Web Vulnerability Scanners”. English. *Security and Communication Networks* **2017** (2017), pp. 1–14.

[2] Antunes, N. & Vieira, M. “Defending against Web Application Vulnerabilities”. English. *Computer* **45**.2 (2012), pp. 66–72.

[3] Antunes, N. & Vieira, M. “Penetration Testing for Web Services”. *Computer* **47**.2 (2014), pp. 30–36.

[4] Antunes, N. & Vieira, M. “On the Metrics for Benchmarking Vulnerability Detection Tools”. *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 505–516.

[5] Argaw, S. T. et al. “The state of research on cyberattacks against hospitals and available best practice recommendations: a scoping review”. English. *BMC Medical Informatics and Decision Making* **19** (2019).

[6] Austin, A. & Williams, L. “One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques”. *2011 International Symposium on Empirical Software Engineering and Measurement*. 2011, pp. 97–106.

[7] Austin, A. “Improving the Security of Electronic Health Record Systems”. MA thesis. North Carolina State University Computer Science, 2011.

[8] Austin, A. et al. “A comparison of the efficiency and effectiveness of vulnerability discovery techniques”. English. *Information and Software Technology* **55**.7 (2013), pp. 1279–1288.

[9] Biancotti, C. *Cyber attacks: preliminary evidence from the Bank of Italy’s business surveys*. Questioni di Economia e Finanza (Occasional Papers) 373. Bank of Italy, Economic Research and International Relations Area, 2017.

[10] Chen, C. et al. “Penetration Testing in the IoT Age”. *Computer* **51**.4 (2018), pp. 82–85.

[11] Chen, C. et al. “A systematic review of fuzzing techniques”. *Computers Security* **75** (2018), pp. 118 –137.

[12] Cho, J.-H. et al. “STRAM: Measuring the Trustworthiness of Computer-Based Systems”. English. *ACM Computing Surveys (CSUR)* **51**.6 (2019), pp. 1–47.

[13] Choi, S. J. et al. “Data breach remediation efforts and their implications for hospital quality”. *Health Services Research* **54**.5 (2019), pp. 971–980. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/1475-6773.13203`.

[14] *Common Weakness Enumeration (CWE)*. 2018.

[15] *Computer Security Resource Center (CSRC)*. 2019.

[16] Coventry, L. & Branley, D. “Cybersecurity in healthcare: A narrative review of trends, threats and ways forward”. *Maturitas* **113** (2018), pp. 48 –52.

[17] *Electronic Health Records*. 2012.

[18] Farhadi, M. et al. “Static Analysis of HIPPA Security Requirements in Electronic Health Record Applications”. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 02. 2018, pp. 474–479.

[19] Gebizli, C. S. & Sozer, H. “Impact of Education and Experience Level on the Effectiveness of Exploratory Testing: An Industrial Case Study”. English. IEEE, 2017, pp. 23–28.

[20] Hemphill, T. A. & Longstreet, P. “Financial data breaches in the U.S. retail economy: Restoring confidence in information technology security standards”. English. *Technology in Society* **44** (2016), pp. 30–38.

[21] Itkonen, J. et al. “The Role of the Tester’s Knowledge in Exploratory Software Testing”. English. *IEEE Transactions on Software Engineering* **39**.5 (2013), pp. 707–724.

[22] Kim, S. et al. “Software Vulnerability Detection Methodology Combined with Static and Dynamic Analysis”. English. *Wireless Personal Communications* **89**.3 (2016), pp. 777–793.

[23] Klees, G. et al. “Evaluating Fuzz Testing”. English. ACM, 2018, pp. 2123–2138.

[24] Liang, H. et al. “Fuzzing: State of the Art”. English. *IEEE Transactions on Reliability* **67**.3 (2018), pp. 1199–1218.

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

[26] Morrison, P. J. et al. “Are vulnerabilities discovered and resolved like other defects?” English. *Empirical Software Engineering* **23**.3 (2018), pp. 1383–1421.

[27] Muñoz, F. R. et al. “Analyzing the traffic of penetration testing tools with an IDS”. English. *The Journal of Supercomputing* **74**.12 (2018), pp. 6454–6469.

[28] Nunes, P. et al. “Benchmarking Static Analysis Tools for Web Security”. *IEEE Transactions on Reliability* **67**.3 (2018), pp. 1159–1175.

[29] *Open Web Application Security Project (OWASP)*. 2019.

[30] (OWASP), O. W. A. S. P. *Top 10 2017*. Tech. rep. 2017.

[31] *OWASP ZAP Wiki*. 2016.

[32] Pfahl, D. et al. “How is exploratory testing used? A state-of-the-practice survey”. English. ACM, 2014, pp. 1–10.

[33] Rezaeibagha, F. et al. “A systematic literature review on security and privacy of electronic health record systems: technical perspectives.” *Health Information Management Journal* **44**.3 (2015), pp. 23 –38.

[34] Seng, L. K. et al. “The approaches to quantify web application security scanners quality: a review”. English. *International Journal of Advanced Computer Research* **8**.38 (2018), pp. 285–312.

[35] Smith, B. & Williams, L. “On the Effective Use of Security Test Patterns”. *2012 IEEE Sixth International Conference on Software Security and Reliability*. 2012, pp. 108–117.

[36] Smith, B. & Williams, L. “Systematizing security test case planning using functional requirements phrases”. English. ICSE ’11. ACM, May 21, 2011, pp. 1136–1137.

[37] Spanos, G. & Angelis, L. “The impact of information security events to the stock market: A systematic literature review”. English. *Computers Security* **58** (2016), pp. 216–229.

[38] *Survey Shows Industrial Cybersecurity Breaches Lead to Financial Losses*. English. 2019.

[39] *Symantec Internet Secuirty Threat Report (ISTR) 2019*. Tech. rep. 2019.

[40] *The IEEE Center for Secure Design Showcases the Top 10 Most Significant Software Security Design Flaws*. English. 2014.

[41] Votipka, D. et al. “Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes”. English. IEEE, 2018, pp. 374–391.

[42] Williams, L. et al. “Engineering Security Vulnerability Prevention, Detection, and Response”. English. *IEEE Software* **35**.5 (2018), pp. 76–80.

[43] YARAGHI, N. & GOPAL, R. D. “The Role of HIPAA Omnibus Rules in Reducing the Frequency of Medical Data Breaches: Insights From an Empirical Study: The Role of HIPAA in Reducing Medical Data Breaches”. English. *The Milbank Quarterly* **96**.1 (2018), pp. 144–166.

APPENDIX

APPENDIX

A

SYSTEMATIC MANUAL PENETRATION

A.1 Test Patterns

Section 2 (Authentication) Test Pattern

ASVS Sources: 2.1, 2.2, 2.3, 2.5

No test pattern was used for Section 2.

Section 4 (Access Control) Test Pattern

4 - # <action> <functionality in OpenMRS>

ASVS Sources: 4.1.3, 4.1.4, 4.1.5, 4.2.1, 4.2.2, 4.3.1, 13.1.2

Assumptions: User account Grace has been created with Full privilege level and no capabilities.

Procedure:

1. Authenticate as admin.
2. Open the user interface for < functionality in OpenMRS >.
3. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
4. Logout as admin.
5. Repeat step 2, either by entering the stored URL, or by repeating the interface actions.
6. Authenticate as Grace.
7. Repeat step 2, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. **When applicable**: administrative interfaces use appropriate multi-factor authentication to prevent unauthorized use.
2. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

Section 5 (Validation, Sanitization and Encoding Verification Requirements) Test Pattern

5 - # <action> <functionality in OpenMRS>

ASVS Sources: 5.1.1, 5.1.3, 5.1.4, 5.1.5, 5.2.2, 5.2.4, 5.3.3,5.3.4, 5.3.5, , 5.3.7, 5.3.8, 5.3.10, 5.5.1, 5.5.2, 5.5.3, 5.5.4, 7.4.1, 13.2.2

Input validation attack list at: https://github.com/fuzzdb-project/fuzzdb/blob/master/attack/all-attacks/all-attacks-unix.txt

Assumptions:

Procedure:

1. Authenticate as admin.
2. Open the user interface for < target application functionality >.
3. Using a proxy, < attack > < Action in functionality >.
4. Repeat step 3 for < list of attacks >.

Expected Results:

1. **When applicable**: the target application is protected against HTTP parameter pollution attacks such that < parameters are not accepted or combined >.
2. **When applicable**: structured data is strongly typed and validated against a defined schema including allowed characters, length and pattern (e.g. credit card numbers or telephone, or validating that two related parameters are reasonable, such as checking that suburb and zip/postcode match). The target application should notify the user that < structured data does not match >
3. **When applicable**: URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.
4. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
5. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

Section 7.1 (Log Content) Test Pattern

7.1 - # <action> <functionality in OpenMRS>

ASVS Sources: 3.1.1, 7.1.1, 7.1.2

Assumptions: A doctor, Dr. John Smith, has been created in the target application.

Procedure:

1. Authenticate as admin.
2. Open the user interface for < target application functionality >.
3. < Action in functionality that will cause a log to be created >.
4. Open the < action > records for today's date.

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log sensitive data as defined under local privacy laws or relevant security policy.

A.2 Executed Test Cases

A.2.1 Authentication (Section 2)

Section 2 Resources and Common Function Instructions

How to create a User Account

1. Login to localhost:/openmrs.login.html as username = "admin" and password = "Admin123". You may chose any location if prompted.
2. Click "System Administration"
3. Click "Manage Accounts"
4. Click "Add New Account" (top left)
5. Fill in a "Family Name" (last name), a "Given Name" (first name), and check "Male" or "Female".
6. Check "Add User Account?" and fill in a username.
7. Select "Full" from the drop-down titled "Privilege Level"
8. Type in a password and confirm password (see documentation for default password requirements).
9. Select "Force Password Change" as needed by test case.
10. Select "Capabilities" as needed by test case

Password Requirements

By default, OpenMRS Reference Application 1.8 requires:

1. Password cannot match username
2. Password minimum length = 8
3. Password requires at least one digit
4. Password requires at least one non-digit
5. Password requires at least one upper case and one lower case non-digit characters.

How to nagivate to the "Change Password" page as a user.

