ABSTRACT

AUSTIN, ANDREW DALE. Improving the Security of Electronic Health Record Systems. (Under the direction of Dr. Laurie Williams).

In the United States, the American Recovery and Reinvestment Act of 2009 (ARRA) provides monetary incentives to healthcare providers for using electronic health record (EHR) systems rather than paper records. By 2015, the ARRA also introduces financial penalties for providers who fail to adopt EHR systems. These legislated financial incentives and penalties are driving mass adoption of EHR systems.

In order to qualify for these incentives and avoid penalties, healthcare providers must use an electronic health record system that meets the U.S. Department of Health and Human Services (HSS) certification criteria. Rather than create a new certification body to certify compliance, HSS designates 3rd party organizations as an Authorized Testing and Certification Body to test and certify each individual EHR system.

How secure are these certified EHR systems? In our research, we examined two questions pertaining to improving the security of electronic health record systems: *1.) Are there any weaknesses in the existing security certification criteria that we can improve on? 2.) How can we improve vulnerability detection efforts in large scale software systems such as electronic health record systems?*

To answer these two questions, we performed three case studies. The first study we conducted examined a popular open source health record system in conjunction with the CCHIT security criteria to find weaknesses and areas of improvement in the certification criteria. The second study generalized our initial findings to a commercial EHR system. Our

third and final study compared a variety of vulnerability discovery techniques to determine their effectiveness on large software systems, such as electronic health record systems.

In our first two case studies we were able to exploit a range of common code-level security vulnerabilities. These common vulnerabilities would not be detected by the 2011 security test scripts from the Certification Commission for Health Information Technology, one of the oldest and well known Authorized Testing and Certification Body. Based on this finding we recommend augmenting the existing security criteria with misuse cases to better model attacker behavior. We also recommend using the augmented security criteria as entry criteria to the EHR certification process. Before spending time certifying EHR systems for functionality, certification bodies should have confidence that basic security issues have been addressed.

In our third 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.

With legislative policy in the United States dictating the adoption of EHR systems it is important to take the time to implement EHR systems in a secure manner. Based on the results of our research, we feel strongly that there is room for improvement for implementing

secure EHR systems. This improvement could be accomplished by augmenting the security criteria that EHR systems implement and by taking the time to do a thorough security analysis using vulnerability discovery tools suited to finding both design- and implementation-level security vulnerabilities.

Improving the Security of Electronic Health Record Systems

by
Andrew Dale Austin

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

2011

APPROVED BY:

_____	_____
Dr. Laurie Williams Committee Chair	Dr. Emerson Murphy-Hill
_____	
Dr. Annie Antón	

DEDICATION

To *Mom, Dad, Adam,* and *Briana*.

BIOGRAPHY

Andrew Dale Austin was born in Charlotte, NC. He received his high school education at Independence High School. He went on to receive a Bachelors of Science with a minor in Mathematics from North Carolina State University. While in high school and college, Andrew worked at software engineering internships at Wachovia, IBM, and Applied Research Associates where he developed software. Andrew was part of the *Realsearch* software engineering group at NC State under the direction of Dr. Laurie Williams. He will be graduating with a Master's degree in December 2011.

ACKNOWLEDGMENTS

Working for and with Laurie the past several years has been a wonderful experience. Her experience and knowledge provided many opportunities to meet interesting people, learn new things, and even travel to exciting places. I could not ask for a better adviser.

The past several years working in the *Realsearch* group have been a wonderful learning experience for me. Andy, Ben, Yonghee, Pat, and Jerrod have all helped provide me with a greater understanding of science and lots of great research to read and contemplate. Thanks to Mei for answering all my questions about grad school and for the great memories with Andy and Laurie at ICSE in Capetown.

Thanks to my dad for introducing me to the world of computers and software development at an early age, he unleashed a love of building software when he introduced me to QBasic. Thanks to my mom for support through college, grad school, and life. Thanks also to my brother who has always been there to talk about anything.

Thanks to my wife, Briana. Words cannot describe the impact of the encouragement and advice she provides.

Thank you to Stephen Roller for always having somebody to talk about research problems, software, and startups with.

This work was sponsored by the Agency for Healthcare Research Quality.

Some of the work in this thesis incorporates material from the following conference proceedings:

A. Austin, B. Smith, and L. Williams, "Towards Improved Security Criteria for Certification of Electronic Health Record Systems". Proceedings of the Second Workshop on Software Engineering in Healthcare (SEHC 2010), co-located with ICSE, Cape Town, South Africa, pp. 68-73, 2010.

B. Smith, A. Austin, M. Brown, J. King, J. Lankford, A. Meneely, L. Williams, "Challenges for Protecting the Privacy of Health Information: Required Certification Can Leave Common Vulnerabilities Undetected", Proceedings of the Security and Privacy in Medical and Home-care Systems (SPIMACS 2010) Workshop, co-located with CCS, Chicago, IL, pp. 1-12, 2010.

A. Austin and L. Williams, "One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques," in Empirical Software Engineering and Measurement (ESEM), Banff, Alberta, Canada, 2011, p. to appear.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Improving the Security of Electronic Health Record Systems

1. INTRODUCTION

In the United States, the American Recovery and Reinvestment Act of 2009 (ARRA) provides monetary incentives to healthcare providers for using electronic health record (EHR) systems rather than paper records. By 2015, the ARRA also introduces financial penalties for providers who fail to adopt EHR systems [1]. These legislated financial incentives and penalties are driving mass adoption of EHR systems.

In order to qualify for these incentives and avoid penalties, healthcare providers must use an electronic health record system that meets the U.S. Department of Health and Human Services (HSS) certification criteria. Rather than create a new certification body to certify compliance, HSS designates 3rd party organizations as Authorized Testing and Certification bodies to test and certify each individual EHR system [2].

