2011 International Symposium on Empirical Software Engineering and Measurement

One Technique is Not Enough:

A Comparison of Vulnerability Discovery Techniques

Andrew Austin and Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, USA
andrew_austin@ncsu.edu, williams@csc.ncsu.edu

***Abstract*—Security vulnerabilities discovered later in the development cycle are more expensive to fix than those discovered early. Therefore, software developers should strive to discover vulnerabilities as early as possible. Unfortunately, the large size of code bases and lack of developer expertise can make discovering software vulnerabilities difficult. To ease this difficulty, many different types of techniques have been devised to aid developers in vulnerability discovery. *The goal of this research is to improve vulnerability detection by comparing the effectiveness of vulnerability discovery techniques and to provide specific recommendations to improve vulnerability discovery with these techniques.* We conducted a case study on two electronic health record systems to compare four discovery techniques: systematic and exploratory manual penetration testing; static analysis; and automated penetration testing. In our case study, we found empirical evidence that no single technique discovered every type of vulnerability. We discovered almost no individual vulnerabilities with multiple discovery techniques. We also found that systematic manual penetration testing found the most design flaws, while static analysis found the most implementation bugs. Finally, we found the most effective vulnerability discovery technique in terms of vulnerabilities discovered per hour was automated penetration testing. These results suggest that if one has limited time to preform vulnerability discovery one should conduct automated penetration testing to discover implementation bugs and systematic manual penetration testing to discover design flaws.**

Keywords-security, vulnerability, static analysis, penetration testing, blackbox testing, whitebox testing

I. INTRODUCTION

Security vulnerabilities discovered later in the development cycle are more expensive to fix than those discovered early [1]. Therefore, software developers should strive to discover vulnerabilities as early as possible in the development lifecycle. Unfortunately, modern code bases are increasingly growing, and finding security vulnerabilities is hard. Difficulty in finding security vulnerabilities is further compounded by software developers potentially lacking security expertise. Many tools and techniques have been created to help ease the difficulty in discovering vulnerabilities. Not all tools or techniques are the same, so developers are left to decide how they can best discover vulnerabilities on their own.

In his book, *Software Security: Building Security In*, Gary McGraw draws on his experience as a security researcher and claims [2]: "Security problems evolve, grow, and mutate, just like species on a continent. No one technique or set of rules will ever perfectly detect all security vulnerabilities." Instead, he advocates using a combination of penetration testing (with the aid of a tool) and static analysis vulnerability discovery techniques throughout the software development lifecycle. McGraw's claim is not substantiated with empirical evidence. Empirical evidence may affirm the experience of McGraw, or instead show that security vulnerabilities are better discovered by other tools or techniques. *The goal of this research is to improve vulnerability detection by comparing the effectiveness of vulnerability discovery techniques and to provide specific recommendations to improve vulnerability discovery with these techniques.*

To accomplish our research goal, we conducted a case study to examine vulnerability discovery techniques on two web-based electronic health record systems: Tolven Electronic Clinician Health Record (eCHR) [1] and OpenEMR[2]. These two systems are currently used within the United States to store patient records. In our case study, we conducted exploratory and systematic manual penetration testing; static analysis; and automated penetration testing. We classified the vulnerabilities found as either implementation bugs or design flaws. We then manually analyzed each discovered vulnerability to determine if the same vulnerability could be found by multiple vulnerability discovery techniques.

The contributions of this paper are as follows:

- A comparison of the type and number of vulnerabilities found with exploratory manual penetration testing, systematic manual penetration testing, static analysis, and automated penetration testing.
- Empirical evidence indicating which discovery techniques should be used to find both implementation- and design-level vulnerabilities.
- An evaluation of the efficiency of each vulnerability discovery technique based on the metric vulnerabilities discovered per hour.

[1] http://sourceforge.net/projects/tolven/
[2] http://www.oemr.org/

978-0-7695-4604-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ESEM.2011.18

The rest of the paper is organized as follows; Section II provides background information requiring familiarity to understand the paper. Section III discusses related academic work. Section IV describes our case study and its methodology. Section V gives our study results. Section VI discusses our results and provides analysis. Section VII discusses our limitations. Section VIII summarizes our conclusions. Finally, Section IX talks about possible future work.

II. BACKGROUND

This section describes the terminology used throughout the paper and gives background information on the types of security issues one may encounter when doing security analysis.

A. Vulnerability Discovery Techniques

Penetration testing is testing that is identifying "the unspecified and insecure side effects of 'correct' application functionality [3]". Penetration testing is not focused on verifying the program specification. Manual **penetration testing** is penetration testing performed without the aid of an automated tool [4]. We make the distinction between two types of manual penetration testing: exploratory manual testing and systematic testing. **Exploratory manual penetration testing** is manual penetration testing without a test plan. Instead, exploratory manual penetration testing is a security evaluation based on the tester's instinct and prior experience. **Systematic manual penetration testing** is testing that follows a predefined test plan rather than exploration. To reduce testing time and take advantage of repetitive nature of testing, tools have been devised to automatically perform many of the same tasks that one does in manual penetration testing. These tools are called **automated penetration testing** tools [4].

Rather than looking at the security of an application from a user perspective, tools can also look for security issues by examining the code directly. Automated **Static analysis** examines software in an abstract fashion by evaluating the code without executing it with the aid of a tool [5] [6]. This examination can be performed by evaluating either source code, machine code, or object code of an application to obtain a list of potential vulnerabilities found within the source. Static analysis can be performed using a variety of techniques, from scanning with simple patterns [5], data flow analysis [7], to even model checking [8].