1. Make sure you are logged out of OpenMRS
2. Navigate back to the login page (localhost:8080/openmrs/login.htm)
3. Create a new user account or login with a username and password for a user account that you have already created.
4. From the home page (localhost;8080/openmrs/referenceapplication/home.page) click on your username, then click on "My Account"
5. Click on "Change Password"

2.1 Password Security Requirements Test Cases

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Min 12 Characters"
1	1	2.1.1-1	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account up until step 7 (creating a password)
2 Type in a password that is less than 12 characters and follows all other password requirements, such as "123456789aB".
3 Check "Force Password" change.
4 Try to submit password and create user account.
5 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should notify the administrator that passwords must be at least 12 characters long.
OpenMRS should not accept any password less than 12 characters long, even if it follows all other password requirements.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Min 12 Characters"
2	1	2.1.1-2	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account up until step 7 (creating a password)
2 Type in a password that is exactly 12 characters and follows all other password requirements, such as "123456789aBc"
3 Check "Force Password" change.
4 Try to submit password and create user account.
5 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the password that is both 12 characters and follows all other password requirements.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Min 12 Characters"
3	1	2.1.1-3	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Cases (2.1.1-1 / 2.1.1-2).
2 Ensure you are logged out of OpenMRS (navigate to localhost:8080/openmrs/appui/header/logout.action if needed)
3 Navigate to localhost:8080/openmrs/login.htm and login as the new user account you created.
4 When Prompted for a password change, type in a password that is less than 12 characters and follows all other password requirements, such as "123456789aB".
5 Try to submit password and create user account.
6 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should notify the user that passwords must be at least 12 characters long. OpenMRS should not accept any password less than 12 characters long, even if it follows all other password requirements.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Min 12 Characters"
4	1	2.1.1-4	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Cases (2.1.1-1 / 2.1.1-2).
2 Ensure you are logged out of OpenMRS (navigate to localhost:8080/openmrs/appui/header/logout.action if needed)
3 Navigate to localhost:8080/openmrs/login.htm and login as the new user account you created.
4 When Prompted for a password change, type in a password that is exactly 12 characters and follows all other password requirements, such as "123456789aBc"
5 Try to submit password and create user account.
6 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the password that is both 12 characters and follows all other password requirements.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Allow 64 Characters"
5	2	2.1.2-1	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account up until step 7 (creating a password)
2 Type in a password that is exactly 64 characters and follows all other password requirements, such as "aBcd1234567890123456789012345678901234567890123456789012345678901234567890".
3 Check "Force Password" change.
4 Try to submit password and create user account.
5 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the 64 character password.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Allow 64 Characters"
6	2	2.1.2-2	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.2-1
2 Ensure you are logged out of OpenMRS (navigate to localhost:8080/openmrs/appui/header/logout.action if needed)
3 Navigate to localhost:8080/openmrs/login.htm and login as the new user account you created.
4 When Prompted for a password change, Type in a password that is exactly 64 characters and follows all other password requirements, such as aBCD1234567890123456789012345678901234567890123456789012345678901234567890.
5 Try to submit password and create user account.
6 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the 64 character password.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Allow spaces; no truncation"
7	3	2.1.3-1	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account up until step 7 (creating a password)
2 Type in a password that contains a space in the middle of the password that also meets all other password requirements, such as "aB 45678"
3 Check "Force Password" change.
4 Try to submit password and create user account.
5 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the password with the space.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Allow spaces; no truncation"
8	3	2.1.3-2	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.3-1.
2 Ensure you are logged out of OpenMRS (navigate to localhost:8080/openmrs/appui/header/logout.action if needed)
3 Navigate to localhost:8080/openmrs/login.htm and login as the new user account you created.
4 When Prompted for a password change, type in a password that contains a space in the middle of the password that also meets all other password requirements, such as "Ba 45678"
5 Try to submit password and create user account.
6 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the password with the space.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Allow spaces; no truncation"
9	3	2.1.3-3	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account up until step 7 (creating a password)
2 Type in a password that contains a space in the end of the password that also meets all other password requirements, such as "aB34567 "
3 Check "Force Password" change.
4 Try to submit password and create user account.
5 Signout of admin account and login as the new user with the password that contains the space at the end.

Expected Results

1 OpenMRS should accept the password with the space without truncation.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"Allow spaces; no truncation"
10	3	2.1.3-4	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.3-3.
2 Ensure you are logged out of OpenMRS (navigate to localhost:8080/openmrs/appui/header/logout.action if needed)
3 Navigate to localhost:8080/openmrs/login.htm and login as the new user account you created.
4 When Prompted for a password change, type in a password that contains a space in the end of the password that also meets all other password requirements, such as "bA34567 "
5 Try to submit password and create user account.
6 Signout and then try to log back in the same user account with the newly created password that contains a space at the end.

Expected Results

1 OpenMRS should accept the password with the space without truncation.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"single Unicode code point is permitted and considered 1 character"
11	4	2.1.4-1	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account up until step 7 (creating a password)
2 Type in a password that meets all password requirements and is 64 kanji characters (example given by ASVS), such as
" ああああいうえおかきけこさあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさ "
3 Check "Force Password" change.
4 Try to submit password and create user account.
5 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the unicode password as one unicode code point = one character

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"single Unicode code point is permitted and considered 1 character"
12	4	2.1.4-2	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.4-1.
2 Ensure you are logged out of OpenMRS (navigate to localhost:8080/openmrs/appui/header/logout.action if needed)
3 Navigate to localhost:8080/openmrs/login.htm and login as the new user account you created.
4 When Prompted for a password change, type in a password that meets all password requirements and is 64 kanji characters (example given by ASVS), such as
" いいいいあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさあいうえおかきけこさ "
5 Try to submit password and create user account.

6 Signout and then try to log back in the account with the newly created password

Expected Results

1 OpenMRS should accept the unicode password as one unicode code point = one character

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"User can change password" and "Old Password is Required"
13	5 and 6	2.1.5-1	620	5.1.1.2	

Repeatable Steps:

1 Make sure you are logged out of OpenMRS
2 Navigate back to the login page (localhost:8080/openmrs/login.htm)
3 Create a new user account or login with a username and password for a user account that you have already created.
4 From the home page (localhost;8080/openmrs/referenceapplication/home.page) click on your username, then click on "My Account"
5 Click on "Change Password"
6 Complete the form. Note whether or not the application asks the user for their current password. Create a new password that meets all the default password requirements.
7 Signout and try to log back in as that user with the old password.
8 Signout and try to log back in as that user with the new password.

Expected Results

1 Openmrs should require the current password of a user when the user is changing their password.
2 OpenMRS should require the new password when allowing a user to change their password.
3 OpenMRS should allow the user to change their password, and should only accept that new password for logging in as that user.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"set a new non-breached password."
14	7	2.1.7-1	521	5.1.1.2	

Repeatable Steps:

1 Follow the instructions to create a new user account up until step 7 (creating a password).
2 Use a password that has been breached/leaked that follows all other OpenMRS password requirements.
Ensure "Force Password Change" is checked (step 8). Breached Password Example: "Password1"
2 Try to submit user and password and create new account.
3 If allowed to submit, logout and log back in as that user with the newly created password.

Note: Checked with https://haveibeenpwned.com/Passwords

Expected Results

1 OpenMRS should not accept the known breached/leaked password. User should not be created

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"set a new non-breached password."
15	7	2.1.7-2	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.7-1.
2 When Prompted for a password change, type in a password that has been breached/leaked that follows all other OpenMRS password requirements.
Ensure "Force Password Change" is checked. Breached Password Example: "A12345678"
3 Try to submit password and create user account.
4 If allowed to submit, logout and log back in as that user with the newly created password.

Note: Checked with https://haveibeenpwned.com/Passwords

Expected Results

1 OpenMRS should not accept the known breached/leaked password. User should not be created

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"set a new non-breached password."
16	7	2.1.7-3	521	5.1.1.2	

Repeatable Steps:

1 Follow the instructions to create a new user account or use an account you have already created (that is not admin).
2 Follow the instructions to navigate to the "Change Password" page.
3 Try to change the user's password with a password that has been breached/leaked that follows all other OpenMRS password requirements. Breached Password Example: "Qwerty12"
4 If allowed to submit, logout and log back in as that user.

Note: Checked with https://haveibeenpwned.com/Passwords

Expected Results

1 OpenMRS should not accept the known breached/leaked password. User should not be created

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"password strength meter"
17	8	2.1.8-1	521	5.1.1.2	

Repeatable Steps:

1 Follow the instructions to create a new user account up until step 7 (creating a password).
2 Use a password that follows all OpenMRS password requirements. Ensure "Force Password Change" is checked (step 8).
3 While typing in password, note whether or not a password strength meter is being used to give you feedback on the strength of your password.
4 Try to submit user and password and create new account.

Expected Results

1 OpenMRS should provide a password strength meter in some form to help users set a stronger password.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"password strength meter"
18	8	2.1.8-2	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.8-1.
2 When Prompted for a password change, type in a password that follows all OpenMRS password requirements.
3 While typing in password, note whether or not a password strength meter is being used to give you feedback on the strength of your password.
4 Try to submit password and create user account.

Expected Results

1 OpenMRS should provide a password strength meter in some form to help users set a stronger password.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"password strength meter"
19	8	2.1.8-3	521	5.1.1.2	

Repeatable Steps:

1 Follow the instructions to create a new user account or use an account you have already created (that is not admin).

2 Follow the instructions to navigate to the "Change Password" page.
3 While typing in a new password, note whether or not a password strength meter is being used to give you feedback on the strength of your password.
4 Try to submit password and create user account.

Expected Results

1 OpenMRS should provide a password strength meter in some form to help users set a stronger password.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"no password composition rules"
20	9	2.1.9-1	521	5.1.1.2	

Repeatable Steps:

1 Follow the instructions to create a new user account up until step 7 (creating a password).
2 Try a password with at least 12 characters that contains only lowercase alpha charcters, such as "aaajjjbbbuuu". Ensure "Force Password Change" is checked (step 8).
3 Try to submit user and password and create new account.
4 If that password is successful, repeat step 1 with a password with at least 12 characters that contains only uppercase alpha charcters, such as AAAJJJBBBUUU. Ensure "Force Password Change" is checked (step 8).
5 Try to submit user and password and create new account.
6 If that password is successful, repeat step 1 with a password with at least 12 characters that contains both lowercase and uppercase alpha charcters, such as AAAjjjbbbUUU. Ensure "Force Password Change" is checked (step 8).
7 Try to submit user and password and create new account.
8 If that password is successfull, repeat step 1 with password with at least 12 characters that contains only digits, such as "2864579513258745697". Ensure "Force Password Change" is checked (step 8).
9 Try to submit user and password and create new account.
10 If that password is successfull, repeat step 1 with the following password: `~!@#$%^&*()_+{}|\][:"';<>?/.,/*-+. . Ensure "Force Password Change" is checked (step 8).
11 Try to submit user and password and create new account.
12 If that password is successfull, repeat step 1 with password with at least 12 characters that contains uppercase alpha, lowercase alpha, and digits, such as aaJJ2864579513258745697. Ensure "Force Password Change" is checked (step 8).
13 Try to submit user and password and create new account.
14 If that password is successfull, repeat step 1 with password with at least 12 characters that contains uppercase alpha, lowercase alpha, digits, and all symbols, such as azJJ96~!@#$%^&*()_+{}|\][:"';<>?/.,/*-+. . Ensure "Force Password Change" is checked (step 8).
15 Try to submit user and password and create new account.

Expected Results

1 OpenMRS should not use any password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. The password should be accepted if the minimum character count is reached and if the password is not commonly-used, is not expected, and is not compromised.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"no password composition rules"
21	9	2.1.9-2	521	5.1.1.2	

Repeatable Steps:

1 Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.9-1
2 When prompted for a password change, try a password with at least 12 characters that contains only lowercase alpha charcters, such as "aaajjjbbbuuu".
3 Try to submit user and password and create new account.
4 If that password is successful, repeat step 1 with a password with at least 12 characters that contains only uppercase alpha charcters, such as "AAAJJJBBBUUU".
5 Try to submit user and password and create new account.
6 If that password is successful, repeat step 1 with a password with at least 12 characters that contains both lowercase and uppercase alpha charcters, such as "AAAjjjbbbUUU".
7 Try to submit user and password and create new account.
8 If that password is successfull, repeat step 1 with password with at least 12 characters that contains only digits, such as "2864579513258745697".
9 Try to submit user and password and create new account.
10 If that password is successfull, repeat step 1 with the following password: `~!@#$%^&*()_+{}|\][:"';<>?/.,/*-+.
11 Try to submit user and password and create new account.
12 If that password is successfull, repeat step 1 with password with at least 12 characters that contains uppercase alpha, lowercase alpha, and digits, such as "aaJJ2864579513258745697".
13 Try to submit user and password and create new account.
14 If that password is successfull, repeat step 1 with password with at least 12 characters that contains uppercase alpha, lowercase alpha, digits, and all symbols, such as azJJ96~!@#$%^&*()_+{}|\][:"';<>?/.,/*-+.
15 Try to submit user and password and create new account.