One of the first organizations to be designated an Authorized Testing and Certification Body, is the Certification Commission for Health Information Technology (CCHIT) [3]. Prior to the introduction of ARRA, CCHIT was already certifying EHR systems for functionality, interoperability, and security. By the time the ARRA legislated a certification requirement, CCHIT was already the industry leader in EHR certification [4]. When the new legislated criteria were introduced, CCHIT modified its existing criteria to fulfill the ARRA legislated requirements.

In October 2009, CCHIT augmented its existing certification criteria with additional security criteria and provided corresponding black box test scripts. The CCHIT security criteria provide specific requirements intended to establish a minimum level of security of an

EHR system. The test scripts provide step-by-step black box test cases to be manually executed in an effort to ensure these criteria are met.

One would imagine that an EHR system that passed all government required certification criteria, including those that attempt to verify security, would be secure. *But how secure are these EHR systems really?* In our research, we examined two questions pertaining to improving the security of electronic health record systems:

1.) Are there any weaknesses in the existing security certification criteria that we can improve on?
2.) How can we improve vulnerability detection efforts in large scale software systems such as electronic health record systems?

To answer these two questions, we performed three case studies. The first study we conducted examined a popular open source health record system in conjunction with the CCHIT security criteria to find weaknesses of the certification criteria. The second study generalized our initial findings to a commercial EHR system. Our third and final study, focused on improving vulnerability detection processes in large scale software systems, such as electronic health record systems.

In summary, this research makes the following contributions:

- A security evaluation of several open source electronic health record systems.
- An examination of the weaknesses of the security certification criteria provided by CCHIT as well as suggestions for improvement.
- A comparison of the type and number of vulnerabilities discovered with various vulnerability detection techniques.

- Empirical evidence indicating which vulnerability discovery technique 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.

The rest of this paper is organized as follows: Section 2 presents a background of electronic health record systems and software security. Section 3 describes related work in these areas. Section 4 describes our two case studies pertaining to improving security certification criteria. Section 5 presents our case study on improving the vulnerability discovery process. Section 6 provides our recommendations and concludes our work.

2. BACKGROUND

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

2.1 Certification Commission for Health Information Technology

The Certification Commission of Health Care Information Technology (CCHIT) began certifying electronic health record systems in 2006 [5]. Shortly thereafter it was recognized as a certification body by HSS [5]. When the American Recovery and Reinvestment Act was approved by congress in 2009, CCHIT began a new certification program to address the new legislated requirements.

At time of writing, there are 286 CCHIT ambulatory certification criteria. These criteria primarily define required functionality [6]. The 286 criteria map to 213 individual test

scripts. These scripts adhere to six scenarios that reflect common day to day tasks that occur during ambulatory care [7].

46 of the current 286 criteria are related to security. Associated with these 46 security criteria are 60 test scripts and 52 “self-attestation” test scripts. In the self-attestation test scripts the development team must simply "provide supporting documentation as evidence of the product's compliance” [8]. The currently existing security criteria primarily deal with features like encryption and passwords.

2.2 Implementation Bugs and Design Flaws

McGraw divides security faults into two important groups: design flaws, which are high-level problems associated with the architecture of the software; and implementation bugs, which are code-level software problems. Security faults from each group generally occur with the same frequency as the other in any given software project [9].

2.3 Vulnerability Discovery Techniques

Penetration testing is testing that is identifying “the unspecified and insecure side effects of ‘correct’ application functionality [10]”. Penetration testing is not focused on verifying the program specification. Manual penetration testing is penetration testing performed without the aid of an automated tool [11]. 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 [11].

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 [12] [13]. 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 [12], data flow analysis [14], to even model checking [15].

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 positive, 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.

2.4 Vulnerability Types

The following implementation bug descriptions are based on their Common Weakness Enumeration[1] (CWE) descriptions. Cross-Site scripting (XSS) (CWE-79) vulnerabilities occur when input is taken from a user and not correctly validated, allowing for malicious

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

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 [16]. 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 [9]. 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. RELATED WORK

This section describes related research to the topics presented in this research, specifically, papers pertaining to electronic health record system security and vulnerability discovery techniques.

3.1 EHR Security

Several other researchers are exploring security of electronic health record systems. In his doctoral dissertation, Maghazil discussed perceptions of the security of computer-based and paper-based health record systems. He found that hospital employees believed paper-based records to be more secure, but he also found that these same employees believed that electronic-based systems provided more accurate information [18].

Noordende analyzed the Dutch Electronic Patient Dossier System. This system is a national system for exchanging medical records that contains almost all patient information for health patients in the Netherlands. Noordende found that a fundamental problem with the system was insider threats, where people with legitimate access to the system could access patients' record of any other patient in the system. Noordende concluded a system requiring

explicit consent in all but emergency situations would help address this fundamental problem with the national system [19].

Garson and Adams also examined an existing electronic health record system to learn more about the potential security issues. In their study they looked the Mobile Emergency Triage system. Garson and Adam's primary concern was patient privacy and they suggested a policy based encryption scheme to provide access control [20].

The primary difference between our work and these related works is that our work primarily deals with code level issues from a software engineer's perspective rather than high level architectural issues and design features.

3.2 Vulnerability Discovery

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 [21]. They found more SQL injection vulnerabilities with static analysis than with automated penetration testing tools. 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. [22] 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 [23] [24] 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. [25] 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. [26] 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 [27] 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 [28] [29]. As a result, the penetration tester's creativity and skill greatly influence the results of a successful manual penetration test.