Techniques for discovering software vulnerabilities are not perfect and they sometimes incorrectly label code as containing a fault. This mislabeling is called a **false positive,** as opposed to a **true postive**, when faults are correctly identified. Therefore, developers must manually examine each potential fault reported by these tools to determine if they are false positives. We call potential faults that have security implications **potential vulnerabilities.**

Security faults can be divided into two groups: design flaws and implementation bugs. **Design flaws** are high-level problems associated with the architecture of the software. **Implementation bugs** are code-level software problems [2]. We will identify these classes each time we uncover new security vulnerabilities in this paper.

B. Vulnerability Types

The following implementation bug descriptions are based on their Common Weakness Enumeration descriptions[3]. **cross-Site scripting (XSS)** (CWE-79) vulnerabilities occur when input is taken from a user and not correctly validated, allowing for malicious code to be injected into a web browser and subsequently displayed to the end user. **SQL Injection** (CWE-89) vulnerabilities occur when user input is not correctly validated and the input is directly used in a database query. Not validating the input allows malicious user to directly manipulate the data returned by the database to potentially obtain sensitive information. A **dangerous function** (CWE-242) vulnerability occurs when a method is used within code that is inherently insecure or deprecated. Such methods or functions should not be used because attackers can use common knowledge of their weakness to exploit the application. A **path manipulation** (CWE-22) vulnerability occurs when users are allowed to view files or folders outside of those intended by the application. An **error information leak** (CWE-209) vulnerability occurs when information or an error is displayed directly to a user. These errors can contain sensitive information or even authentication credentials to allow attackers greater access to the application. A failure to set the **HTTPOnly attribute** allows for non-http access to browser cookies. Such a vulnerability allows client site code to access the cookies particularly allowing session information or other sensitive data to be stolen in cross site scripting or phishing attacks [9]. A **hidden field manipulation** (CWE-472) vulnerability occurs when data in hidden fields are not properly validated and the field is implicitly trusted. Trusting this form of user input can lead to issues such as SQL injection and cross site scripting, or can allow inaccurate information to be inserted into the database. A **command injection** (CWE-78) vulnerability occurs when input from the user is directly executed. This vulnerability allows malicious users to directly execute commands on the host as a trusted user.

There are also several vulnerabilities that are design flaws [10]. We will examine several. A **nonexistent access control** (CWE-285) vulnerability occurs when access to a particular URL is not protected, granting anyone, including malicious users access. A **lack of auditing** (CWE-778) vulnerability occurs when a critical event is not logged or recorded. A **Trust Boundary Violation (**CWE-501) occurs when trusted and untrusted data is mixed in a data structure. **Dangerous File Upload** (CWE-434) can occur when the system is not properly designed to handle potentially malicious files.

[3] The Common Weakness Enumeration is a community developed dictionary of software weakness types [10].

III. Related Work

Researchers have already examined some differences between vulnerability discovery techniques. Autunes and Vieira compared the effectiveness of static analysis and automated penetration testing in detecting SQL injection vulnerabilities in web services [11]. They found more SQL injection vulnerabilities with static analysis than with automated penetration testing. They also found that both static analysis and automated penetration testing had a large false positive rate. In our work we focus on more than just static analysis and automated penetration testing as discovery techniques. We also look at more of a variety of vulnerabilities to compare techniques.

Research by Doupé, et al. [12] evaluated 11 automated penetration testing tools. In their evaluation they found that modern automated penetration tools had trouble accessing all resources provided by an application due to weaknesses in crawling algorithms. Automated penetration testing tools particularly had trouble with Flash and JavaScript. Additionally, they found that some types of vulnerabilities such as command injection, file inclusion and cross site scripting via Flash were difficult for automated penetration tools to find.

Suto [13] [14] conducted two studies in which he evaluated seven commercial automated penetration testing tools. In his studies, he found that tools missed many vulnerabilities because they could not properly reach all pages of the web applications. He also found that most commercial tools had a large number of false positives.

Baca et al. [15] found that the average developers were unable to determine if a static analysis alert was a security issue. They found that having experience with static analysis doubled the number of correct true positive classifications, while having both security experience and static analysis tripled correct classification over average developers.

Rutar, et al. [16] conducted a case study on five static analysis tools comparing their effectiveness. They found that the tools discovered non-overlapping bugs that were not found by the other tools.

McGraw and Steven [17] published an article on the pitfalls of comparing static analysis tools. They state that two tools will perform differently on code bases of the same language because of coding style and internal rules used by the tools. They also claim that tool operators and configuration can greatly influence vulnerability discovery.

Much work in the past pertaining to manual penetration testing has focused on the lack of scientific process in penetration testing. Several researchers have concluded that manual penetration testing is more of an art than a science [18] [19]. As a result, the penetration tester's creativity and skill greatly influence the results of a successful manual penetration test.

IV. Case Study

This section describes the subjects chosen for our case study as well as our methodology.

A. Subject Selection

Due to legislative requirements in the United States, development and adoption of electronic health record (EHR) systems has suddenly increased. To help our case study generalize to large real world systems, we have chosen two open source EHR systems as subjects for study. The two systems we studied were Tolven eCHR and OpenEMR.

Tolven eCHR is an open source EHR system. The project has 12 contributing developers, and commercial support is provided by Tolven, Inc. Some additional characteristics of Tolven eCHR are provided in Table I. **OpenEMR** is an open source EHR system. The project has a community of 17 contributing developers and at least 23 organizations providing commercial support within the United States [20]. Additional characteristics of OpenEMR are provided in Table I.