Expected Results

1 OpenMRS should not use any password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. The password should be accepted if the minimum character count is reached and if the password is not commonly-used, is not expected, and is not compromised.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"no password composition rules"
22	9	2.1.9-3	521	5.1.1.2	

Repeatable Steps:

1 Follow the instructions to create a new user account or use an account you have already created (that is not admin).
2 Follow the instructions to navigate to the "Change Password" page.
3 Try a password with at least 12 characters that contains only lowercase alpha charcters, such as "aaajjjbbbuuu".
4 Try to submit user and password and create new account.
5 If that password is successful, repeat step 1 with a password with at least 12 characters that contains only uppercase alpha charcters, such as "AAAJJJBBBUUU".
6 Try to submit user and password and create new account.
7 If that password is successful, repeat step 1 with a password with at least 12 characters that contains both lowercase and uppercase alpha charcters, such as "AAAjjjbbbUUU".
8 Try to submit user and password and create new account.
9 If that password is successfull, repeat step 1 with password with at least 12 characters that contains only digits, such as "2864579513258745697".
10 Try to submit user and password and create new account.
11 If that password is successfull, repeat step 1 with the following password: `~!@#$%^&*()_+{}|\][:"';<>?/.,/*-+. . Ensure "Force Password Change" is checked (step 8).
12 Try to submit user and password and create new account.
13 If that password is successfull, repeat step 1 with password with at least 12 characters that contains uppercase alpha, lowercase alpha, and digits, such as "aaJJ2864579513258745697".
14 Try to submit user and password and create new account.
15 If that password is successfull, repeat step 1 with password with at least 12 characters that contains uppercase alpha, lowercase alpha, digits, and all symbols, such as azJJ96~!@#$%^&*()_+{}|\][:"';<>?/.,/*-+.
16 Try to submit user and password and create new account.

Note: Checked ALL passwords with https://haveibeenpwned.com/Passwords

Expected Results

1 OpenMRS should not use any password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. The password should be accepted if the minimum character count is reached and if the password is not commonly-used, is not expected, and is not compromised.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"no periodic rotation or history requirements"
23	10	2.1.10-1	521	5.1.1.2	

Repeatable Steps:

1 Search the OpenMRS docs for credential rotation and password histroy requirements.
2 If no such rotation or requirements were found, navigate to the applications Global Properties module:
    1 Login as admin
    2 Click on "System Administration"
    3 Click on "Manage Global Properties"
3 Look for any properties that relate to credential rotation or password history requirements.

Expected Results

1 OpenMRS should not require passwords to be changed arbitrarily (e.g., periodically), unless there is evidence of compromise of the authenticator.
2 No periodic credential rotation or password history requirements should documented.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"paste, browser password helpers, and external password managers"
24	11	2.1.11-1	521	5.1.1.2	

Repeatable Steps:

1. Ensure you are using a browser that has the ability to manage passwords. E.g.: Firefox, Chrome.
2. Follow the instructions to create a new user account up until step 7 (creating a password). Check "Force Password Change" (Step 8).
3. Type in a password that meets all default password requirements into a text editor and "paste" into the OpenMRS password textbox.
4. If that is successful, submit password and create the user.
5. Note whether or not the browser prompts you to save your password to the browser.
6. If the browser prompts you to save your password, logout of the application. Try to log back in using the browser's password manager (browser should fill in newly created password for the same user).
7. If that is successful, repeat steps 1-2 with a password manager, such as bitwarden.
8. Type in a password that meets all the default requriements.
9. Submit password and create the user.
10. Note whether or not password manager was allowed to auto-save username and password.
11. If the password manager was successful in auto-saving username and password, logout of the application and log back in using the password manager auto-fill function.

Note: To use Bitwarden's auto-save and auto-fill functions, you will need to install the plugin/extension to your browwer of choice.
Note: sudo snap install bitwarden

Expected Results

1. OpenMRS should allow "paste" functionality for passwords. Password should be "pasted" correctly from text editor to application's password textbox.
2. OpenMRS should allow for the use of browser username and password managers. Auto-save and auto-fill of username and password via browser manager should be successful.
3. OpenMRS should allow for the use of password managers. Auto-save and auto-fill of username and password via password manager should be successful.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"paste, browser password helpers, and external password managers"
25	11	2.1.11-2	521	5.1.1.2	

Repeatable Steps:

1. Ensure you are using a browser that has the ability to manage passwords. E.g.: Firefox, Chrome.
2. Create a new user account with "Force Password Changed" checked (step 8) or re-use the account you just created in Test Case 2.1.11-1. Login with that new account.
3. When prompted to change passwords, type in a password that meets all default password requirements into a text editor and "paste" into the OpenMRS "New Password" textbox.
4. If that is successful, submit password.
5. Note whether or not the browser prompts you to save your password to the browser.
6. If the browser prompts you to save your password, logout of the application. Try to log back in using the browser's password manager (browser should fill in newly created password for the same user).
7. If that is successful, repeat steps 1-3 with a password manager, such as bitwarden.
8. Type in a password that meets all the default requriements.
9. Submit password.
10. Note whether or not password manager was allowed to auto-save username and password.
11. If the password manager was successful in auto-saving username and password, logout of the application and log back in using the password manager auto-fill function.

Note: To use Bitwarden's auto-save and auto-fill functions, you will need to install the plugin/extension to your browwer of choice.
Note: sudo snap install bitwarden

Expected Results

1. OpenMRS should allow "paste" functionality for passwords. Password should be "pasted" correctly from text editor to application's password textbox.
2. OpenMRS should allow for the use of browser username and password managers. Auto-save and auto-fill of username and password via browser manager should be successful.
3. OpenMRS should allow for the use of password managers. Auto-save and auto-fill of username and password via password manager should be successful.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	"paste, browser password helpers, and external password managers"
26	11	2.1.11-3	521	5.1.1.2	

Repeatable Steps:

1. Ensure you are using a browser that has the ability to manage passwords. E.g.: Firefox, Chrome.
2. Follow the instructions to create a new user account or use an account you have already created (that is not admin).
3. Follow the instructions to navigate to the "Change Password" page.
4. When prompted to change passwords, type in a password that meets all default password requirements into a text editor and "paste" into the OpenMRS "New Password" textbox.
5. If that is successful, submit password.
6. Note whether or not the browser prompts you to save your password to the browser.
7. If the browser prompts you to save your password, logout of the application. Try to log back in using the browser's password manager (browser should fill in newly created password for the same user).
8. If that is successful, repeat steps 1-3 with a password manager, such as bitwarden.
9. Type in a password that meets all the default requriements.
10. Submit password.
11. Note whether or not password manager was allowed to auto-save username and password.
12. If the password manager was successful in auto-saving username and password, logout of the application and log back in using the password manager auto-fill function.

Note: To use Bitwarden's auto-save and auto-fill functions, you will need to install the plugin/extension to your browwer of choice.
Note: sudo snap install bitwarden

Expected Results

1. OpenMRS should allow "paste" functionality for passwords. Password should be "pasted" correctly from text editor to application's password textbox.
2. OpenMRS should allow for the use of browser username and password managers. Auto-save and auto-fill of username and password via browser manager should be successful.
3. OpenMRS should allow for the use of password managers. Auto-save and auto-fill of username and password via password manager should be successful.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	" temporarily view password or view the last typed character of the password"
27	12	2.1.12-1	521	5.1.1.2	

Repeatable Steps:

1. Follow the instructions to create a new user account up until step 7 (creating a password). Check "Force Password Change" (Step 8).
2. Type in a password that meets all the default requirements, such as "aA12345678"
2. Note whether or not there exists a temporarily password viewer (often a button that can be clicked next to the password textbox) or if the last typed character of the password is viewable for a short period of time.
3. If the password could be viewed temporarily, submit and create the user account.
4. Logout of admin and into the newly created user account.
5. When prompted to change passwords, repeat step 2.
6. If the new password could be viewed temporarily, submit the password change.
7. Follow the instructions to navigate to the "Change Password" page.
8. Repeat step 2 with the "New Password" fields.

Expected Results

1. A user should be able to either temporarily view the entire masked password, or temporarily view the last typed character of the password when creating a password.
2. All three tests should be successful (a viewer does exist)

2.2 General Authenticator Requirements

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	Anti-Automation Controls
1	1	2.2.1-1	307	5.1.5.2, 5.1.4.2, 5.1.1.2, 5.2.2	

Repeatable Steps:

1	Download a free automated authentication tool like Hydra (https://tools.kali.org/password-attacks/hydra)
2	Use the tool to attempt to login to a user registered in OpenMRS over 100 times per hour. Use a wrong password everytime.
3	After the brute force attack, try to login as the user normally.

Expected Results

1	OpenMRS should stop the brute force attack by the use of soft lockouts, rate limiting, CAPTCHA, increasing delays between attemps, IP address restrictions, risk-based restrictions (such as location or first login on a device), and recent attemps to unlock the account.
2	OpenMRS should employ some mechanism to stop the user from logging in normally after a large amount of brute force attempts, or a mechanism to authenticate safely. (E.g.: rate limiting will stop the user from logging in.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	Weak Authenticators
2	2	2.2.2-1	304	5.2.10	

Repeatable Steps:

1	Follow the instructions to create a user or use a user that has already been created.
2	Login to OpenMRS as that user.

Expected Results

1	OpenMRS should not employ weak authenticators, such as SMS, email, and time based one-time passwords (OTPs), as primary login methods.
2	OpenMRS should employ a strong primary authentication method, such as loggin in with a strong password the user previously created.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	Secure Notifications
3	3	2.2.3-1	320	-	

Repeatable Steps:

1	Follow the instructions to create a user or use a user that has already been created.
2	Follow the instructions to navigate to the "Change Password" page and changed the user's password.

Expected Results

1	OpenMRS should send a secure notification to the user that their credentials were updated/changed.
2	The notification must be secure. For example, neither the old or new credentials should be displayed in plaintext within the notification.

Actual Results **Researcher:**

1

2.3 Authenticator Lifecycle Requirements

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T
1	1	2.3.1-1	330	5.1.1.2, Appendix 3 (A.3)

Randomly Generated Initial Passwords

Repeatable Steps:

1. Follow the instructions to create a new user account up until step 8 (creating a password).

2. Try to generate a random initial password for the user. If no tool exists, go to step 3. If a password could be generated, skip to step 4.
3. Type in a password for the new user that is less than 6 characters
4. Try to submit password and create user
5. If the password fails, create a password manually that follows all of the default OpenMRS requirements (see documentation).
6. Ensure that "Force Password Change" is NOT checked.
7. Submit password and create user.
8. Logout of the admin account and login as the newly created user.

Expected Results:

1. OpenMRS should employ a method to generate secure, initial passwords that are at least 6 characters long and that expire after a short period of time.
2. OpenMRS should force the user to eventually change passwords and to not let the user keep that inital password for the long term.

2.5 Credential Recovery Requirements

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	Password Hints
1	2	2.5.2-1	640	5.1.1.2	

Repeatable Steps:

1. Follow the instructions to create a new user account.
2. Logout of the admin account and into that newly created user account.

Expected Results

1. When a new user account is created in OpenMRS, at no point should "password hints" or "knowledge-based authetnication"/"secret questions" be present.
2. User is never asked to input password hints or give answers to questions with the intention to use as authentication later.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	Password Credential Recovery
2	3	2.5.3-1	640	5.1.1.2	

Repeatable Steps:

1. Follow the instructions to create a new user account or use an already created user.
2. Navigate to the login page (http://localhost:8080/openmrs/login.htm)
3. Try to recover your password by clicking on the "Can't log in?" link at the bottom right of the login box (underneath the login button).
4. Follow any instructions given by OpenMRS to recover your password.

