Systematizing Security Test Planning Using Functional Requirements Phrases

Ben Smith, Laurie Williams
North Carolina State University
890 Oval Drive
Raleigh, NC 27695-8206 USA
+1(919) 515-7926
[ben_smith, laurie_williams]@ncsu.edu

ABSTRACT

Security experts use their knowledge to attempt attacks on an application in an exploratory and opportunistic way in a process known as penetration testing. However, building security into a product is the responsibility of the whole team, not just the security experts who are often only involved in the final phases of testing. Through the development of a black box security test plan, software testers who are not necessarily security experts can work proactively with the developers early in the software development lifecycle. The team can then establish how security will be evaluated such that the product can be designed and implemented with security in mind. *The goal of this research is to improve the security of applications by introducing a methodology that uses the software system's requirements specification statements to systematically generate a set of black box security tests.* We propose a methodology for the systematic development of a security test plan based upon the key phrases of functional requirement statements. We used our methodology on a public requirements specification to create 137 tests and executed these tests on five electronic health record systems. The tests revealed 253 successful attacks on these five systems, which are used to manage the clinical records for approximately 59 million patients, collectively. If non-expert testers can surface the more common vulnerabilities present in an application, security experts can attempt more devious, novel attacks.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Verification, Documentation

Keywords

security, testing, verification, application-level, penetration testing, vulnerabilities, requirements, medical records

ISSTA'11, July 17–21, 2011, Toronto, Canada.

1. INTRODUCTION

Security experts use their knowledge to attempt attacks on an application in an exploratory and opportunistic way in a process known as penetration testing. Penetration testing and similar techniques require the security expert's knowledge to be effective [2]. For example, a security tester might browse through a web application, find a form, and submit several attacks to test that the system properly validates input. She will use the knowledge of successful attacks to drive her next attempt. A software tester with no security training would lack the knowledge to pursue defects the same way an expert does.

Due to time and resource constraints, building security into a product must be the responsibility of the whole team, not just the security experts who are often only involved in the final phases of testing [16]. Everyone from the independent test team, to the developer writing unit tests, should be enabled to emulate attacker behavior and work towards security throughout the development process. Through the development of a black box *security* test plan that is based on functional requirements specifications, software testers who are not necessarily security experts can work proactively with the developers early in the software development lifecycle. The team can then establish how security will be evaluated such that the product can be designed and implemented with security in mind. A black box security test plan is fundamentally different than a black box functional test plan. A security test plan focuses on ensuring a malicious user is not able to force unintended functionality in the system [29]. A functional test plan focuses on ensuring intended functionality is present for a benevolent user [5, 23].

We propose a methodology that uses the component parts (key action phrase, key object phrase, and supporting information) in each requirements statement to guide the tester in developing an effective security test. Key action phrases like *display* or *view* signal the need for test cases that attempt to circumvent the system's authorization mechanisms, whereas supporting information like *links* or *external documents* signal the need for test cases involving the injection of a URL that points to a dangerous website. Currently, test case creation using our methodology is manual. In the future, we will develop a tool that uses natural language processing to partially automate this methodology.

The goal of this research is to improve the security of applications by introducing a methodology that uses the software system's requirements specification statements to systematically generate a set of black box security tests. We evaluated our methodology by using a requirements

specification to create a black box security test plan for four open source and one proprietary electronic health record (EHR) systems. We executed the resultant test cases on these five released EHR systems that are currently used to manage the records of over 59 million patients: OpenEMR[1], ProprietaryMed[2], WorldVistA[3], Tolven[4], and PatientOS[5].

The contributions of this paper are as follows:

- To the best of our knowledge, we contribute the first well-defined methodology for systematically developing a black box test plan specifically *focused on security* by using the system's requirements specifications.
- We present five case studies of EHR systems that demonstrate the effectiveness of the methodology in that it uncovers 253 successful attacks in the targeted systems. An expert and a non-expert software tester replicated these attacks. We reported these attacks to the respective development organizations.

The rest of this paper is organized as follows: Section 2 presents background information about software security, testing and requirements. Next, Section 3 presents our methodology for developing security tests from the requirements specifications. Then, Section 4 presents the case study evaluating our methodology, including more detail on our chosen study subjects, the test plan we developed, and the test results for each system. Finally, Section 5 presents our limitations and Section 6 concludes.

2. BACKGROUND AND RELATED WORK

This section reviews the background and work related to our proposed methodology.

2.1 Software Security

Eradicating software vulnerabilities before a product's release is crucial to a software development organization's reputation [28]. Secure software methodologies, such as the Security Development Lifecycle (SDL) [16] and OWASP's Comprehensive Lightweight Application Security Process (CLASP) [11], advocate considering security throughout the lifecycle. These processes include techniques such as the development of security requirements, threat modeling [14, 27], automated static analysis [10], risk analysis and misuse cases [24]. The concept of *building security in* prescribes that developers and testers consider system security from the outset of the project and design the system to be protected from malicious attack [19]. For example, towards the time of the product's release in the SDL, an independent security team must finish a "security push." This team must sign off on the product before its release, closing any unfixed security issues and review the system's threat models to ensure that all possible avenues of attack have been secured [16]. One important aspect of producing secure software is the execution of black box tests related to security, also known as penetration testing [2]. The success of current security assurance techniques that occur late in the product's lifecycle, such as penetration testing, vary based on the skill, knowledge, and experience of testers [2]. Other security techniques resemble "security by checklist" where developers follow a set of guidelines that are not tailored to a specific product [7].