TABLE I. Characteristics of Tolven eCHR and OpenEMR

	Tolven eCHR	OpenEMR
Language	Java	PHP
Version Evaluated	RC1 (5/28/2010)	3.1.0 (8/29/2009)
Lines of Code (counted by CLOC1.08[4])	466,538	277,702

B. Case Study Methodology

We first collected the vulnerabilities that each vulnerability discovery technique discovered. We then classified whether each of the vulnerabilities were true or false positives. The next five subsections examine the steps of our case study methodology in detail.

1) Exploratory Manual Penetration Testing

To keep other discovery techniques from biasing our exploratory manual penetration testing, we conducted exploratory manual penetration testing prior to conducting vulnerability discovery with other techniques. To perform exploratory manual penetration testing, the first author manually attempted to exploit various components of the test subjects in an ad-hoc manner. The exploratory manual penetration testing was conducted by authenticating with the target application and manually navigating through each page trying various attacks. The first author drew on his application security experience and his knowledge of the target applications to look for a variety of vulnerabilities in the system. The first author used supplemental tools like web browsers, JavaScript debuggers (e.g. Firebug[5]) and http proxies (e.g. WebScarab[6]) for viewing raw http requests.

2) Static Analysis

To perform static analysis, we used Fortify 360 v.2.6[7]. Fortify 360 supports analysis of a variety of languages including both PHP and Java. To evaluate these two

[4] http://cloc.sourceforge.net/
[5] http://getfirebug.com/
[6] http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
[7] https://www.fortify.com/products/fortify360/index.html

languages we chose the options "Show me all issues that have security implications" and "No I don't want to see code quality issues". Fortify 360 generated a list of potential vulnerabilities when scanning was complete.

3) Automated Penetration Testing

To conduct automated penetration testing, we used IBM Rational AppScan 8.0[8]. Rational AppScan conducts a black box security evaluation of the website by crawling the web application and attempting a variety of attacks. To use AppScan, we provided authentication credentials to the systems so that the tool could login to both our test subjects. We left the default scanning options selected for our automated penetration testing. AppScan generated a list of potential vulnerabilities when scanning was complete.

4) Systematic Manual Penetration Testing

One vulnerability discovery method, proposed by Smith and Williams [21], suggests using a software systems functional requirement specification's English statements to systematically generate security tests to surface security vulnerabilities. The authors created these tests by breaking the systems functional requirement statements into distinct phrase types such "Action Phrase" and "Object Phrase." Using these two phrases the authors then propose a systematic method to generate security tests using common patterns. Since the authors have provided a detailed test plan [21] and have run their test plan on our subjects, we will use the results they obtained for our study.

5) False Positive Classification

Both static analysis and automated penetration testing generate a list of potential vulnerabilities that must be classified as either true or false positives. To perform this classification, the first author manually examined each individual vulnerability. For static analysis, we examined the line of code classified as vulnerable and also examined related methods. For automated penetration testing, false positive classification was performed by looking at the raw HTTP requests generated and confirming if the attempted exploit was actually visible in the raw output or accepted as trusted input. For both tools, sometimes the first author had to attempt to manually recreate the attack through the application to confirm whether the potential vulnerability was a true positive.

V. RESULTS

This section describes our results.

A. Exploratory Manual Penetration Testing

For Tolven eCHR, the first author spent approximately fifteen man-hours preforming exploratory manual penetration testing. After fifteen hours of evaluation, we were unable to find any security issues in Tolven eCHR based on our exploratory manual penetration testing. To compare this discovery technique with other techniques, we computed an efficiency metric, vulnerabilities discovered per hour. Since we discovered no vulnerabilities in Tolven eCHR with exploratory manual testing, our vulnerabilities per hour metric is 0.

In prior work [22], we conducted an extensive security evaluation of OpenEMR with a team of six researchers and 30 man-hours of evaluation. Because discovery of vulnerabilities can signal the penetration tester to other likely vulnerabilities, we continued our evaluation of OpenEMR for a longer period than Tolven eCHR. During our manual testing we were able to find 12 security vulnerabilities throughout the OpenEMR application. Fig. 1 provides a breakdown of the types of vulnerabilities discovered.

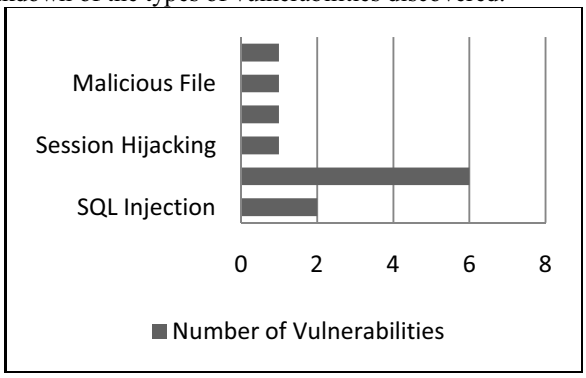

Figure 1. Vulnerabilities Found In OpenEMR with Exploratory Manual Penetration Testing.

Because all of the 12 vulnerabilities discovered were discovered manually and we were able to exploit each one, they are all considered true positives. All of the bugs found with exploratory manual penetration testing were implementation bugs with the exception of the malicious file upload bug, which was a design flaw. All of the implementation bugs found with exploratory manual penetration testing were caused by lack of input validation. Since exploratory manual penetration testing found 12 vulnerabilities in 30 hours, the efficiency metric is 0.40 vulnerabilities per hour.