Expected Results

1. OpenMRS should not reveal the current password in any way during password credential recovery.

Actual Results **Researcher:**

1

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T	Changed Authentication factor
4	5	2.5.5-1	304	6.1.2.3	

Repeatable Steps:

1. Complete Test Case unique ID 2.5.3-1
2. Follow the instructions to navigate to the "Change Password" page and change the user's password. (See documentation for default password requirements)

Expected Results

1. When a user changes or replaces an authentication factor (e.g. username, password, email...anything used to login) OpenMRS should notify that user of the change.

Actual Results **Researcher:**

1

A.2.2 Access Control (Section 4)

4-1 View Patient Data

Assumptions: User account Grace has been created with Full privilege level (have to select a level…) and no capabilities. Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient Rose Flower. Select patient to view their profile.
4. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
5. Logout as admin.
6. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.
7. Authenticate as Grace.
8. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-2 Edit Patient Demographic Information

Assumptions: User account Grace has been created with Full privilege level and no capabilities. Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient Rose Flower. Select patient to view their profile. Click Show Contact Info next to their name. Then, click the new Edit button to edit their contact information.
4. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
5. Logout as admin.
6. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.
7. Authenticate as Grace.
8. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-3 Edit Patient Contact Information

Assumptions: User account Grace has been created with Full privilege level and no capabilities. Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient Rose Flower. Select patient to view their profile. Click Edit next to their name to edit their demographic information.
4. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
5. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.
6. Authenticate as Grace.
7. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-4 Edit Patient Allergies

Assumptions: User account Grace has been created with Full privilege level and no capabilities. Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for managing allergies for patient Rose Flower.
3. Add new allergy:
    a. Food : Beef
    b. Reactions: Fever
    c. Severity: Mild
4. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface. Using a proxy, note/save the POST request created.
5. Logout as admin.
6. Repeat steps 2-3, either by sending the same POST request with the following parameters, or by repeating the interface actions.
    a. allergenType=FOOD
    b. otherCodedAllergen=NON_CODED:
    c. codedAllergen=162544 (which is chocolate in my database).
7. If the previous step was successful, authenticate as admin and delete the allergy from the patient's list of allergies. Logout as admin.
8. Authenticate as Grace.
9. Repeat steps 2-3, either by sending the same POST request with the following parameters, or by repeating the interface actions.
    a. allergenType=Food
    b. otherCodedAllergen=NON_CODED:
    c. codedAllergen=162544 (which is chocolate in my database).

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings. The POST request with the second allergen should not be saved to the database.

4-5 Patient Visit Management

Assumptions: User account Grace has been created with Full privilege level and no capabilities. Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for viewing patient information for patient Rose Flower.
3. Click “Start Visit” found under General Actions. Select Outpatient for Visit Type and confirm.
4. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
5. Logout as admin.
6. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.
7. Authenticate as Grace.
8. Repeat steps 2-3, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-6 View Active Visits

Assumptions: User account Grace has been created with Full privilege level and no capabilities. Patient Rose Flower has been created and was checked in for an Outpatient visit.

Procedure:

1. Authenticate as admin.
2. Open the user interface for viewing active patient visits. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
3. Logout as admin.
4. Repeat step 2, either by entering the stored URL, or by repeating the interface actions.
5. Authenticate as Grace.
6. Repeat step 2, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-7 Add Patient Vitals

Assumptions: User account Grace has been created with Full privilege level and no capabilities. Patient Rose Flower has been created and was checked in for an Outpatient visit.

Procedure:

1. Authenticate as admin.
2. Open the user interface for adding patient vitals. Select patient Rose Flower. Click “Yes, Record Vitals” to confirm.
3. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
4. Logout as admin.
5. Repeat step 2, either by entering the stored URL, or by repeating the interface actions.
6. Authenticate as Grace.
7. Repeat steps 2, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-8 Manage User Accounts

Assumptions: User account Grace has been created with Full privilege level and no capabilities. User account John Smith has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding a user account. Search for user John Smith. Click the pencil icon under Action to view their account information.
3. Under User Account Details, click Retire and type “123” for the reason.
4. Using a proxy, observe and record the DELETE request created by this action.
5. Restore the user by clicking “Restore”.
6. Logout as admin.
7. Try to repeat step 3, by sending the same DELETE request.
8. If the previous step was successful, authenticate as admin and restore the user. Then logout as admin again.
9. Try to repeat step 3, by sending the same DELETE request.

NOTE: You can actually authenticate with Postman. To the login page, POST username=username, password=password, sessionLocation=6. Then, send request

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.
2. Administrative interfaces use appropriate multi-factor authentication to prevent unauthorized use. Before the DELETE request is processed, the user is authenticated again.

4-L9 Legacy Manage User Accounts

Assumptions: User account Grace has been created with Full privilege level and no capabilities. User account John Smith has been created.

Procedure:

1. Authenticate as admin.
2. Open the Legacy user interface for finding a user account. Search for user John Smith. Click on the user to access their legacy account information.
3. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
4. Logout as admin.
5. Repeat step 2, either by entering the stored URL, or by repeating the interface actions.
6. Authenticate as Grace.
7. Repeat steps 2, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-L10 Legacy Manage Patient Accounts

Assumptions: User account Grace has been created with Full privilege level and no capabilities. Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the Legacy user interface for finding a patient account. Search for patient Rose Flower. Click on the patient to access their legacy account information.
3. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
4. Logout as admin.
5. Repeat step 2, either by entering the stored URL, or by repeating the interface actions.
6. Authenticate as Grace.
7. Repeat steps 2, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.

4-L10 Legacy Reporting

Assumptions: User account Grace has been created with Full privilege level and no capabilities. At least one patient has been created since yesterday.

Procedure:

1. Authenticate as admin.
2. Open the Legacy user interface for generating reports.
3. Select List of New Patient Registrations from the list of Available Reports.
4. In Start date, select yesterday's date. Click Request Report.
5. When the report is finished, View Report. It should contain at least one patient that was registered.
6. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
7. Logout as admin.
8. Repeat step 5, either by entering the stored URL, or by repeating the interface actions.
9. Authenticate as Grace.
10. Repeat step 5, either by entering the stored URL, or by repeating the interface actions.

Expected Results:

1. Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.
2. Administrative interfaces use appropriate multi-factor authentication to prevent unauthorized use. Before viewing the report with sensitive information, the authorized user should be authenticated again.

A.2.3 Validation, Sanitization, and Encoding (Section 5)

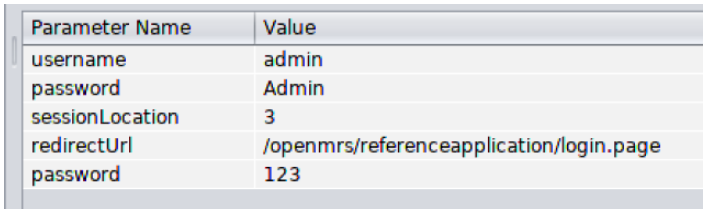
5-1 Login

Assumptions: None.

Procedure:

1. Open the user interface for login.
2. Using a proxy, try to login as admin with part of the admin's password in one "password" parameter and the rest of the password in a second "password" parameter. (See picture below)
3. For username, type in admin. Using a proxy, for the password parameter, enter an attack string from the injection attack list in the password parameter.
4. Repeat step 3 with four additional attack strings.
5. Using a proxy, enter an attack string from the URL redirect attack list.

Step 2 Diagram:



Parameter Name	Value
username	admin
password	Admin
sessionLocation	3
redirectUrl	/openmrs/referenceapplication/login.page
password	123

Expected Results:

1. The target application is protected against HTTP parameter pollution attacks such that the password is not concatenated during step 2 and the admin was not authenticated.
2. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
3. URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.
4. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-2 Find Patient

Assumptions: The Patient Rose Flower has been created. (See test cases for Section 7)

Procedure:

1. Authenticate as admin.

2. Open the user interface for finding a patient.
3. For the search field, enter an attack string from the injection attack list in the password parameter.
4. Repeat step 3 with four additional attack strings.

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-3 Edit Patient Demographic Information

Assumptions: The Patient Rose Flower has been created. The patient John Smith has been created. (See Test Cases for Section 7)

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient Rose Flower. Select patient to view their profile. Click Edit next to their name to edit their demographic information.
4. Copy/Save the patientId in the URL for Rose Flower.
5. Repeat steps 2-3 for patient John Smith.
6. Add Rose's patientId such that it comes before John's in the URL GET request. (See the example URL below)
7. Using a proxy, for the givenName parameter, enter an attack string from the injection attack list in the password parameter.
8. Repeat step 7with four additional attack strings.
9. Repeat steps 7-8 with parameters: gender, birthday, and birthdateEstimated.
10. Using a proxy, for the returnUrl header parameter, enter an attack string from the URL redirect attack list.

Example GET URL where 10 is Rose's Id and 18 is John's:
http://localhost:8080/openmrs/registrationapp/editSection.page?**patientId=10&patientId=18**&sectionId=demographics&appId=referenceapplication.registrationapp.registerPatient&returnUrl=%2Fopenmrs%2Fcoreapps%2Fclinicianfacing%2Fpatient.page%3FpatientId%3D28656a2d-0974-4c50-8376-2c1bde687396%26

Expected Results:

1. The target application is protected against HTTP parameter pollution attacks such that the application does not switch to Rose's or any other patient's account when multiple patientIds are present.
2. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
3. URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.
4. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-4 Edit Patient Contact Information

Assumptions: Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient Rose Flower. Select patient to view their profile. Click Show Contact Info next to their name. Then, click the new Edit button to edit their contact information.
4. Using a proxy, for the address parameter, enter an attack string from the injection attack list.
5. Repeat step 4 with four additional attack strings.
6. Repeat steps 4-5 with the cityVillage, postalCode, and phoneNumber parameters.
7. Using a proxy, enter:
    a. City: Raleigh
    b. State: NC
    c. Country: US
    d. Postal code: 66012

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. Structured data is strongly typed and validated against a defined schema including allowed characters, length and pattern (e.g. credit card numbers or telephone, or validating that two related parameters are reasonable, such as checking that suburb and zip/postcode match). The target application should notify the user that the postal code and city do not match.
3. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support

personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-5 Edit Patient Allergies

Assumptions: Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for managing the allergies of patient Rose Flower.
3. Click "Add" to add a new allergy.
4. Using a proxy, for the allergenType parameter, enter an attack string from the injection attack list.
5. Repeat step 4 with four additional attack strings.
6. Repeat steps 4-5 for the parameters: codedAllergen, comment, allergyReactionConcepts, and severity.

NOTE: To modify allergies, click on the pencil icon next to ALLERGIES

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-6 Add Patient Visit Note and Diagnosis

Assumptions: Patient Rose Flower has been created and has been checked into an Outpatient visit. The database contains demo diagnoses.

Procedure:

1. Authenticate as admin.
2. Open the user interface for viewing the patient information for patient Rose Flower.
3. Click "Visit Note" under Current Visit Actions to add a visit note and diagnosis.
4. For clinical note enter "comment1". Enter a primary diagnosis: Cough. Enter a secondary diagnosis: Acute Pain. Save encounter.
5. Click on the pencil icon next to the Encounter to edit it.

6. Using a proxy, save the encounter after changing the parameters to include w10 (the clinical note parameter) twice:
    a. &w10=comment3
    b. &w10=comment4
7. Using a proxy, for the w10 parameter, enter an attack string from the injection attack list.
8. Repeat step 4 with four additional attack strings.
9. Repeat steps 5-6 for the parameters: personId, createVisit, encounterDiagnoses, and w5.
10. Using a proxy, for the returnUrl parameter, enter an attack string from the URL redirect attack list.