4. IMPROVING SECURITY CRITERIA FOR CERTIFICATION OF ELECTRONIC HEALTH RECORD SYSTEMS

This section describes the research we conducted focusing on improving the certification criteria of electronic health record systems. In the following sections we describe two case studies where we examined one open source and on proprietary EHR. *The goal of these case studies is to improve the security assessment within EHR system certification processes by empirically assessing the ability of current security certification criteria to surface a range of vulnerability types.*

4.1 OpenEMR Case Study

To examine the CCHIT security criteria in detail, we performed a case study to examine how the security issues we found in one well known open source system compared to the types of issues one could find with the CCHIT security criteria. Our implementation level evaluation consists of analyzing the results of two web application security tools: IBM's Rational AppScan[2], which performs automated penetration testing; and Fortify 360[3], which performs security-focused static analysis.

4.1.1 Method

This section describes our methodology for selecting and evaluating the target system OpenEMR. We performed the evaluation using Windows Vista Business, Service Pack 2, on a virtual machine with a 2.65Ghz Intel Core Duo and 1.00GB of RAM. OpenEMR was configured to run using Apache 2.2, MySQL v5.1.8, and PHP v5.2.11.

[2] http://www-01.ibm.com/software/awdtools/appscan/
[3] http://www.fortify.com/products/fortify-360/

4.1.1.1 Subject Selection

OpenEMR is an open source EHR system licensed under the GPL [30]. In June 2009, OpenEMR was listed as one of the top ten community-based open source health care projects, according to Black Duck Software [31]. The project has a community of 23 contributing developers and at least 7 companies providing commercial support within the United States [30]. At the time of the original research, OpenEMR was actively pursuing CCHIT certification, and it has since become an ONC-ATB Complete Ambulatory EHR [30]. These facts make OpenEMR an ideal candidate to evaluate because of the ease in which one can access both the source code and support resources. Table 1 lists some additional characteristics of OpenEMR.

Table 1. Characteristics of OpenEMR

Language	PHP
Version Evaluated	3.1.0 (8/29/2009)
Lines of Code (counted by CLOC1.08[4])	277,702

4.1.1.2 Static Analysis

We performed automated static analysis on OpenEMR using the static analysis tool Fortify 360 v5.7. Fortify 360 is a tool focused on security and is able to analyze a variety of languages, including both PHP and Java, which is why it was selected over other static analysis tools. The application was analyzed with the options "Show me all issues that may have security implications" and "No, I don't want to see code quality issues" to only detect potential vulnerabilities. Once the automated analysis was completed, two researchers independently examined each potential vulnerability and its corresponding source code to

[4] http://cloc.sourceforge.net/

classify it as either a true positive or a false positive. Once each potential vulnerability was independently categorized, the two researchers compared their findings. In the event of a disagreement, the researchers examined the potential vulnerability's source together and debated their opinion until a consensus was reached on the validity of each potential vulnerability. Once this consensus was reached, researchers compared the vulnerability against the CCHIT security test scripts to assess whether a test script could have surfaced the identified vulnerability.

Figure 1 presents an example of a SQL injection vulnerability (bolded) that Fortify 360 detected and labeled as "SQL Injection (Input Validation and Representation, Data Flow)". Figure 1 is an example of a static analysis true positive.

```
<?
    $name = $_POST['name'];
    $query = "SELECT id, amount FROM users WHERE name = '$name'";
    $result = mysql_query($query);
?>
```

Figure 1. SQL Injection Example True Positive (PHP)

Figure 2 shows an example of what Fortify 360 labeled (line bolded) as a "Password in Comment – Hardcoded passwords can compromise security in a way that cannot be easily remedied." Figure 2 is a false positive because there is no hardcoded password contained within the code comments, instead the tool simply detects the usage of the word 'password' in the code comments.

```
") VALUES ( "
.
    "'', "
. // username
    "'', "
. // password
```

Figure 2. SQL Injection Example False Positive (PHP)

4.1.1.3 Automated Penetration Testing

To conduct automated penetration testing, we used IBM Rational AppScan v7.8. Rational AppScan performs security testing of web applications, regardless of implementation language or platform. As with the Fortify analysis, two researchers independently examined each potential vulnerability and its corresponding source code to classify it as either a true positive or a false positive. Once each alert was independently categorized, the two researchers compared their findings. In the event of a disagreement, the researchers examined the potential vulnerability's source together and debated their opinion until a consensus was reached.

AppScan was set to scan starting from the OpenEMR's login page. AppScan allows the tester to configure a login policy, which essentially consists of a series of recorded HTTP exchanges to "teach" the tool how to gain authorized access to the system. We configured AppScan to check for "Application Only" tests, which excludes "Infrastructure Tests," which are targeted directly at specific application servers or frameworks, such as Apache Tomcat or Wordpress. We informed AppScan prior to the scan that the application under test was written using PHP, and used MySQL on the backend.

Although AppScan uncovered security issues with OpenEMR, such as the **Directory Listing Pattern** vulnerability type, not every vulnerability AppScan reported was a true positive. One example we encountered of a false positive was the **Email Address Pattern** vulnerability type, which AppScan uses to search for anything in HTTP responses coming from a web application that may resemble e-mail addresses. A webform used in OpenEMR

for controlling batch communication contained an example email address of `your@example.com`, which was not actually a security vulnerability.

4.1.2 Results

This section describes the results of our evaluation conducted on OpenEMR.

4.1.2.1 Static Analysis

Using Fortify 360, we discovered 1,210 potential vulnerabilities related to security with OpenEMR. After the removal of false positives, we determined that there were 440 true positive implementation flaws that would not be detected by the CCHIT certification security test scripts. Table 2 presents a summary of the data we collected.