B. Static Analysis

Static analysis for Tolven eCHR generated a list of 3,765 potential vulnerabilities. Despite only scanning for security issues, there were 1450 issues reported had no security implications. For example, Fortify 360 reported "J2EE Bad Practices" and "Code Correctness" issues even after explicitly scanning only for security issues. Removing the non-security issues reported by Fortify 360 resulted in a total of 2,315 issues with security implications.

The first author spent about 18 hours manually classifying these potential vulnerabilities as either true or false positives. Speed of classification was greatly enhanced by the Fortify 360 user interface. Lines containing potential vulnerabilities could be viewed with a single click and vulnerabilities in a single file could also be grouped. The speed of classification was also influenced by the similarity and quantity of false positives. For example, many XSS vulnerabilities had similar structure and layout, so the analysis involved checking for differences in a common

[8] http://www-01.ibm.com/software/awdtools/appscan/

pattern and determining how those differences influenced the potential vulnerability. Because of these similar issues, it could take 5-10 seconds to evaluate a line of code in some cases, or up to several minutes for more complicated issues. After pruning for false positives, 50 true positive vulnerabilities were identified giving a 98% false positive rate. We found 50 true positives in 18 hours of testing, for a vulnerabilities per hour measurement of 2.78. Table II breaks down the types of vulnerabilities discovered and their false positive rates.

TABLE II. STATIC ANALYSIS VULNERABILITIES IN TOLVEN ECHR

Type	True Positives	False Positives	False Positive Rate
SQL Injection	5	24	83%
Cross Site Scripting	28	182	87%
System Information Leak	13	441	97%
Header Manipulation	2	1	33%
File Upload Abuse	2	0	0%
Weak Cryptography or Randomness	0	111	100%
Weak Access Control	0	225	100%
Command Injection	0	2	100%
Denial of Service	0	57	100%
J2EE Misconfiguration	0	19	100%
LDAP Issues	0	28	100%
HTTP Verb Tampering	0	2	100%
JavaScript Hijacking	0	39	100%
Log Forging	0	114	100%
Deprecated Method	0	18	100%
Misused Authentication	0	2	100%
Password Management	0	337	100%
Path Manipulation	0	151	100%
Poor Logging	0	218	100%
Privacy Violation	0	31	100%
Race Condition	0	31	100%
Resource Injection	0	9	100%
Setting Manipulation	0	21	100%
Trust Boundary Violation	0	10	100%
Unsafe Reflection	0	19	100%
Weak XML Schema	0	173	100%
Total	**50**	**2265**	**98%**

With an overall false positive rate of 98%, most of the time spent in analyzing the potential vulnerabilities result in a false positive. Static analysis did best in pointing out common input validation attacks such as SQL injection, and XSS. Despite finding these issues, the false positive rates for detecting these vulnerabilities was still high.

Static analysis did quite poorly on several types of vulnerabilities. One was "Weak Cryptography or Randomness." Every time a pseudo-random number generator was used, static analysis labeled it as a potential vulnerability. In Tolven eCHR, the security of the application did not depend on these pseudo-random numbers so every occurrence was a false positive. Similarly, every time Tolven eCHR printed output to the console or threw an exception, static analysis would label it as a "System Information Leak." In practice, none of these issues would be displayed to the end user. Finally, a large number of false positives were labeled as "Password Management" issues. Simply having strings such as "password" or "******" in comments would trigger this alert.

OpenEMR under Fortify 360 generated a list of 5,036 potential vulnerabilities. The first author spent approximately 40 man hours going through all the potential vulnerabilities classifying them as either a true positive or a false positive. After pruning false positives, 1,321 true positive vulnerabilities were identified giving a false positive rate of 74%. With static analysis we found 1,321 true positives vulnerabilities in 40 hours. This gives us a vulnerabilities discovered per hour metric of 32.40. Table III summarizes our findings.

TABLE III. STATIC ANALYSIS VULNERABILITIES IN OPENEMR

Type	True Positives	False Positives	False Positive Rate
SQL Injection	984	12	1%
Cross Site Scripting	171	3138	95%
System Information Leak	29	56	66%
Hidden Fields	119	15	11%
Path Manipulation	7	86	92%
Dangerous Function	7	0	0%
HTTPOnly Not Set	1	0	0%
Dangerous File Inclusion	2	110	98%
File Upload Abuse	1	8	88%
Command Injection	0	44	100%
Insecure Randomness	0	23	100%
Password Management	0	36	100%
Header Manipulation	0	17	100%
Other	0	170	100%
Total	**1321**	**3715**	**74%**

Static analysis was able to find 984 SQL injection vulnerabilities in OpenEMR. OpenEMR uses a custom method that has insufficient input validation to execute all database queries. To determine if an invocation was vulnerable, the first author only had to look at the method invocation parameters. This significantly sped up the time to evaluate SQL injection potential vulnerabilities. Static analysis also reported 3,309 XSS issues in OpenEMR. While 171 of these issues were true positives, the vast majority of them were not. Instead the input was actually validated in some way and the tool failed to correctly understand this validation.

C. Automated Penetration Testing

Running AppScan on Tolven eCHR resulted in 37 security issues after roughly eight hours of unattended scanning. It took roughly one hour to go through the 37 potential vulnerabilities. Only 22 of these 37 issues were true positives, giving a 40% false positive rate. Since we found 22 true positives in one hour of evaluation, the vulnerabilities per hour metric is 22.00. Table IV provides our results.