Expected Results:

1. The target application is protected against HTTP parameter pollution attacks such that the clinical note parameter was not changed to "comment3", "comment4", or any combination.
2. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
3. URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.
4. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-7 Add Patient Vitals

Assumptions: Patient Rose Flower has been created and has been checked into an Outpatient visit.

Procedure:

1. Authenticate as admin.
2. Open the user interface for adding patient vitals. Select patient Rose Flower.
3. Using a proxy, save the encounter after changing the parameters to include w8 (the height parameter) twice:
    a. &w8=165
    b. &w8=160
4. Using a proxy, for the w8 parameter, enter an attack string from the injection attack list.
5. Repeat step 4 with four additional attack strings.
6. Repeat steps 4-5 for the parameters: createVisit, htmlFormId, w1, and w10.
7. Using a proxy, for the returnUrl parameter, enter an attack string from the URL redirect attack list.

NOTE: Height is the only required parameter.

Expected Results:

1. The target application is protected against HTTP parameter pollution attacks such that the height parameter was not set as “165”, “160”, or any combination.
2. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
3. URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.
4. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system’s environment).

5-8 Find User Account

Assumptions: None.

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding user accounts.
3. For the search field, enter an attack string from the injection attack list.
4. Repeat step 4 with four additional attack strings.

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system’s environment).

5-9 Retire User

Assumptions: User John Smith has already been created.

Procedure:

1. Authenticate as admin.
2. Open the user interface for finding user accounts.
3. Search for user John Smith. Click the pencil icon under Action to view their user account information.
4. Using a proxy, for the Retire reason text field, enter an attack string from the injection attack list.

5. Repeat step 4 with four additional attack strings.

NOTE: reason shows up in header of DELETE request.

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-10 Create User Account

Assumptions: None.

Procedure:

1. Authenticate as admin.
2. Open the user interface for managing user accounts. Click Add New Account.
3. Using a proxy, for the familyName parameter, enter an attack string from the injection attack list.
4. Repeat step 4 with four additional attack strings.
5. Repeat steps 3-4 with the following parameters: addUserAccount, capabilities, gender, privilegeLevel.

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-11 Manage Encounter Roles

Assumptions

1. An encounter role has already been created.

Procedure

1. Authenticate as the admin.
2. Open the user interface for managing encounter roles.

3. Click “Add New Encounter Role”.
4. Using a proxy, for the name parameter, enter an attack string from the injection attack list.
5. Repeat step 4 with four additional attack strings.
6. Repeat steps 4-5 with the description parameter.
7. Navigate back to the user interface for managing encounter roles.
8. Select an encounter role and click on the “X” action to delete it.
9. Repeat steps 4-5 for the reason parameter.

NOTE: Chose this to represent management APIs that are using JSON.

Expected Results

1. The target application is protected against the deserialization of untrusted data (such as JSON, XML and YAML parsers). The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system’s environment).

5-L12 Legacy Find User Accounts

Assumptions: Users Jake, Jane, John, and Julie Smith have been created.

Procedure:

1. Authenticate as admin.
2. Open the legacy user interface for finding a user account.
3. Search for “Smith”. There should be users Jake, Jane, John, and Julie Smith.
4. Using a proxy, send both name=Jane and name=John in the created GET request.
5. Using a proxy, for the name parameter, enter an attack string from the injection attack list.
6. Repeat step 5 with four additional attack strings.
7. Repeat steps 5-6 with the following parameters: includeDisabled, role, action.

NOTE: parameters in header as type url.

Expected Results:

1. The target application is protected against HTTP parameter pollution attacks such that the name parameter with values of “Jane”, “John”, nor any combination was accepted as a search criteria.
2. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.

3. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-L13 Legacy Create User Accounts

Assumptions: None.

Procedure:

1. Authenticate as admin.
2. Open the legacy user interface for managing user accounts. Click Add User. Click next.
3. Using a proxy, for the person.names[0].familyName parameter, enter an attack string from the injection attack list.
4. Repeat step 4 with four additional attack strings.
5. Repeat steps 3-4 with the following parameters: person.gender, providerCheckBox, roleStrings, and username.

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).

5-L15 Legacy Find Patient

Assumptions: None.

Procedure:

1. Authenticate as admin.
2. Open the legacy user interface for finding a patient.
3. Using a proxy, for the c0-param0=string: parameter, enter an attack string from the injection attack list in the password parameter. (This is the name search field)
4. Repeat step 3 with four additional attack strings.
5. Repeat steps 3-4 with the following parameters: c0-id, c0-param3=boolean:, callCount, c0-scriptName.
6. Using a proxy, for the page parameter, enter an attack string from the URL redirect attack list.

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).
3. URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.

5-L16 Legacy Edit Patient Data

Assumptions: Patient Rose Flower has been created.

Procedure:

1. Authenticate as admin.
2. Open the legacy user interface for managing patients. Search for patient Rose Flower. Select patient to view their legacy account profile.
3. Using a proxy, for the names[0].preferred parameter, enter an attack string from the injection attack list in the password parameter. (This is the name search field)
4. Repeat step 3 with four additional attack strings.
5. Repeat steps 3-4 with the following parameters: names[0]voidReason, deadDate, address[0].stateProvince, and names[0].middleName.
6. Using a proxy, POST the following parameters:
    a. addresses[0].cityVillage: Raleigh
    b. addresses[0].stateProvince: NC
    c. addresses[0].country: US
    d. addresses[0].postalCode: 66012
7. Using a proxy, POST the identifiers[0].identifier parameter as "abc".

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).
3. Structured data is strongly typed and validated against a defined schema including allowed characters, length and pattern (e.g. credit card numbers or telephone, or validating that two related parameters are reasonable, such as checking that suburb and

zip/postcode match). The target application should notify the user that the postal code and city do not match.

4. Structured data is strongly typed and validated against a defined schema including allowed characters, length and pattern (e.g. credit card numbers or telephone, or validating that two related parameters are reasonable, such as checking that suburb and zip/postcode match). The target application should notify the user that the identifier “abc” is not valid for the OpenMRS ID identifier type.

A.2.4 Error Handling and Logging Verification (Section 7)

7.1-1 Register a Patient

Assumptions: None.

Procedure:

1. Authenticate as the admin.
2. Open the user interface for registering a new patient.
3. Type in “Rose” for given name, “Flower” for family name, select “Female” for gender, provide estimated years of age “35”, and “555 Road, City, NC, US 27607” as the address. Confirm new patient.
4. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-2 View Patient Data

Assumptions: The patient Rose Flower has been created. (See 7.1.1 Register a Patient)

Procedure:

1. Authenticate as the admin.
2. Open the user interface for finding a patient.
3. In the search field, type in the patient name “Rose Flower”. Select patient to view their profile.
4. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-3 Edit Patient Data

Assumptions: The patient Rose Flower has been created. (See 7.1.1 Register a Patient)

Procedure:

1. Authenticate as the admin.
2. Open the user interface for finding a patient.
3. In the search field, type in the patient name “Rose Flower”. Select patient to view their profile.
4. Click the Edit button next to Show Contact Info to edit Rose’s name data.
5. Type “Thorn” for middle name and click Save Form, then click Confirm.
6. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-4 Patient Encounter Management

Assumptions: Patient Rose Flower has been created. (See 7.1.1 Register a Patient)

Procedure:

1. Authenticate as the admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient name “Rose Flower”. Select patient to view their profile.
4. Under General Actions (on the right), click Start Visit. Select Visit Type “Facility Visit” and click Confirm.
5. Click End Visit, then Yes to confirm.
6. Click on the visit (date will appear under the Visits tab).
7. Delete the visit (right side).
8. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-5 Modify Patient Conditions

Assumptions: Patient Rose Flower has been created. (See 7.1.1 Register a Patient)

Procedure:

1. Authenticate as the admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient name “Rose Flower”. Select patient to view their profile.
4. Click on the pencil icon under Conditions to add a condition to the patient.
5. Under Active Conditions, click Add New Condition.
6. In the text field, type in “fever”, then select the current date from the calendar selection on the right. Save.
7. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom). (Even if this is still broken and won’t save, it will produce a log)

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-6 Add Patient Visit Note and Diagnoses

Assumptions: Patient Rose Flower has already been created. (See 7.1.1 Register a Patient). A visit has been started for patient Rose Flower.

Procedure:

1. Authenticate as admin.
2. Open the user interface for viewing Active Visits.
3. Select patient Rose Flower.
4. Under Current Visit Actions, click Visit Note to add a visit note.
5. In the primary diagnosis field, type in “Cough” and select R05 Cough as a primary diagnosis.
6. In the diagnosis field, type in “fever” and select R50.9 Fever of unknown origin as a Secondary diagnosis.
7. In the Clinical Note text box, type in “Hello Add New Allergy Comment Box 123”.
8. Save the visit and diagnosis.
9. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-7 Modify Patient Allergy List

Assumptions: Patient Rose Flower has been created.

Procedure:

1. Authenticate as the admin.
2. Open the user interface for finding a patient.
3. In the search field, type in patient name Rose Flower. Select patient to view their profile.
4. Click the pencil symbol next to ALLERGIES.
5. Click Add New Allergy.
6. Select the FOOD tab.
7. Select Beef.
8. For Reactions, check Cough, Fever, and RASH.
9. For Severity, select Mild.
10. In the Comment text box, type in "Allergy comment box 123".
11. Save the new allergy.
12. Click the "X" symbol under actions on the newly created allergy to delete it.
13. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today's date. (Scroll to the bottom).

NOTE: This actually did not produce a log for me.

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-8 Adding Patient Vitals

Assumptions: Patient Rose Flower has already been created. A visit has been started for patient Rose Flower.

Procedure:

1. Authenticate as the admin.
2. Open the user interface for capturing the vitals of a currently visiting patient.
3. Select patient Rose Flower. Confirm that it is the correct patient.
4. Enter the following data:
    a. Height: 160 cm
    b. Weight: 50 kg
    c. Temperature: 25 degree C
    d. Pulse: 60 / min
    e. Respiratory rate: 90 / min
    f. Blood Pressure: 110 / 120
    g. Blood oxygen saturation: 85
5. Confirm and save entry.

6. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today's date. (Scroll to the bottom).

Expected Results:
1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-9 User Account Management

Assumptions: None.

Procedure:
1. Authenticate as admin.
2. Open the user interface for managing user accounts (May be under System Administration -> Manage Accounts)
3. Click Add New Account.
4. Create a user with the following information:
    a. Family Name: John
    b. Given Name: Smith
    c. Gender: Male
    d. Add User Account? Yes
    e. Username: jsmith
    f. Privilege Level: Full
    g. Password: Admin123
    h. Capabilities: Registers Patients
5. Save the new user.
6. Click the pencil icon under Action on the new user to view their user profile.
7. Under USER ACCOUNT DETAILS, click Retire. Enter "retire reason box 123" for the reason.
8. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today's date. (Scroll to the bottom).

Expected Results:
1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-L10 Legacy User Account Management

Assumptions: A user account for John Smith was created and retired.

Procedure:

1. Authenticate as admin.
2. Open the legacy user interface for managing user accounts (System Administration->Advanced Administration->Manage Users)
3. Check Include Disabled.
4. In the user search field, type in John. Click Search. Click on the John Smith that was retired (it will be crossed out).
5. Scroll to the bottom of the user account page and click Enable Account to restore the user.
6. Repeat steps 2 and 4 to re-open John Smith's user account. This time, the user will not be crossed out.
7. Save the user. (Scroll to the bottom of the account page).
8. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today's date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-L11 Legacy Register a Patient

Assumptions: None.