Table 2. Static Analysis Summary of OpenEMR

Measure	Value
Total Alerts	1210
True Positives	440
False Positives	770
False Positive Rate	63.64%

These 440 true positive vulnerabilities were broken down into the following types, appearing in order of frequency:

- **Cross-Site Scripting (215)**
- **Nonexistent Access Control (129)**
- **Dangerous Function (24)**
- **Path Manipulation (20)**
- **Error Information Leak (19)**
- **Global Variable Manipulation (9)**

- **Insecure Upload (8)**
- **Improper Cookie Use (7)**
- **HTTP Header Manipulation (4)**
- **Hidden Field Manipulation (3)**
- **Command Injection (2)**

As an example, one true positive was the lack of a "user specific secret in order to prevent an attack." The file admin.php, a source file belonging to the events calendar component, requires no authentication for the import and export of data.

Another example of a vulnerability found by fortify is the use of an insecure or deprecated method. One method, `mysql_escape_string()` was used four times throughout the OpenEMR codebase. This method does not properly escape input by taking into account the current character set and is deprecated and even removed in the most recent releases of PHP [32]. Figure 3 shows one example of OpenEMR using this deprecated code (bolded).

```
foreach ($_POST as $k => $var) {
  if (! is_array($var)) $_POST[$k] =
  mysql_escape_string($var);
  echo "$var\n";
}
```

Figure 3. mysql_escape_string() deprecated method (PHP)

4.1.2.2 Automated Penetration Tests

Using Rational AppScan, we discovered 140 potential vulnerabilities with OpenEMR. After the removal of false positives, we determined that there were 130 implementation flaws that would not be detected by the CCHIT certification security test scripts. Table 3 presents a summary of the data we collected.

Table 3. Automated Penetration Test Summary of OpenEMR

Measure	Value
Total Alerts	140
True Positives	130
False Positives	10
False Positive Rate	7.14%

These 130 true positive vulnerabilities were broken down into the following types, appearing in order of frequency:

- **Cross-Site Scripting (50)**
- **Phishing Through Frames (25)**
- **Cross-Site Request Forgery (22)**
- **Error Message Information Leak (14)**
- **SQL Injection (4)**
- **JavaScript Cookie References (6)**
- **Directory Listing (6)**
- **Password Not Encrypted (2)**
- **Path Disclosure (1)**

One vulnerability type AppScan discovered was **Error Message Information Leakage** that displayed the entire structure of a SQL query (see Figure 4). The page a

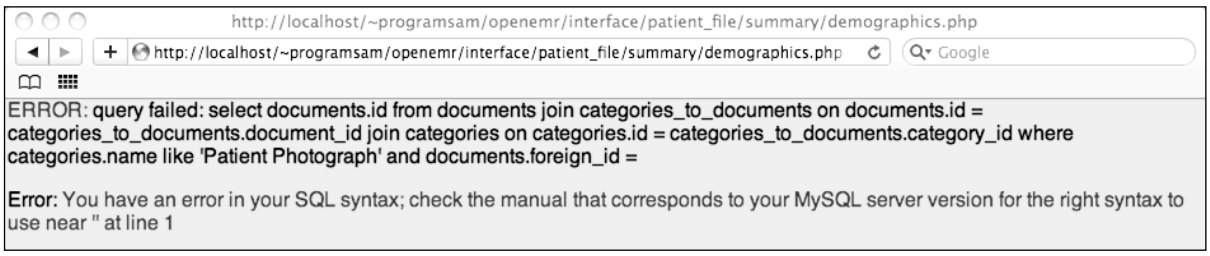

Figure 4. Error Message Information Leak in OpenEMR

practitioner would use to view a patient's personal information, demographics.php, is one example. AppScan was able to set the parameter `set_pid`, which controls the logic of the select query used to browse through patient information, to `null`, which caused OpenEMR to display the result in Figure 4. Such a fault is a dangerous result because it allows the attacker to know what the structure of the query looks like, which makes future SQL injection attacks easier.

Another vulnerability type discovered was **Cross-Site Scripting**. An extended demographics display page, demographics_full.php, has a parameter for `set_pid`, which is the parameter required for the patient's identifier. AppScan set this parameter to:

```
>"'><script>alert(135190)</script>&is_new=>"'><script>alert(135190)</script>
```

Browsing to the vulnerable file caused OpenEMR to execute this script, thus indicating that the application is vulnerable to Cross-Site Scripting.

Of the 130 true positives found by AppScan, 61 were also found previously by Fortify. This overlap in results acts as additional confirmation of the validity of these true positives. The lack of overlap in the vulnerabilities we found helped motivate the research in Chapter 5 of this work.

4.2 ProprietaryMed Case Study

In an effort to generalize our findings related to OpenEMR to other EHR systems, we conducted a similar case study on a proprietary EHR system. The key difference between this study, and the study on OpenEMR, is that we were not provided access to the EHR system's source code for proprietary reasons. Our evaluation was entirely black box. The developers of the EHR system also requested we change the name of this system to protect the system's

identity. Throughout this paper we use the name "ProprietaryMed" to differentiate the system from OpenEMR. More details about ProprietaryMed can be found in the next section under subject selection.

4.2.1 Method

This section describes our methodology for selecting and evaluating the target system ProprietaryMed. We performed the evaluation using a variety of operating systems to attack the system including Windows 7 and Ubuntu Linux 9.10. We also found it useful to have an additional server for various attacks (e.g. session stealing and phishing) that also ran Ubuntu Linux 9.10 and Apache 2.2.12.

4.2.1.1 Subject Selection

ProprietaryMed is a web-based EHR system for use in primary care practices. ProprietaryMed is developed on a Microsoft stack comprising of C#, ASP.NET, WCF web service, and JavaScript. The team developing ProprietaryMed consists of 12 developers. The version we evaluated was version 1.0 which was released in March 2010. The EHR system was implemented in approximately 900 files consisting of over 120,000 lines.