TABLE IV. AUTOMATED PENETRATION TEST VULNERABILITIES IN TOLVEN ECHR

Type	True Positives	False Positives	False Positive Rate
Session Identifier Not Updated	0	3	100%
Cross Site Request Forgery	0	1	100%
Cacheable SSL Page	0	9	100%
Missing HttpOnly Attribute	5	0	0%
System Information Leak	17	0	0%
Email Address Pattern	0	2	100%
Total	**22**	**15**	**40%**

Seventeen occurrences of "System Information Leak" and five occurrences of "Missing HTTPOnly Attribute" vulnerabilities were true positive vulnerabilities. The only considerable number of false positives occurred with the type "Cacheable SSL Page." All these issues occurred with common JavaScript libraries like jQuery as the cacheable resource and were subsequently deemed false positives.

AppScan found 735 potential vulnerabilities in OpenEMR after six and a half hours scanning. The first author spent roughly ten hours going through all of these issues and classifying if they were either true positives or false positives. After classification, 710 true positives remained from the 735 potential vulnerabilities, giving a false positive rate of 3%. The low false positive rate is especially good considering automated penetration testing found 710 true positive vulnerabilities We found 710 true positive vulnerabilities in 10 hours of evaluation, giving us a vulnerabilities per hour metric of 71.00.

Table V shows the breakdown of the type of vulnerabilities found. Automated penetration testing did particularly well at finding input validation vulnerabilities such as SQL injection, XXS, and Error Information Leak vulnerabilities. Of these three types of vulnerabilities, the false positive rate was 0%.

TABLE V. AUTOMATED PENETRATION TESTING VULNERABILITIES IN OPENEMR

Type	True Positives	False Positives	False Positive Rate
Cross Site Scripting	7	0	0%
SQL Injection	214	0	0%
System Information Leak	467	0	0%
Directory Traversal	18	0	0%
Email Address Patterns	0	5	100%
Missing HTTP Only Attribute	4	0	0%
HTML Information Leak	0	3	100%
JavaScript Cookie Manipulation	0	6	100%
Phishing Through Frames	0	8	100%
Session ID Not Updated	0	1	100%
Unencrypted Login	0	2	100%
Total	**710**	**25**	**3%**

Looking at the results of the automated penetration test, OpenEMR had an order of magnitude more true positives than Tolven eCHR. The difference in the number of true positives is due largely to the fact that OpenEMR fails to adequately validate user input. This lack of input validation leads to a majority of the issues in OpenEMR such as XSS, SQL injection, system information leak, and directory traversal. Both applications did poorly at output validation, opting to rely solely on input validation. The Defense in Depth security design principle [23] suggests that that both input and output validation should be used. Such a design flaw was not caught by automated penetration testing. The inability to find such a design flaw is due in part to the difficulty of automated penetration testing in looking beyond the user interface to see what the application is actually doing with the data at the code level.

D. Systematic Manual Penetration Testing

In the original systematic security test plan proposal, the authors conducted a case study that included both OpenEMR and Tolven eCHR. The authors test plan included 137 black box tests. The following results are pulled directly from their case study for comparison. The authors spent 60 man hours evolving their test plan methodology and creating their test plan. Between six and eight man hours were spent testing each EHR system [21].

OpenEMR failed 63 of 137 tests. Fig. 2 breaks down the vulnerabilities found in OpenEMR with systematic manual penetration testing. Since the authors found 63 vulnerabilities in 67 hours, the vulnerabilities per hour metric is 0.94. Also note that this number maybe be low as the sixty hour number

given by the authors included time for the evolution of their methodology.

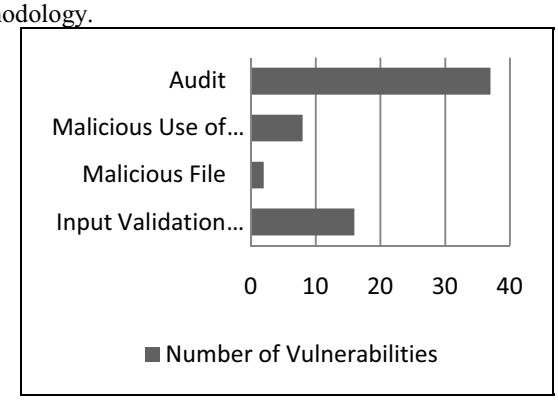

Figure 2. Vulnerabilities Found In OpenEMR with Systematic Manual Penetration Testing

All of the input validation vulnerabilities found by systematic manual penetration testing were implementation bugs. These 16 implementation bugs were comprised of 15 XSS vulnerabilities and one SQL injection vulnerability. The rest of the issues reported by the systematic manual penetration test were design issues.

Tolven eCHR failed 37 of 137 tests. Since the authors found 37 vulnerabilities in 67 hours, the vulnerabilities per hour metric is 0.55. Fig. 3 breaks down the vulnerabilities found in Tolven eCHR with the systematic manual penetration test.

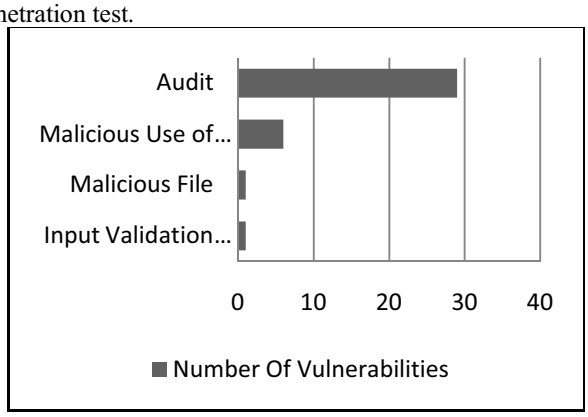

Figure 3. Vulnerabilities Found In Tolven eCHR with Systematic Manual Penetration Testing

In Tolven eCHR there was only one input validation vulnerability discovered with the systematic security test plan. This input validation vulnerability was an error information leak vulnerability. The vulnerability is therefore an implementation bug. The other 36 vulnerabilities were all design issues.