Procedure:

1. Authenticate as the admin.
2. Open the legacy user interface for managing patients. Click Create a New Patient.
3. Type in "Black Flower" for name, select "Male" for gender, provide estimated years of age "45", and click "Create Person"
4. Verify information previously entered, and enter "555 Road, City, NC, US 27607" as the address. For identifier, type in "123456", select Identifier Type "OpenMRS ID" and "Amani Hospital" for location. Save the new patient.
5. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today's date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-L12 Legacy View Patient Data

Assumptions: The patient Black Flower has been created. (See 7.1-L11 Legacy Register a Patient)

Procedure:

1. Authenticate as the admin.
2. Open the legacy user interface for finding patients.
3. In the search field, type in the patient name “Black Flower”. Select patient to view their profile.
4. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom).

NOTE: This did not generate a log for me.

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-L13 Legacy Edit Patient Data

Assumptions: The patient Black Flower has been created. (See 7.1-L11 Legacy Register a Patient)

Procedure:

1. Authenticate as the admin.
2. Open the legacy user interface for finding a patient.
3. In the search field, type in the patient name “Black Flower”. Select patient to view their user information.
4. Go to the “Demographics” tab and click “Edit this Patient”
5. Type “Thorn” for middle name and click Save Patient.
6. Scroll to the bottom of the page to delete the patient. For reason, type in “legacy hello Reason for Deleting Patient 123”.
7. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today’s date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

7.1-L14 Legacy Generate Reports

Assumptions: At least one new patient has been registered.

Procedure:

1. Authenticate as the admin.
2. Open the legacy user interface for generating a report.
3. From the list of Available Reports, click on List of New Patient Registrations.
4. Select yesterday as the start date and then click Request Report to generate the report.
5. Open the OpenMRS server logs at http://localhost:8080/openmrs/admin/maintenance/serverLog.form and find the records for today's date. (Scroll to the bottom).

Expected Results:

1. The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.
2. The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.

A.3 Test Case Results

A.3.1 Authentication (Section 2)

Unique ID	Expected Result #	Expected Result Description	Actual Results	Status (pass/fail)	Researcher	Notes
2.1.1-1	1	OpenMRS should notify the administrator that passwords must be at least 12 characters long. OpenMRS should not accept any password less than 12 characters long, even if it follows all other password requirements.	User account was successfully made.	fail	VIK	Reason for failure: OpenMRS provided a message upon entering a password indicating that it must be at least 8 characters long. Confirmed in Global Properties that the min length is 8.
2.1.1-2	1	OpenMRS should accept the password that is both 12 characters and follows all other password requirements.	User account was successfully made.	pass	VIK	
2.1.1-3	1	OpenMRS should notify the user that passwords must be at least 12 characters long. OpenMRS should not accept any password less than 12 characters long, even if it follows all other password requirements.	Successfully changed password.	fail	VIK	Reason for failure: Was able to change password to one that was less than 12 characters.
2.1.1-4	1	OpenMRS should accept the password that is both 12 characters and follows all other password requirements.	Successfully changed password.	pass	VIK	
2.1.2-1	1	OpenMRS should accept the 64 character password.	System accepted 64 character password	pass	VIK	
2.1.2-2	1	OpenMRS should accept the 64 character password.	System accepted 64 character password	pass	VIK	
2.1.3-1	1	OpenMRS should accept the password with the space.	System accepted password with a space character	pass	VIK	
2.1.3-2	1	OpenMRS should accept the password with the space.	System accepted password with a space character	pass	VIK	
2.1.3-3	1	OpenMRS should accept the password with the space without truncation.	System accepted the password without truncation	pass	VIK	Note: I found a strange error while doing this test case. If you are making a new user account, type in both passwords so they match and then you can continue to append onto the "retype password" field and can still submit. Upon signing in, I confirmed that only the first password field was the one that was stored.
2.1.3-4	1	OpenMRS should accept the password with the space without truncation.	System accepted the password without truncation	pass	VIK	Note: same phenomenon that I described above happens in the "enter a new password" screen caused by the "force password change" button
2.1.4-1	1	OpenMRS should accept the unicode password as one unicode code point = one character	System accepted password with 64 kanji characters, one capital letter, and one number	pass	VIK	Question: the instructions say that the password should be 64 kanji as well as meet all requirements, so am I ok to append "1A" to the end of my string that I used as a password?
2.1.4-2	1	OpenMRS should accept the unicode password as one unicode code point = one character	System accepted password with 64 kanji characters, one capital letter, and one number	pass	VIK	
2.1.5-1	1	Openmrs should require the current password of a user when the user is changing their password.	System requires user to input current password as part of password change request	pass	VIK	
2.1.5-1	2	OpenMRS should require the new password when allowing a user to change their password.	System requires that the user also input a new password as part of password change request	pass	VIK	
2.1.5-1	3	OpenMRS should allow the user to change their password, and should only accept that new password for logging in as that user.	System only accepts new password after successful submission of password change request	pass	VIK	
2.1.7-1	1	OpenMRS should not accept the known breached/leaked password. User should not be created	System accepted a breached password	fail	VIK	Reason for failure: System accepted "Password1" as a new password
2.1.7-2	1	OpenMRS should not accept the known breached/leaked password. User should not be created	System accepted a breached password	fail	VIK	Reason for failure: System accepted "Password1" as a new password
2.1.7-3	1	OpenMRS should not accept the known breached/leaked password. User should not be created	System accepted a breached password	fail	VIK	Reason for failure: System accepted "Password1" as a new password
2.1.8-1	1	OpenMRS should provide a password strength meter in some form to help users set a stronger password.	System did not show a password strength meter	fail	VIK	
2.1.8-2	1	OpenMRS should provide a password strength meter in some form to help users set a stronger password.	System did not show a password strength meter	fail	VIK	
2.1.8-3	1	OpenMRS should provide a password strength meter in some form to help users set a stronger password.	System did not show a password strength meter	fail	VIK	
2.1.9-1	1	OpenMRS should not use any password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. The password should be accepted if the minimum character count is reached and if the password is not commonly-used, is not expected, and is not compromised.	System did not allow a password of only lowecase alpha characters	fail	VIK	Confirmed that there are several password restriction rules in place under Global Properties.
2.1.9-2	1	OpenMRS should not use any password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. The password should be accepted if the minimum character count is reached and if the password is not commonly-used, is not expected, and is not compromised.	System did not allow a password of only lowecase alpha characters	fail	VIK	

Unique ID	Expected Result #	Expected Result Description	Actual Results	Status (pass/fail)	Researcher	Notes
2.1.9-3	1	OpenMRS should not use any password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. The password should be accepted if the minimum character count is reached and if the password is not commonly-used, is not expected, and is not compromised.	System did not allow a password of only lowecase alpha characters	fail	VIK	
2.1.10-1	1	OpenMRS should not require passwords to be changed arbitrarily (e.g., periodically), unless there is evidence of compromise of the authenticator.	No such rules found in the system	pass	VIK	
2.1.10-1	2	No periodic credential rotation or password history requirements should documented.	No such rules found in the system	pass	VIK	
2.1.11-1	1	OpenMRS should allow "paste" functionality for passwords. Password should be "pasted" correctly from text editor to application's password textbox.	User is capable of pasting passwords	pass	VIK	
2.1.11-1	2	OpenMRS should allow for the use of browser username and password managers. Auto-save and auto-fill of username and password via browser manager should be successful.	System allows for browser to autofill usernames and passwords	pass	VIK	
2.1.11-1	3	OpenMRS should allow for the use of password managers. Auto-save and auto-fill of username and password via password manager should be successful.	Bitwarden did not prompt to save login credentials at signup. Bitwarden did not prompt to input credentials at login.	fail	VIK	Note: I'm not entirely sure if this test is a failure. Perhaps bitwarden just isn't the best software to use?
2.1.11-2	1	OpenMRS should allow "paste" functionality for passwords. Password should be "pasted" correctly from text editor to application's password textbox.	User is capable of pasting passwords	pass	VIK	
2.1.11-2	2	OpenMRS should allow for the use of browser username and password managers. Auto-save and auto-fill of username and password via browser manager should be successful.	System allows for browser to autofill usernames and passwords	pass	VIK	
2.1.11-2	3	OpenMRS should allow for the use of password managers. Auto-save and auto-fill of username and password via password manager should be successful.	Bitwarden did not prompt to save login credentials at signup. Bitwarden did not prompt to input credentials at login.	fail	VIK	Note: I'm not entirely sure if this test is a failure. Perhaps bitwarden just isn't the best software to use?
2.1.11-3	1	OpenMRS should allow "paste" functionality for passwords. Password should be "pasted" correctly from text editor to application's password textbox.	User is capable of pasting passwords	pass	VIK	
2.1.11-3	2	OpenMRS should allow for the use of browser username and password managers. Auto-save and auto-fill of username and password via browser manager should be successful.	System allows for browser to autofill usernames and passwords	pass	VIK	
2.1.11-3	3	OpenMRS should allow for the use of password managers. Auto-save and auto-fill of username and password via password manager should be successful.	Bitwarden did not prompt to save login credentials at signup. Bitwarden did not prompt to input credentials at login.	fail	VIK	Note: I'm not entirely sure if this test is a failure. Perhaps bitwarden just isn't the best software to use?
2.1.12-1	1	A user should be able to either temporarily view the entire masked password, or temporarily view the last typed character of the password when creating a password.	User can see masked version of password	pass	VIK	
2.1.12-1	2	All three tests should be successful (a viewer does exist)	User can see masked version of password in all three circumstances of typing a pasword	pass	VIK	
2.2.1-1	1	OpenMRS should stop the brute force attack by the use of soft lockouts, rate limiting, CAPTCHA, increasing delays between attemps, IP address restrictions, risk-based restrictions (such as location or first login on a device), and recent attemps to unlock the account.	System stops user from logging in after multiple failed login attempts.	pass	VIK	Note: executed manually, no longer able to login with the account used for the test but no message or notification was displayed that I have been locked out.
2.2.1-1	2	OpenMRS should employ some mechanism to stop the user from logging in normally after a large amount of brute force attempts, or a mechanism to authenticate safely. (E. g.: rate limiting will stop the user from logging in.	System stops user from logging in after multiple failed login attempts.	pass	VIK	Note: executed manually, no longer able to login with the account used for the test but no message or notification was displayed that I have been locked out.
2.2.2-1	1	OpenMRS should not employ weak authenticators, such as SMS, email, and time based one-time passwords (OTPs), as primary login methods.	System does not use weak authenticators for logging users in	pass	VIK	
2.2.2-1	2	OpenMRS should employ a strong primary authentication method, such as loggin in with a strong password the user previously created.	System only accepts the password tied to the user's account	pass	VIK	
2.2.3-1	1	OpenMRS should send a secure notification to the user that their credentials were updated/changed.	Browser makes a small alert saying the password was changed	pass	VIK	What is a "secure notification"? There is a note in the test document saying this version of OpenMRS does not send emails
2.2.3-1	2	The notification must be secure. For example, neither the old or new credentials should be displayed in plaintext within the notification.	Browser notification does not contain credentials	pass	VIK	
2.2.3-2	1	OpenMRS should send a secure notification to the user that a login from an unknown/risky location was attempted.		can't run		need to find way to change ip address of the VM

Unique ID	Expected Result #	Expected Result Description	Actual Results	Status (pass/fail)	Researcher	Notes
2.2.3-2	2	The notification must be secure. For example, no credentials or other account information should be displayed in plaintext within the notification.		can't run		need to find way to change ip address of the VM
2.3.1-1	1	OpenMRS should employ a method to generate secure, initial passswords that are at least 6 characters long and that expire after a short period of time.	System does not provide password generation services	fail	VIK	
2.3.1-1	2	OpenMRS should force the user to eventually change passwords and to not let the user keep that inital password for the long term.	System does not require user to change password	fail	VIK	Note: test document brings up a good point that we don't know/don't have a limit
2.5.2-1	1	When a new user account is created in OpenMRS, at no point should "password hints" or "knowledge-based authetnication" /"secret questions" be present.	System does not ask for password hints or security questions when making a new account	pass	VIK	
2.5.2-1	2	User is never asked to input password hints or give answers to questions with the intention to use as authentication later.	System does not ask for password hints or security questions when making a new account	pass	VIK	
2.5.3-1	1	OpenMRS should not reveal the current password in any way during password credential recovery.	System says to contact system administrator	pass	VIK	Is this really a pass? If there is no real password recovery system then are we even able to test this....
2.5.4-1	1	OpenMRS should not use any shared or default accounts such as "root", "admin", or "sa".		can't run		Instructions for this test seem incomplete, there is an admin account but it is not the default when making a new account
2.5.4-1	2	All usernames should be specific to the user and not shared.	System does not allow admin to create an account with a non-unique username	pass	VIK	
2.5.6-1	1	Authentication recovery mechanisms utilized by OpenMRS should be secure (such as by via token OTP (one-time password), mobile push notification, or offline mechanism)		can't run	VIK	See test 2.5.3-1, there isn't a recovery procedure that we can follow to complete this test.