ProprietaryMed is used in real primary care practices. At the time of our evaluation it was installed in 14 practices and used by roughly 80 clinical and non-clinical staff. The company that develops ProprietaryMed estimated the system stores electronic health records for 21,000 patients. Table 4 summarizes these characteristics.

Table 4. Characteristics of ProprietaryMed

Language	C#/ASP.NET
Version Evaluated	1.0 (3/10/2010)
Lines of Code	120,000

4.2.1.2 Security Evaluation

We did not have access to the source code of ProprietaryMed, therefore, it was not possible to evaluate the system using code analysis tools such as Fortify 360. Instead, we created a team of security researchers to perform a black box security audit. The team was comprised of five graduate students and one undergraduate. The students all had experience in developing commercial systems and varying levels of software security knowledge.

Our security evaluation used several tools to assist our black box testing. WebScarab[5] is an application that can act as a proxy to intercept local http request. This tool allowed us a greater understanding of what the application was sending and receiving from the server. It also allowed us to manipulate the raw requests easily to aid our attacks. Another tool we found useful in attacking ProprietaryMed was Firebug. Firebug[6] is primarily known for its JavaScript debugging capabilities. Having a JavaScript debugger allowed us to gain greater understanding of how the application worked on the client side and provided features for manipulating client side elements more easily.

[5] https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

[6] http://getfirebug.com/

4.2.2 Results

Our manual security evaluation uncovered several security flaws in ProprietaryMed. Since each of these vulnerabilities was discovered with black box testing of a live application, they are all true positives.

4.2.2.1 Cross Site Scripting

The most common vulnerability found in our security evaluation was cross site scripting. We exploited six cross-site scripting vulnerabilities in ProprietaryMed. Table 5 lists of each vulnerable feature.

Table 5. Vulnerable Features in ProprietaryMed

Add Medication
Add Allergy
Add Patient Notes
Edit Patient Identifier
Set Patient Issue
Edit Treatment Plan

In all cases, these cross-site scripting vulnerabilities could easily be prevented by using both input and output validation.

4.2.2.2 Session Hijacking

In ProprietaryMed we were also able to steal user's sessions and impersonate them. These attacks leverage many of the cross-site scripting vulnerabilities mentioned in the previous section. Each of these session hijacking attacks could easily be prevented by designating the session cookie as HTTPOnly. The HTTPOnly cookie flag is browser protection that most modern browsers implement. The HTTPOnly cookie flag signifies to the browser that client side code should not be able to read or modify the cookies.

ProprietaryMed failed to take advantage of this security protection. Instead, malicious users could transmit session information to their own server with the following code:

```
<script>window.open("http://ourserver.fake.com/hacking/savecookie.php?cookie=" +
document.cookie, "_new");</script>
```

4.2.2.3 PDF Exploits

ProprietaryMed allows users of the EHR to upload clinical documents to a patient's medical record. This opens a new avenue for attacks on EHR systems. Many modern applications that read PDF files will execute JavaScript or other code embedded inside the PDF. In some instances, attackers can even embed and execute executables without needing JavaScript. Often, such functionality is considered a feature by various vendors, rather than security threats [33]. In our security evaluation, we were able to launch calc.exe on client machines when they opened a PDF uploaded to the EHR system.

4.3 Limitations

There are several limitations that may affect the validity of our two case studies. The results reported by automated tools may not be entirely comprehensive. Future testing could use additional tools to discover faults not originally detected. Researchers examined each potential vulnerability and determined if they were either true or false positives. This examination could possibly introduce human error, which could affect our results. The application we studied, OpenEMR, was not currently CCHIT-certified at the time of our original research (although it has since been certified). ProprietaryMed is also not currently a certified EHR. We only conducted two case studies on two software project, which could potentially not be representative. We tried to mitigate this issue by conducting on one open

source application and one proprietary one. Future studies should investigate the security posture of other open source healthcare applications and other proprietary EHR systems.

4.4 Recommendations

The following section outlines are recommendations for improving the CCHIT security certification criteria. We recommend two changes. The first is to augment the existing criteria with misuse cases. The second is to treat security criteria as entry criteria. Each of these recommendations is explained below.

4.4.1 Augmenting Existing Test Scripts

After analyzing the test scripts and the associated security criteria, we have determined that the CCHIT security criteria only address some design flaws and ignore possible implementation bugs completely. For example, one CCHIT criteria states that "When passwords are used, the system shall support case-sensitive passwords that contain typeable alpha-numeric characters in support of ISO-646/ECMA-6 (aka US ASCII)." Another CCHIT security criterion states that "[t]he system shall provide the ability for authorized administrators to assign restrictions or privileges to users/groups". None of the other 54 non-documentation related CCHIT security criteria test for any potential implementation bugs.

Our results revealed that many of the errors, including numerous input validation vulnerabilities, are disregarded in the existing CCHIT criteria and test scripts.
Consider a misuse case that could be created for the situation where a patient creates his or her own web form to edit another patient's records using the page demographics.php within OpenEMR. The resultant security requirement from this misuse case would indicate that the page demographics.php should contain a user-specific secret, created at runtime, to prevent

an attack of this type. This implementation bug is taken from an example that is detected by Fortify 360 and can be seen in our results.

A generalized form of this misuse case should be included in the CCHIT security test scripts. Rather than a test script that specifically describes an attack on demographics.php, a misuse case can be written to capture the attack pattern demonstrated in the test that would occur for all electronic health records system. In this example, the misuse case would read something similar to "*A patient attempts to modify another patient's demographics that he or she is not authorized to view or edit*."