In both subjects, systematic manual penetration testing found a majority of design issues, but it also found several implementation bugs. Since both types of vulnerabilities occur with roughly equal frequency in the wild, having a technique that finds both is important [2].

VI. Analysis & Discussion

The first subsection discusses and analyzes the vulnerabilities discovered. The second subsection discusses the efficiency of the various discovery techniques, while the third subsection talks about several vulnerabilities the discovery techniques discussed failed to find.

A. Comparing Vulnerabilites Discovered

In this section, we provide results that aggregate the specific vulnerabilities found with Tolven eCHR and OpenEMR. To gain a better understanding of when to use each types of discovery tools, we compare how effective one discovery tool was at detecting the specific vulnerabilities found with other tools. We compared every vulnerability found with the other discovery techniques to every vulnerability we found with static analysis. We chose to compare everything to static analysis initially because it reported the most number of true positives.

First, we compared the vulnerabilities we discovered with static analysis to every vulnerability found with the other discovery techniques. A breakdown vulnerabilities found with static analysis compared to all the other discovery techniques can be found in Table VI. The second column represents the unique number of vulnerabilities found of each particular class using static analysis, while the third through fifth columns represents how many of those vulnerabilities were discovered with each corresponding vulnerability discovery technique.

TABLE VI. VULNERABILITIES FOUND IN STATIC ANALYSIS COMPARED TO ALL OTHER DISCOVER TECHNIQUES

Vulnerability Type	Static Analysis	Manual Testing	Automated Testing	Security Test Plan
SQL Injection	989	2	0	1
Cross Site Scripting	199	3	5	5
System Information Leak	42	0	0	0
Hidden Fields	119	0	0	0
Path Manipulation	7	0	0	0
Dangerous Function	7	0	0	0
No HTTPOnly Attribute	1	0	0	0
Dangerous File Inclusion	2	0	0	0
File Upload Abuse	3	0	0	0
Header Manipulation	2	0	0	0
Total	**1371**	**5/1371**	**5/1371**	**6/1371**

The details of Table V will be discussed in each of the following subsections.

1) Exploratory Manual Penetration Testing

A comparison in the types of vulnerabilities from both EHR systems found with exploratory manual penetration testing and static analysis can be found in Fig. 4.

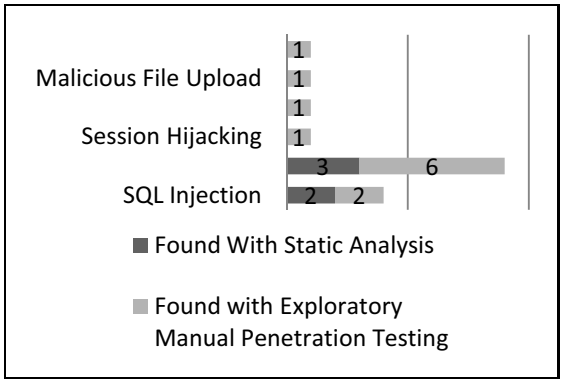

Figure 4. Vulnerabilities Found In Both Exploratory Manual Penetration Testing and Static Analysis

Static analysis was able to find all the SQL injection vulnerabilities found by exploratory manual penetration testing. However, static analysis was able to only find three of the XSS vulnerabilities out of six. Other types of vulnerabilities found with manual testing were not discovered with static analysis. Other static analysis tools would not likely be able to find these issues either; they occur due to the interaction between application components (e.g. browser, server configuration, etc.). These results suggest that only doing static analysis and not some form of black box testing potentially leaves many types of vulnerabilities undiscovered. Similarly, automated penetration testing was unable to find any of the issues discovered by static analysis.

2) Systematic Manual Penetration Testing

A comparison in the types of vulnerabilities from both EHR systems found with systematic manual penetration testing and static analysis can be found in Fig. 5.

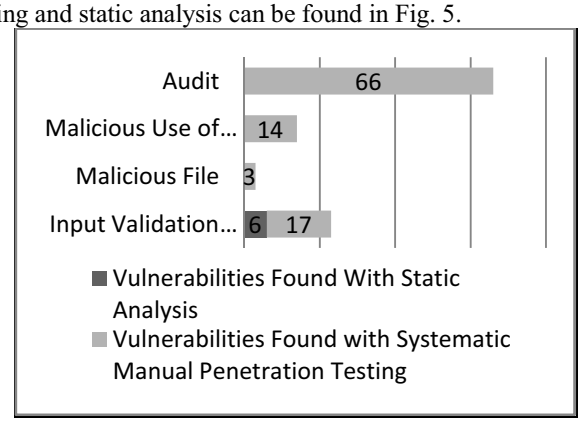

Figure 5. Vulnerabilities Found In Both Systematic Manual Penetration Testing and Static Analysis

With the systematic security test plan, Smith and Williams [21] found 17 input validation vulnerabilities. Of these 17 vulnerabilities, we were able to find six of these with static analysis. The other types of vulnerabilities found with the systematic security test plan were not found by static analysis. All the audit issues the systematic test plan found could not be found with static analysis. Instead, full system tests would have to be used to ensure that adequate auditing and logs were created when specific features were used within the test subjects. These audit vulnerabilities were all design flaws, as were all the malicious use of security function vulnerabilities and the malicious file vulnerabilities. The input validation vulnerabilities were implementation bugs.