A.3.2 Access Control (Section 4)

Unique ID Part 1	Unique ID Part 2	OpenMRS Functionality	OpenMRS User Interface Location	Expected Result #	Expected Results	Actual Results	Status (pass/fail)	Researcher	Notes
4-	1	View Patient Data	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=coreapps.findPatient	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	only the admin could access the page	pass	monica	
4-	2	Edit Patient Demographic Information	Find Patient -> View Patient -> Edit	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	I was able to access the edit patient form as user Grace	fail	monica	
4-	3	Edit Patient Contact Information	Find Patient -> View Patient -> Show Contact Info -> Edit	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	I was able to access the edit patient information form as user Grace	fail	monica	
4-	4	Edit Patient Allergies	Find Patient -> Allergies	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	User grace was able to send the POST request while logged in via Postman. grace can also view the page via URL, but cannot POST from the URL	fail	monica	
4-	5	Patient Visit Management	Find Patient -> Start Visit	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	only the admin could access the page	pass	monica	
4-	6	View Active Visits	http://localhost:8080/openmrs/coreapps/activeVisits.page?app=coreapps.activeVisits	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	I was able to access the page as user Grace	fail	monica	
4-	7	Add Patient Vitals	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=referenceapplication.vitals	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	I was able to access the page as user Grace	fail	monica	
4-	8	Manage User Accounts	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	Only the admin could send successfully retire the user	pass	monica	Grace was able to send the request and get an error page back, but the retire request did not go through
	8	Manage User Accounts	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	2	Administrative interfaces use appropriate multi-factor authentication to prevent unauthorized use. Before the DELETE request is processed, the user is authenticated again.	no re-authentication. Just prompted for a reason.	fail	monica	
4-	L9	Legacy Manage User Accounts	http://localhost:8080/openmrs/admin/users/users.list	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	I was able to access the page as user Grace	fail	monica	
4-	L10	Legacy Manage Patient Accounts	http://localhost:8080/openmrs/admin/patients/index.htm	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	I was able to access the page as user Grace	fail	monica	
4-	L11	Legacy Reporting	http://localhost:8080/openmrs/module/reporting/dashboard/index.form	1	Only the admin (an authorized user of the functionality) should be able to access/modify the target application functionality. The user, Grace, should only be able to authenticate, view the home page, and access their account settings.	I was able to access the page as user Grace	fail	monica	The URL did not work, but I was able to get back and view the report by clicking around the interface
4-	L11	Legacy Reporting	http://localhost:8080/openmrs/module/reporting/dashboard/index.form	2	Administrative interfaces use appropriate multi-factor authentication to prevent unauthorized use. Before viewing the report with sensitive information, the authorized user should be authenticated again.	no re-authentication	fail	monica	

A.3.3 Validation, Sanitization, and Encoding (Section 5)

Unique ID Part 1	Unique ID Part 2	OpenMRS Functionality	OpenMRS User Interface Location	Expected Result #	Expected Results	Actual Results	Status (pass/fail)	Researcher	Notes
5-	1	Login	http://localhost:8080/openmrs/login.htm	1	Admin was not authenticated	Admin was not authenticated	pass	VIK	
5-	1	Login	http://localhost:8080/openmrs/login.htm	2	Attack strings are neutralized/sanitized/rejected before insertion	Attack strings are neutralized/sanitized/rejected before insertion	pass	VIK	attack strings used: (" or "a"="a , #' , %00../../../../../../etc/passwd , %0Aid , %3Cscript%3Ealert(%22X%20SS%22);%3C/script%3E , %99999999999s)
5-	1	Login	http://localhost:8080/openmrs/login.htm	3	URL forwards or redirects only allow for whitelisted destination or show a warning	URL forward was not allowed, input was sanitized	pass	VIK	redirect attack string used: %2f{example.org}//
5-	1	Login	http://localhost:8080/openmrs/login.htm	4	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	2	Find Patient	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=coreapps.findPatient	1	Attack strings are neutralized/sanitized/rejected before insertion	Attack strings are neutralized/sanitized/rejected before insertion	pass	VIK	attack strings used: (" or "a"="a , #' , %00../../../../../../etc/passwd , %20'sleep%2050' , %3cscript%3ealert("XSS");%3c/script%3e)
5-	2	Find Patient	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=coreapps.findPatient	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	3	Edit Patient Demographic Info	Find Patient -> View Patient -> Edit	1	Application is protected against HTTP parameter pollution, does not display the inserted patientID's info	Application is NOT protected against HTTP parameter pollution, it displayed the inserted patientID's info	fail	VIK	
5-	3	Edit Patient Demographic Info	Find Patient -> View Patient -> Edit	2	Attack strings are neutralized/sanitized/rejected before insertion	Attack strings are stored	fail	VIK	attack strings used: (%60 , %3cscript%3ealert("XSS");%3c/script%3e , %0a ping -i 30 127.0.0.1 %0a , %00/etc/passwd%00 , $NULL) its worth noting that while all of these attack strings got stored as is, none of them seemed to be exectuting even the JS alert strings
5-	3	Edit Patient Demographic Info	Find Patient -> View Patient -> Edit	3	URL forwards or redirects only allow for whitelisted destination or show a warning	URL forward was not allowed	pass	VIK	redirect attack string used: ?url=http://%7Bexample.org%7D
5-	3	Edit Patient Demographic Info	Find Patient -> View Patient -> Edit	4	In case of an error, a generic message is displayed which does not give the attacker information about the system	Detailed system stack trace was displayed	fail	VIK	stack trace includes lines that show program architechture and dependencies
5-	4	Edit Patient Contact Info	Find Patient -> View Patient -> Show Contact Info -> Edit	1	Attack strings are neutralized/sanitized/rejected before insertion	Attack strings are stored	fail	VIK	attack strings used: (!' , ") or ("a"="a , "><script>alert(1)</script> , #xA, %20$(sleep%2050)) again, the strings are stored but seem to not be executing
5-	4	Edit Patient Contact Info	Find Patient -> View Patient -> Show Contact Info -> Edit	2	Structured data is strongly typed - can detect unreasonable input combinations such as mismatch in city and ZIP	Accepted invalid input (City: Raleigh, State: NC, Country: US, Zip: 66012)	fail	VIK	
5-	4	Edit Patient Contact Info	Find Patient -> View Patient -> Show Contact Info -> Edit	3	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	5	Edit Patient Allergies	Find Patient -> Allergies	1	Attack strings are neutralized/sanitized/rejected before insertion	Attack strings caused stack traces that revealed system information	fail	VIK	attack strings used: ($null , %20n , %0a ping -i 30 127.0.0.1 %0a, &#60) all of these caused a stack trace with valuable information since they are not enums in the codebase
5-	5	Edit Patient Allergies	Find Patient -> Allergies	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		

5-	6	Add Patient Visit Note and Diagnosis		1	Application is protected against HTTP parameter pollution, does not display comment3 or comment4	Application is protected from HTTP parameter pollution	pass	VIK	
5-	6	Add Patient Visit Note and Diagnosis		2	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings are stored	fail	VIK	attack strings used: (< , ' UNION ALL SELECT ,)))))))))) , ..%u2216 , ;id|) all are stored but do not seem to execute
5-	6	Add Patient Visit Note and Diagnosis		3	URL forwards or redirects only allow for whitelisted destination or show a warning	URL forward was not used	pass	VIK	redirect attack string used: ?view=///{example.org}
5-	6	Add Patient Visit Note and Diagnosis		4	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	7	Add Patient Vitals	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=referenceapplication.vitals	1	Application is protected against HTTP parameter pollution, does not display comment3 or comment4	Application is protected from HTTP parameter pollution	pass	VIK	
5-	7	Add Patient Vitals	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=referenceapplication.vitals	2	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings are neutralized and are not stored	pass	VIK	attack strings used: (xsstest , `ping 127.0.0.1` , \00\00 , <<script>alert("XSS"); //<</script> , 2147483647)
5-	7	Add Patient Vitals	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=referenceapplication.vitals	3	URL forwards or redirects only allow for whitelisted destination or show a warning	URL forward did not occur	pass	VIK	redirect attack string used: %2f{example.org}
5-	7	Add Patient Vitals	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=referenceapplication.vitals	4	In case of an error, a generic message is displayed which does not give the attacker information about the system	error messages are simple and general	pass	VIK	there was some small error handling done by the app (ie. small red text above the field saying "not a number" or "cannot exceed value"
5-	8	Find User Account	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	1	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings are neutralized	pass	VIK	attack strings used: (" or "a" ="a , #' , %00../../../../../../etc/passwd , %20'sleep%2050' , %3cscript%3ealert ("XSS");%3c/script%3e)
5-	8	Find User Account	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	9	Retire User	Find User -> Edit -> Retire	1	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings are neutralized (cannot validate if they are stored or not)	pass	VIK	attack strings used: ($null , <script>alert(1)</script> , %20d , & id , ' UNION ALL SELECT)
5-	9	Retire User	Find User -> Edit -> Retire	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	10	Create User Account	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	1	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings are neutralized	pass	VIK	attack strings used: (%60 , %3cscript%3ealert("XSS");%3c/script%3e , %0a ping -i 30 127.0.0.1 %0a , %00/etc/passwd%00 , $NULL)
5-	10	Create User Account	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	2	In case of an error, a generic message is displayed which does not give the attacker information about the system	small error messages above the fields are simple and general	pass	VIK	
5-	11	Manage Encounter Roles	http://localhost:8080/openmrs/adminui/metadata/encounters/encounterroles/manageEncounterRoles.page#/list	1	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings were not sanitized and were stored	fail	VIK	attack strings used: (!' , '') or ("a"="a , "><script>alert(1)</script> , #xA, %20$(sleep%2050))
5-	11	Manage Encounter Roles	http://localhost:8080/openmrs/adminui/metadata/encounters/encounterroles/manageEncounterRoles.page#/list	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?