Such additions to the security test scripts for CCHIT would motivate the creation of additional security criteria. The change would also promote secure coding practices by encouraging developers to build security in early in the development process, a philosophy supported by McGraw [9]. Rather than waiting until late in the development cycle to execute static analysis and penetration testing tools, misuse cases would encourage developers of EHR systems to think about security early in the software lifecycle and to help ensure that EHR systems actually protect our health records.

4.4.2 Security Criteria as Entry Criteria

Since the privacy of patient records is an important component of the security of an EHR system and one of the biggest fears impeding electronic health record adoption, we recommend treating the augmented security certification criteria as entry criteria. Treating security criteria as entry criteria means that the certification process does not proceed until the software system demonstrates it possess some basic level of security protections prior to the certification process. Such a change to the certification process would inform users of the

EHR systems that the system was able to withstand common threats. Entry criteria would also help motivate development organizations to think about security as more than a checklist of features, but instead, a continuous process throughout the software development lifecycle.

5. IMPROVING VULNERABILITY DISCOVERY IN ELECTRONIC HEALTH RECORD SYSTEMS

Security vulnerabilities discovered later in the development cycle are more expensive to fix than those discovered early [34]. 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. The domain of electronic health records is no different from other domains in this regard.

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 [9]: "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.

5.1 Empirical Study

To test McGraw's claim we performed an empirical study with a goal *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 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)[7] and OpenEMR[8]. 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.

5.1.1 Subject Selection

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 6. **OpenEMR** is an open source EHR system. The project has a community of 17 contributing developers and at least 7 organizations providing commercial

[7] http://sourceforge.net/projects/tolven/
[8] http://www.oemr.org/

support within the United States [30]. Additional characteristics of OpenEMR and Tolven eCHR are provided in Table 6.

Table 6. Characteristics of OpenEMR and Tolven eCHR

	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[9])	466,538	277,702

5.1.2 Method

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.

5.1.2.1 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 testers 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 testers drew on their application security experience and knowledge of the target applications to look for a variety of vulnerabilities in the system. The testers used

[9] http://cloc.sourceforge.net/

supplemental tools like web browsers, JavaScript debuggers (e.g. Firebug[10]) and http proxies (e.g. WebScarab[11]) for viewing raw http requests.

5.1.2.2 Static Analysis

To perform static analysis, we used Fortify 360 v.2.6[12]. Fortify 360 can examine a variety of languages including both PHP and Java. To evaluate these two 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.

5.1.2.3 Automated Penetration Testing

To conduct automated penetration testing, we used IBM Rational AppScan 8.0[13]. 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.

5.1.2.4 Systematic Manual Penetration Testing

One vulnerability discovery method, proposed by Smith and Williams [35], suggests using a software systems functional requirement specification's English statements to systematically generate security tests to surface security vulnerabilities. Smith and Williams created these tests by breaking the systems functional requirement statements into distinct

[10] http://getfirebug.com/
[11] http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
[12] https://www.fortify.com/products/fortify360/index.html
[13] http://www-01.ibm.com/software/awdtools/appscan/

phrase types such “Action Phrase” and “Object Phrase.” Using these two phrases the Smith and Williams then propose a systematic method to generate security tests using common patterns. Since Smith and Williams have provided a detailed test plan [35] and have run their test plan on our subjects, we will use the results they obtained for our study.

5.1.2.5 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, we 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 tester had to attempt to manually recreate the attack through the application to confirm whether the potential vulnerability was a true positive.

5.1.3 Results

This section describes our results for each type of vulnerability discovery technique.

5.1.3.1 Exploratory Manual Penetration Testing

For Tolven eCHR, the we 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 [36], 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. Figure 5 provides a breakdown of the types of vulnerabilities discovered.

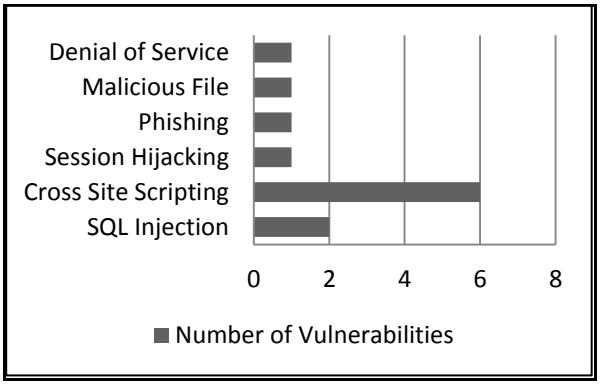

Figure 5. Vulnerabilities Found In OpenEMR with Exploratory Manual Penetration Testing.

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.

5.1.3.2 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.

We 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 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 7 breaks down the types of vulnerabilities discovered and their false positive rates.

Table 7. 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. We 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 8 summarizes our findings.

Table 8. 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, we 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.

5.1.3.3 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 9 provides our results.

Table 9. 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. We 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 10 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 10. 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 [37] 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.

5.1.3.4 Systematic Manual Penetration Testing

In the original systematic security test plan proposal, Smith and Williams conducted a case study that included both OpenEMR and Tolven eCHR. Smith and William's test plan included 137 black box tests. The following results are pulled directly from their case study for comparison. Smith and William's 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 [12].

OpenEMR failed 63 of 137 tests. Figure 6 breaks down the vulnerabilities found in OpenEMR with systematic manual penetration testing. Since we 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 included time for the evolution of their methodology.