Systematic manual penetration testing also found more vulnerabilities compared to exploratory manual penetration testing. Systematic manual penetration testing found all of the vulnerabilities discovered by exploratory manual penetration testing in OpenEMR. The systematic manual test plan also found vulnerabilities in Tolven eCHR even though exploratory manual penetration testing did not.

3) Automated Penetration Testing

A breakdown in vulnerabilities found with automated penetration testing compared to static analysis can be found in Fig. 6. With automated penetration testing we found seven XSS vulnerabilities. Using static analysis we were only able to find five of these seven vulnerabilities. No other vulnerabilities were found by both automated penetration testing and by static analysis. Static analysis did find many vulnerabilities of the same type, but they were not the same vulnerabilities as automated penetration testing and often not even in the same file. One example of this would be the SQL injection class of vulnerabilities. Static analysis was able to find 989 of these vulnerabilities, but they were not the same individual vulnerabilities found with automated penetration testing. These results suggest that using just static analysis or automated penetration testing would be insufficient in discovering the vast majority of vulnerabilities.

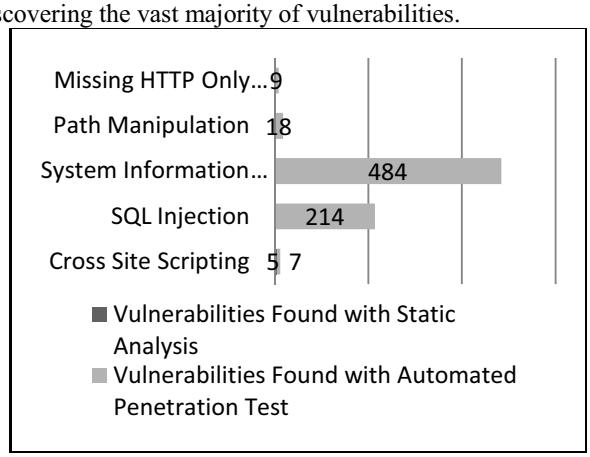

Figure 6. Vulnerabilities Found In Both Automated Penetration Testing

B. Vulnerabilities Per Hour

To get a more complete picture of the vulnerability discovery techniques, we calculate time it took, on average, to discover a vulnerability with each technique. Table VII lists the efficiency calculations for each vulnerability discovery technique. Since Smith and Williams [21] only provided a range we took the average of the times for each evaluation. Also note that the majority of the time for systematic manual penetration testing is creating the test plan, rather than testing the application.

TABLE VII. EFFECIENCY OF VULNERABILITY DISCOVERY TECHNIQUES

Discovery Technique	Vulnerabilities Per Hour	
	Tolven eCHR	OpenEMR
Exploratory Manual Penetration Testing	0.00	0.40
Systematic Manual Penetration Testing	0.94	0.55
Automated Penetration Testing	22.00	71.00
Static Analysis	2.78	32.40

Based on our case study, the most efficient vulnerability discovery technique is automated penetration testing. Static analysis finds more vulnerabilities but the time it takes to classify false positives makes it less efficient than automated testing.

C. Other Observations

Two students conducted a security analysis of Tolven eCHR as part of a class taught by the second author. In their analysis, they found several vulnerabilities that we did not find. Instead, they were discovered using a combination of these and related tools as well as manual testing. Such a finding suggests that individual discovery techniques can inform other vulnerability discovery techniques. Based on these results, using a combination of several techniques to direct your manual testing is an effective way to find additional vulnerabilities..

The first of these is a denial of service attack that occurs due to improper input validation in Tolven eCHR. Fig. 7 contains an attack string that is not properly validated when a doctor edits the personal information of a patient.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [<!ELEMENTbfoo ANY>
<!ENTITY xxe SYSTEM "file:<?xml version="1.0"
encoding="ISO-8859-1"?>
<!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file:////dev/random">]>
<foo>&xxe;</foo>,
```

Figure 7. Input Validation Vulnerability in Tolven eCHR

Other similar input validation vulnerabilities were not caught in manual testing or with any of the discovery tools either. Fig. 8 illustrates an input string that injects a XSS attack.