5-	L12	Legacy Find User Account	http://localhost:8080/openmrs/admin/users/users.list	1	Application is protected against HTTP parameter pollution, does not display results with duplicate name parameters	Application is protected from HTTP parameter pollution attack, query is marked as invalid	pass	VIK	
5-	L12	Legacy Find User Account	http://localhost:8080/openmrs/admin/users/users.list	2	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings do not execute	pass	VIK	attack strings used: (' or username like '% , ' or '1'='1 , ">xxx<P>yyy , $null, !') it seems that the strings never get sanitized (the parameter in the search bar shows the full string, but they never execute) so I will mark it as a pass for now
5-	L12	Legacy Find User Account	http://localhost:8080/openmrs/admin/users/users.list	3	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	L13	Legacy Create User Accounts	http://localhost:8080/openmrs/admin/users/users.list	1	Attack strings are neutralized/sanitized/ rejected before insertion	Attack strings are neutralized and users with malicious strings are not created	pass	VIK	attack strings used: (xsstest , `ping 127.0.0.1` , \00\00 , <<script>alert("XSS"); //<</script> , 2147483647)
5-	L13	Legacy Create User Accounts	http://localhost:8080/openmrs/admin/users/users.list	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	L14	Legacy Register a Patient	http://localhost:8080/openmrs/admin/patients/index.htm	1			skipped		cannot generate patient, no "generate" checkbox appears for ID
5-	L14	Legacy Register a Patient	http://localhost:8080/openmrs/admin/patients/index.htm	2			skipped		
5-	L15	Legacy Find Patient	http://localhost:8080/openmrs/findPatient.htm	1	Attack strings are neutralized/sanitized/ rejected before insertion	attack strings are rejected	pass	VIK	attack strings used: (<IMG SRC="javascript:alert ('XSS');"> , 2147483647, -2147483647, ,@variable,)))))))))))
5-	L15	Legacy Find Patient	http://localhost:8080/openmrs/findPatient.htm	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	L15	Legacy Find Patient	http://localhost:8080/openmrs/findPatient.htm	3	URL forwards or redirects only allow for whitelisted destination or show a warning	it doesn't look like the redirect workef	pass	VIK	redirect string used: {example.org}
5-	L16	Legacy Edit Patient Information	http://localhost:8080/openmrs/admin/patients/index.htm	1	Attack strings are neutralized/sanitized/ rejected before insertion		fail	VIK	
5-	L16	Legacy Edit Patient Information	http://localhost:8080/openmrs/admin/patients/index.htm	2	In case of an error, a generic message is displayed which does not give the attacker information about the system		pass		I didn't get an error message, what should I put?
5-	L16	Legacy Edit Patient Information	http://localhost:8080/openmrs/admin/patients/index.htm	3	Structured data is strongly typed - can detect unreasonable input combinations such as mismatch in city and ZIP	Accepted invalid input (City: Raleigh, State: NC, Country: US, Zip: 66012)	fail	VIK	
5-	L16	Legacy Edit Patient Information	http://localhost:8080/openmrs/admin/patients/index.htm	4	Structured data is strongly typed - notifies user that "abc" is not a valid identifier	Error message pops up and prevents saving of changes while identifier is invalid	pass	VIK	

A.3.4 Logging (Section 7)

Unique ID Part 1	Unique ID Part 2	OpenMRS Functionality	OpenMRS User Interface Location	Expected Result #	Expected Results	Actual Results	Status (pass/fail)	Researcher	Notes
7.1-	1	Register a Patient	http://localhost:8080/openmrs/registrationapp/registerPatient.page?appId=referenceapplication.registrationapp.registerPatient	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	user=admin loged only
7.1-	1	Register a Patient	http://localhost:8080/openmrs/registrationapp/registerPatient.page?appId=referenceapplication.registrationapp.registerPatient	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive info. Patient = Patient#null, method savePatient, User=admin.	pass	monica	
7.1-	2	View Patient Data	Find Patient -> Patient's Profile	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	
7.1-	2	View Patient Data	Find Patient -> Patient's Profile	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive info.	pass	monica	UserService.saveUser. Arguments: User=admin,
7.1-	3	Edit Patient Data	Find Patient -> Patient's Profile	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	Patient=Patient#10
7.1-	3	Edit Patient Data	Find Patient -> Patient's Profile	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive info. Uses a patient identifier: Patient=Patient#10 and User = admin	pass	monica	
7.1-	4	Patient Encounter Management	Find Patient -> Patient's Profile	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	log lists methods saveUser and voidVisit. In the future, we might consider this sensitive information, but it does not violate privacy laws.
7.1-	4	Patient Encounter Management	Find Patient -> Patient's Profile	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive info. User = admin, Visit = Visit #2, String = delete visit (there was no option to leave a reason)	pass	monica	In method ConditionService.save. Arguments: Condition=Condition [hashCode=b8b342c5, uuid=13ca68a9-81bb-4197-b898-754576ce5a4f], | Exiting method save | error at org.openmrs.module.emrapi.conditionslist.Condition.concept
7.1-	5	Modify Patient Conditions	Find Patient -> Patient's Profile -> Conditions	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	moncia	These do not violate privacy laws, but may be considered info disclosure later?
7.1-	5	Modify Patient Conditions	Find Patient -> Patient's Profile -> Conditions	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive data according to local laws.	pass	monica	
7.1-	6	Add Patient Visit Note and Diagnoses	http://localhost:8080/openmrs/coreapps/activeVisits.page?app=coreapps.activeVisits	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	n method EncounterService. saveEncounter. Arguments: Encounter=Encounter: [(no ID) Mon Nov 18 07:58:17 PST 2019 Visit Note Inpatient Ward 10 Visit Note num Obs: [obs id is null, obs id is null, obs id is null] num Orders: 0], | In method ObsService.saveObs. Arguments: Obs=obs id is null, String=null, | User=admin
7.1-	6	Add Patient Visit Note and Diagnoses	http://localhost:8080/openmrs/coreapps/activeVisits.page?app=coreapps.activeVisits	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive data according to local laws.	pass	monica	
7.1-	7	Modify Patient Allergy List	Find Patient -> Patient Profile -> Allergies	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	NO LOGS: The application did not log anything! Only the logs for finding/viewing a patient were produced.
7.1-	7	Modify Patient Allergy List	Find Patient -> Patient Profile -> Allergies	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive data according to local laws.	pass	monica	
7.1-	8	Add Patient Vitals	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=referenceapplication.vitals&	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	In method EncounterService. saveEncounter. Arguments: Encounter=Encounter: [(no ID) Mon Nov 18 08:01:24 PST 2019 Vitals Inpatient Ward 10 Vitals num Obs: [obs id is null, obs id is null, obs id is null, obs id is null, obs id is null, obs id is null, obs id is null, obs id is null, obs id is null] num Orders: 0], | In method ObsService. saveObs. Arguments: Obs=obs id is null, String=null,
7.1-	8	Add Patient Vitals	http://localhost:8080/openmrs/coreapps/findpatient/findPatient.page?app=referenceapplication.vitals&	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive data according to local laws.	pass	monica	
7.1-	9	Manage User Accounts	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	| In method PersonService. savePerson. Arguments: Person=Person(personId=null), | In method UserService. createUser. Arguments: User=jsmith, String=<Arg value ignored>, | In method UserService.retireUser. Arguments: User=clerk, String=retire reason box 123,

Unique ID Part 1	Unique ID Part 2	OpenMRS Functionality	OpenMRS User Interface Location	Expected Result #	Expected Results	Actual Results	Status (pass/fail)	Researcher	Notes
7.1-	9	Manage User Accounts	http://localhost:8080/openmrs/adminui/systemadmin/accounts/manageAccounts.page	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application could potentially log sensitive info in the comment box via user input. Outputs comment box text to log.	fail	monica	uses 6a834023 as the account identifier, lists admin and new user usernames (admin, jsmith)
7.1-	10	Legacy Manage User Accounts	http://localhost:8080/openmrs/admin/users/users.list	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	In method UserService. unretireUser. Arguments: User=jsmith, | In method UserService.saveUser. Arguments: User=jsmith
7.1-	10	Legacy Manage User Accounts	http://localhost:8080/openmrs/admin/users/users.list	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive info defined by local laws.	pass	monica	
7.1-	11	Legacy Register a Patient	http://localhost:8080/openmrs/admin/patients/index.htm	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	In method PatientService. savePatient. Arguments: Patient=Patient#null, |
7.1-	11	Legacy Register a Patient	http://localhost:8080/openmrs/admin/patients/index.htm	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive info defined by local laws.	pass	monica	
7.1-	12	Legacy View Patient Data	http://localhost:8080/openmrs/findPatient.htm	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	NO LOGS
7.1-	12	Legacy View Patient Data	http://localhost:8080/openmrs/findPatient.htm	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application does not log any sensitive info defined by local laws.	pass	monica	NO LOGS
7.1-	13	Legacy Edit Patient Data	http://localhost:8080/openmrs/admin/patients/index.htm	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	| In method PatientService. voidPatient. Arguments: Patient=Patient#12, String=legacy hello reason for deleting patient 123,
7.1-	13	Legacy Edit Patient Data	http://localhost:8080/openmrs/admin/patients/index.htm	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	application could potentially log sensitive info in the comment box via user input. Outputs comment box text to log. I can see String=legacy hello reason for deleting pateint 123	fail	monica	
7.1-	14	Legacy Reporting	http://localhost:8080/openmrs/module/reporting/dashboard/index.form	1	The application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.	application does not log any credentials, payment info, or session tokens	pass	monica	INFORMATION DISCLOSURE OpenmrsUtil. getDirectoryInApplicationDataDirectory(1150) |2019-11-18 08:24:08,805| '/var/lib/tomcat8/.OpenMRS/REPORT_RESULTS' doesn't exist. Creating directories now.
7.1-	14	Legacy Reporting	http://localhost:8080/openmrs/module/reporting/dashboard/index.form	2	The target application does not log other sensitive data as defined under local privacy laws or relevant security policy.	Application lists sensitive information about system environement (/var/lib/tomcat8), but this does not violate local privacy laws like HIPPA.	pass	monica	In method ReportService. saveReportRequest. Arguments: ReportRequest=Report Request (88fabecc-072c-41a3-a332-ae7a9aa1865e)for List of New Patient Registrations to DefaultWebRenderer, |

A.4 Discarded Test Cases

Test Case #	ASVS #	Unique ID	CWE
4	3	2.2.3-2	320

Repeatable Steps:

1. Follow the instructions to create a user or use a user that has already been created.
2. Login to that user from an unknown location (a different device).
3. Repeat step 2 with a different IP address.

Expected Results:

1. OpenMRS should send a secure notification to the user that a login from an unknown/risky location was attempted.
2. The notification must be secure. For example, no credentials or other account information should be displayed in plaintext within the notification.

Test Case #	ASVS #	Unique ID	CWE	N.I.S.T
3	4	2.5.4-1	16	5.1.1.2, Appendix 3 (A.3)

Repeatable Steps:

1. Verify that a shared or default account is not present.

Expected Results:

1. OpenMRS should not use any shared or default accounts such as "root", "admin", or "sa".
2. All usernames should be specific to the user and not shared.

Test Case #	ASVS #	Unique ID	CWE
5	6	2.5.6-1	640

Repeatable Steps:

1. Complete Test Case unique ID 2.5.3-1

Expected Results:

1. Authentication recovery mechanisms utilized by OpenMRS should be secure (such as by via token OTP (one-time password), mobile push notification, or offline mechanism)

5-L14 Legacy Register a Patient

Assumptions: None.

Procedure:

1. Authenticate as admin.
2. Open the legacy user interface for managing patient accounts. Click Create New Patient.
3. Using a proxy, for the personName.givenName parameter, enter an attack string from the injection attack list.
4. Repeat step 4 with four additional attack strings.
5. Repeat steps 3-4 with the following parameters: patient.birthdate, patient.birthdateEstimated, personAddress.address2, and patientId.

NOTE: for ID, use OpenMRS ID, enter any 6-digit number

Expected Results:

1. The injection attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected such that the attack is not executed or stored.
2. If an unexpected or security sensitive error occurs and the target application shows an error message, it is a generic message potentially with a unique ID which support personnel can use to investigate. The generic message does leak sensitive information (such as a stack trace or other internal error details that can provide an attacker with information about the system's environment).