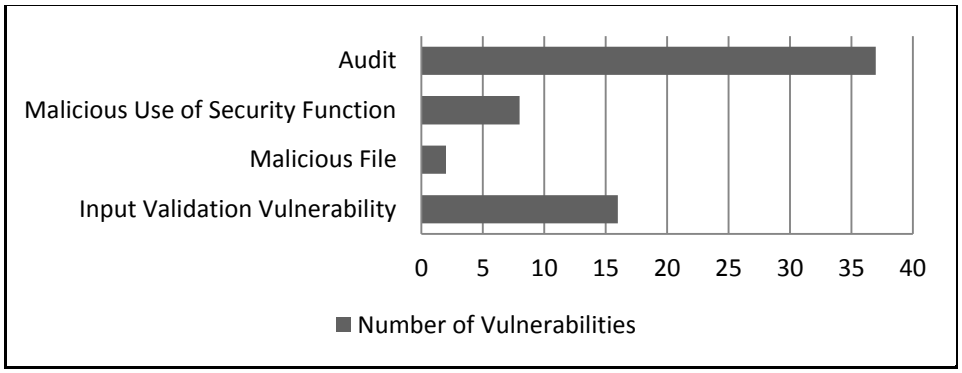

Figure 6. Vulnerabilities Found In OpenEMR with Systematic Manual Penetration Testing

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 we found 37 vulnerabilities in 67 hours, the vulnerabilities per hour metric is 0.55. Figure 3 breaks down the vulnerabilities found in Tolven eCHR with the systematic manual penetration test.

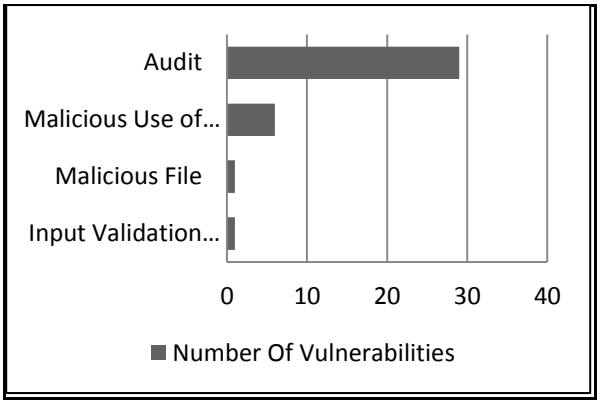

Figure 7. Vulnerabilities Found In Tolven eCHR with Systematic Manual Penetration Testing

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 [10].

5.2 Analysis and 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.

5.2.1 Comparing Vulnerabilities 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 11.

Table 11. Vulnerabilities Found with Static Analysis Compared to All Other Discovery 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 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. The details of Table 11 will be discussed in each of the following subsections.

5.2.1.1 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 Figure 8.

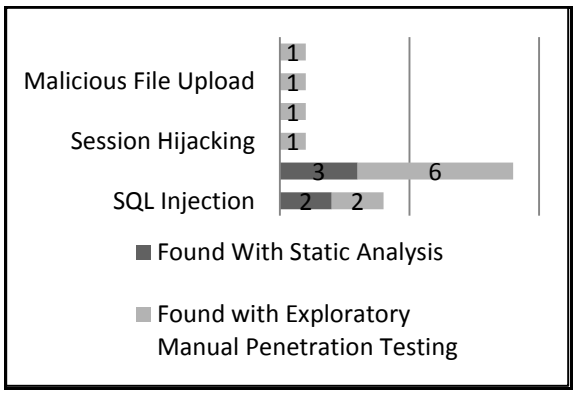

Figure 8. Vulnerabilities Found In Both Exploratory Manual Penetration Testing and Static Analysis

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.

5.2.1.2 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 Figure 9.

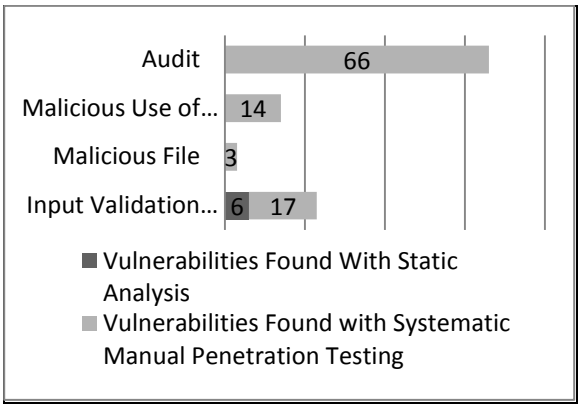

Figure 9. Vulnerabilities Found In Both Systematic Manual Penetration Testing and Static Analysis

With the systematic security test plan, Smith and Williams [35] 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.

5.2.1.3 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.

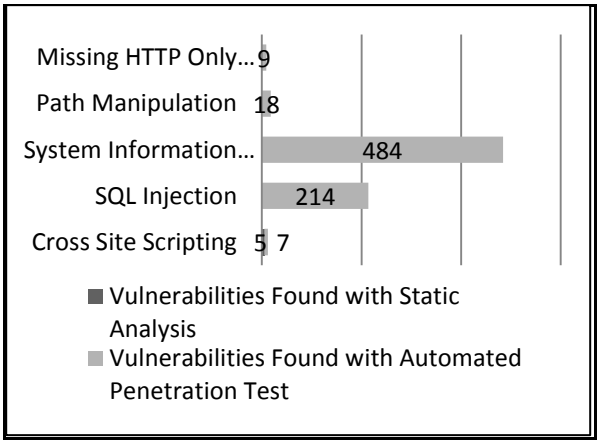

Figure 10. Vulnerabilities Found In Both Automated Penetration Testing

5.2.2 Vulnerabilities Per Hour

To get a more complete picture of the vulnerability discovery techniques, we calculated the time it took, on average, to discover a vulnerability with each technique. Table 12 lists the efficiency calculations for each vulnerability discovery technique. Since Smith and Williams [35] 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 12. Efficiency 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.