```
';alert(String.fromCharCode(88,83,83))//
\';alert(String.fromCharCode(88,83,83))/
/";alert(String.fromCharCode(88,83,83))//
\";alert(String.fromCharCode(88,83,83))//>
</SCRIPT>!--
<SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>=
&{}
```

Figure 8. Input Validation Vulnerability in Tolven eCHR.

This attack worked in five different input fields in Tolven eCHR, all of these input fields had to do with the creation of new lists associated with a patient; for example, a new allergy or medication list.

Another trend we observed was the failure to find security vulnerabilities in Tolven eCHR with exploratory manual penetration testing. Exploratory manual penetration testing relies heavily on the skills of the auditor and even skilled auditors can make mistakes and potentially miss vulnerabilities. The task is also highly influenced by auditor creativity as some security holes may require unintuitive manipulations to exploit successfully. The difference in vulnerabilities discovered with exploratory manual penetration testing may also be related to the number of auditors involved. A small group working together may be better than a single auditor working alone because of the ability to bounce ideas off various members.

VII. LIMITATIONS

The tools we selected to represent static analysis and automated penetrating testing may not be representative of other similar tools. We also only used one tool to measure each discovery technique. Other tools may find different types of vulnerabilities. The domain of healthcare, and particularly the open source electronic medical record applications we selected as study subjects, may not be representative of software applications as a whole. These two previous factors may cause our results not to generalize to other subjects or other domains.

Additionally, humans had opportunity to introduce error in the conducted study. Classifying vulnerabilities as either true positives or false positives is very time consuming and potentially error prone. The speed at which classification occurred may have introduced error. Human error may have caused vulnerabilities to be overlooked in the manual testing portions of our study as well.

Efforts were taken to mitigate these possible sources of error; however, we cannot discount their possibility of occurrence entirely.

VIII. CONCLUSION

In our case study we found that systematic manual penetration testing was more effective in finding vulnerabilities than exploratory manual penetration testing. We found that systematic manual penetration testing was the most effective at finding design flaw vulnerabilities. When compared to automated penetration testing and manual testing techniques, static analysis found different types of vulnerabilities. The result of this finding suggest that one cannot rely on static analysis or automated penetration testing because doing so would cause a large number of vulnerabilities to go undiscovered. Static analysis found the largest number of vulnerabilities in our study, but there were a large number of false positives that had to be pruned in a time consuming process. Finally, in calculating the efficiency of each vulnerability detection technique, we found that automated penetration testing found the most vulnerabilities per hour, followed by static analysis, systematic penetration testing and finally manual penetration testing. These results do not refute the opinions of McGraw

discussed in the introduction, but they do suggest that if one has limited time one should conduct automated penetration testing to discover implementation bugs and systematic manual penetration testing to discover design flaws.

IX. Future Work

There are several areas branching from our work that could benefit from further study. The first would be to accurately determine why different types of vulnerabilities were discovered by one tool and not another. Even within a particular type of vulnerability, static analysis and penetration testing found different vulnerabilities. Future work could look to determine why different vulnerabilities of the same type were found with automated penetration testing and static analysis. Another area of future work could be to repeat our study with desktop applications rather than web applications Many different discovery tools are available for desktop applications that have no known web application counterpart. One such tool, KLEE, a symbolic execution tool, was able to find a several serious bugs in the GNU COREUTILS package, some of which had gone unnoticed for over 15 years [24]. Further study could examine greater variety of discovery techniques available for desktop applications.

Acknowledgment

This work is supported by the Agency for Healthcare Research Quality. We would like to thank Eric Helms and Hua Chen for their work in providing additional security vulnerabilities. Additionally, we would like to thank the members of the Realsearch group for their invaluable feedback on our research and this paper.

References

[1] B. Boehm, *Software Engineering Economics*. USA: Prentice Hall, 1984.

[2] G. McGraw, *Software Security: Building Security In*. Boston, USA: Pearson Education, 2006.

[3] H.H. Thompson, "Application penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, p. 66, Jan.-Feb. 2005.

[4] D Allan, "Web application security: automated scanning versus manual penetration testing," IBM Rational Software, Somers, White Paper 2008.

[5] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76-79, Novemeber - December 2004.

[6] W. Pugh and D. Hovemeyer, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, December 2004.

[7] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22-29, Sept.-Oct 2008.

[8] T. Henzinger, R. Jhala, R. Majumdar, and G Sutre, "Software verification with BLAST," in *Proceedings of the 10th international conference on Model checking software (SPIN'03)*, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 235-239.

[9] The Open Web Application Security Project. (2010, August) HttpOnly. [Online]. http://www.owasp.org/index.php/HttpOnly

[10] The MITRE Corporation. (2011, March) Common Weakness Enumeration. [Online]. http://cwe.mitre.org/

[11] N. Antunes and M. Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services," in *15th IEEE Pacific Rim International Symposium on Dependable Computing*, Shanghai , 2009, p. 301.

[12] A. Doupè, M. Cova, and G. Vigna, "Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Bonn, 2010.

[13] L. Suto, "Analyzing the Effectiveness and Coverage of Web Application," San Francisco, White Paper 2007.

[14] L. Suto, "Analyzing the Accuracy and Time Costs of Web Application Security Scanners," San Franscico, White paper 2010.

[15] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?," in *International Conference on Availability, Reliability and Security (ARES '09)*, Fukuoka, 2009, p. 804.

[16] N. Rutar, C.B. Almazan, and J.S. Foster, "A comparison of bug finding tools for Java," in *15th International Symposium on Software Reliability Engineering*, Saint-Malo, 2004, pp. 245-256.

[17] G. McGraw and J. Steven. (2011, January) informIT. [Online]. http://www.informit.com/articles/article.aspx?p=1680863

[18] D. Geer and J. Harthorne, "Penetration testing: a duet," in *18th Annual Computer Security Applications Conference, 2002*, Las Vegas, 2002, p. 185.

[19] S. Robinson, "The art of penetration testing," in *The IEEE Seminar on Security of Distributed Control Systems*, 2005, p. 71.

[20] OEMR.ORG. (2011, February) OpenEMR Commercial Help. [Online]. http://www.openmedsoftware.org/wiki/OpenEMR_Commercial_Help

[21] B. Smith and L Williams, "Systematizing Security Test Planning Using Functional Requirements Phrases," North Carolina State University, Raleigh, Technical Report TR-2011-5, 2011.

[22] B. Smith et al., "Challenges for Protecting the Privacy of Health Information: Required Certification Can Leave Common Vulnerabilities Undetected," in *Security and Privacy in Medical and Home-care Systems (SPIMACS 2010) Workshop*, Chicago, 2010, pp. 1-12.

[23] S. Barnum and M. Gegick. (September, 2005) Defense in Depth. [Online]. https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/principles/347-BSI.html

[24] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs ," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)* , San Diego, 2008.