5.2.3 Other Observations

Two students conducted a security analysis of Tolven eCHR as part of a class taught at North Carolina State University. 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. Figure 11 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 11. Input Validation Vulnerability in Tolven eCHR

Other similar input validation vulnerabilities were not caught in manual testing or with any of the other discovery tools either. Fig. 12 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 12. 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.

We also 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.

5.3 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 health 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.

5.4 Conclusions

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 suggests 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.

6. CONCLUSIONS AND RECOMMENDATIONS

In our research, we examined two questions pertaining to improving the security of electronic health record systems: *1.) Are there any weaknesses in the existing security certification criteria that we can improve on? 2.) How can we improve vulnerability detection efforts in large scale software systems such as electronic health record systems?*

To answer our first research question we conducted two case studies where we examined one open source and one proprietary EHR system. These systems were examined in conjunction with the CCHIT security criteria to find weaknesses and areas of improvement in the certification criteria. In the two case studies we were able to exploit a range of

common code-level security vulnerabilities. These common vulnerabilities would not be detected by the 2011 security test scripts from the Certification Commission for Health Information Technology. Based on this finding we made two recommends: augmenting the existing security criteria with misuse cases, and treat the security criteria as entry criteria. Misuse cases are more effective at modeling attacker behavior than requirements that take the form "The system shall…". Treating the security criteria as entry criteria to the EHR certification process would influence developers to think about security first and foremost, rather than after the fact when trying to implement a few last minute security features. Before spending time certifying EHR systems for functionality, certification bodies should have confidence that basic security issues have been addressed.

To answer our second research question, we conducted a third case study. In this study, we tested two open source EHR systems using a variety of vulnerability discovery techniques. In using these techniques and analyzing the results, 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.

With legislative policy in the United States dictating the adoption of EHR systems it is important to take the time to implement EHR systems in a secure manner. Based on the results of our research, we feel strongly that there is room for improvement for implementing secure EHR systems. This improvement could be accomplished by augmenting the security criteria that EHR systems implement and by taking the time to do a thorough security analysis using vulnerability discovery tools suited to finding both design- and implementation-level security vulnerabilities.

7. REFERENCES

[1] E. Singer, A Big Stimulus Boost for Electronic Health Records, 2009.

[2] U.S. Department of Health & Human Services. (2010, December) ONC-Authorized Testing and Certification Bodies. [Online]. http://healthit.hhs.gov/portal/server.pt?open=512&mode=2&objID=3120

[3] Certification Commission for Health Information Technology. (2010, August) Certification Commission Among First To Be Approved As ONC-ATCB. [Online]. http://www.cchit.org/media/news/2010/08/certification-commission-among-first-be-approved-onc-atcb

[4] H.P. Office. HHS Officially Recognizes Certification Body to Evaluate Electronic Health Records. [Online]. http://www.hhs.gov/news/press/2006pres/20061026a.html

[5] Certification Commission for Health Information Technology. About CCHIT. [Online]. http://www.cchit.org/about

[6] Certification Commission for Health Information Technology. CCHIT Ambulator Criteria. [Online]. http://www.cchit.org/sites/all/files/CCHIT%20Certified%202011%20Ambulatory%20EHR%20Criteria%2020110517.pdf

[7] Certification Commission for Health Information Technology. CCHIT Ambulator Test Script. [Online]. http://www.cchit.org/sites/all/files/CCHIT%20Certified%202011%20Ambulatory%20EHR%20Test%20Script%2020110517.pdf

[8] Certification Commission for Health Information Technology. Self Attestation Guidance. [Online]. http://www.cchit.org/sites/all/files/CCHIT%20Certified%202011%20Self%20Attestation%20Guidance%20v2.06%2020100726_4.pdf

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

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

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

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

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

[14] 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.

[15] 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.

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

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

[18] M. Maghazil, "A comparative analysis of data security in computer-based and paper-based patient record systems from the perceptions of healthcare providers in major hospitals in saudi arabia," The George Washington University, Doctoral Dissertation 2004.

[19] G. Noordende, "Security in the dutch electronic patient record system," in *Second Annual Workshop on Security and Privacy in Medical and Home-care Cystems*, Chicago, 2010, pp. 21-31.

[20] K. Garson and C. Adams, "Security and privacy system architecture for an e-hospital environment," in *7th Symposium on Identity and Trust on the Internet*, Gaithersburg,

2008, pp. 122-130.

[21] 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.

[22] A. Doupe, 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*, Bonn, 2010.

[23] L. Suto, "Analyzing the Effectiveness and Coverage of Web Application Security Scanners," San Francisco, White Paper 2010.

[24] L. Suto, "Analyzing The Accuracy and Time Costs of Web Application Security Scanners," San Francisco, White paper 2010.

[25] 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*, Fukuoka, 2009, p. 804.

[26] 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.

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

[28] D. Geer and J. Harthorne, "Penetration Testing: A Duet," in *18th Annual Computer Security Applications Conference*, Las Vegas, 2002, p. 185.

[29] S. Robinson, "The Art of Penetation Testing," in *IEEE Seminar on Security of Distributed Control Systems*, 2005, p. 71.

[30] OEMR. About OpenEMR. [Online]. http://www.oemr.org/About_OpenEMR

[31] Black Duck Software. (2009, June) Open Source Software Projects in Health Care Offer Significant Cost Savings, Reports Black Duck Software. [Online]. http://www.blackducksoftware.com/news/releases/2009-06-10

[32] The PHP Group. (2011, September) PHP Manual: mysql_escape_string. [Online]. http://php.net/manual/en/function.mysql-escape-string.php

[33] D. Stevens. (2010, March) Escape From PDF. [Online].

http://blog.didierstevens.com/2010/03/29/escape-from-pdf/

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

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

[36] 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.

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