The Common Criteria for IT Security Evaluation (CC) [6] offer a standard language that allows users of information technology products to compare these products and have confidence in their security levels. When a development team has finished creating their product, a licensed, independent evaluator declares the product as certified for a particular protection profile, which applies either to that specific product, or to a range of systems. Most of the evaluations of products that the CC has published apply to operating systems, databases, key management systems, and network devices. Weber [32] has indicated that the CC might be a good starting place for evaluating EHR systems. We agree with that assessment, but contend that developers need software security test cases that are based not only on the CC, but also the product's requirements specification.

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 [19]. McGraw explains that systems have historically had half design flaws and half implementation bugs [19]. We found both design flaws and implementation bugs in the systems we analyzed for this case study.

Threat modeling with STRIDE [14] involves designing a system for security by considering the six major categories of *threats* that a system may encounter: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. STRIDE defines *elements* that a designed system will have that can be vulnerable to the different threat categories: Data Flows, Data Stores, Processes and Interactors (i.e. other systems and people who interact with the system). According to STRIDE, certain types of threats do not make sense with certain types of elements: for example, data flows are not threatened by Spoofing, although the Interactors who may use those data flows are. As such, STRIDE helps the development team to focus on applicable threat-element pairs. Knowing that a data store is threatened by tampering informs the development team that design decisions should be made to help protect the data store from tampering attacks by performing input validation and other preventions or mitigations. As discussed in Section 3.4, we initially based our security test case development methodology on STRIDE before changing to a requirements-based approach.

2.2 Security and Requirements

Before developers can mitigate the risks of security threats, they must know the requirements for the system's security. Security requirements are often *non-functional*, meaning they specify criteria that are used to judge the operation of a system, as opposed to *functional* requirements that define specific functions or behaviors of the system [34]. Focusing on the system's functional *and* non-functional requirements (including security requirements) will reveal the greatest number of vulnerabilities

[1] http://oemr.org/
[2] ProprietaryMed was developed by an organization that wishes to keep the identity of their product confidential.
[3] http://worldvista.org/
[4] http://tolven.org/
[5] http://patientos.org

[6] http://www.commoncriteriaportal.org

with the fewest number of tests by providing a thorough coverage of the functionality of the system. Several techniques have been constructed for developing and analyzing adequate security requirements [12, 15], improving the traceability of non-functional requirements to help maintain critical system qualities throughout a system's lifetime [9], and developing functional requirements that have security in mind [20].

2.3 Security Testing

The current software development paradigm includes a list of testing strategies to ensure the correctness of an application in functionality and usability as indicated by a requirements specification [5, 23]. With respect to intended correctness, verification typically entails creating test cases designed to discover faults by causing failures. Oracles tell us what the system should do and errors tell us that the system does not do what it is supposed to do. Software *security* testing, however, entails that we validate not only that the system does what it should securely, but also that the system does not do what it is not intended to do [29].

To illustrate the difference, consider the following requirement: "The system shall provide the ability to send 250 character messages between users." The traditional black box testing would result in a test plan that tests several variations on messages sent: trying a zero-length message, trying a message that is too long, sending the message to a non-existent user, attempting to send with no database, etc. Software *security* testing entails trying to turn the message sending functionality into a spamming mechanism, sending malicious links to users within the system, impersonating a different user, and so on. This unintended functionality is not often found in the requirements document unless the team has performed an explicit set of misuse cases or an anti-goal analysis of the system [15].

2.4 Black Box Testing and Requirements

As Beizer indicates, "…The entire requirements list is essential. No magic here. No avoiding it either. The whole list [of requirements] must be tested in detail, or the requirement doesn't work" [4]. Our methodology puts Beizer's philosophy into action: as Section 3.4 indicates, we recommend inspecting each requirement and eliciting any resultant test cases.

Bach indicates that testing should be considered not only as an evaluative process, but also as a means of exploring the meaning and implications of requirements [3]. Bach also explains that test case traceability to requirements is only meaningful if the development team understands *how* the test cases are related to the requirements specification. Bach indicates that the most important value that a test case can offer is that it can force developers to discuss and explore their unstated assumptions about the product's requirements with the test team. Each functional requirement can elicit some form of security concern that our methodology creates a test case to verify.

Whalen and Rajan propose a coverage metric for black box testing that is based on the requirements specification [33]. To determine the value of the metric for a given set of requirements and a black box test plan, the requirements specification must first be translated into a model specification using linear time temporal logic. Whalen and Rajan used their coverage metric in a case study on a flight guidance system and generated a set of requirements-based black box tests that achieved 100% coverage of the requirements. Our methodology may not help achieve 100% requirements coverage using Whalen and Rajan's metric, since our methodology focuses on testing what the system *should not* do and some requirements may not result in security test cases (see Section 3.1).

Martin and Melnik explain that requirements and tests can become indistinguishable, so that development teams can specify behavior by writing tests and then verify that behavior by executing the tests [17]. Martin and Melnik argue for the writing of early acceptance tests as a requirements engineering technique. We view our methodology as an extension of the idea introduced by Martin and Melnik, in the sense that requirements indirectly tell us what the system should not do.

3. METHODOLOGY

Several techniques exist for evaluating the security of software systems today: security requirements [12], misuse cases [24], threat modeling [14, 27], automated static analysis [10], and penetration testing [2]. Each of these techniques plays a role in the prevention and removal of vulnerabilities, but none of these techniques will find every vulnerability. This section provides our methodology for developing software security tests at the application level based on a functional requirements specification. Black box security testing's role is to provide a security evaluation of the product in its environment. Black box testing techniques like penetration testing can uncover vulnerabilities that are dependent on environmental specifics that other forms of testing cannot [26]. Our methodology can augment other existing techniques to additionally allow software testers who are not experts in security to participate in black box security testing. Table 1 presents four examples of functional requirements we will refer to throughout this section.

3.1 Overview

The structure of the requirements statement, as well as certain keywords, can help guide the tester to construct an appropriate type of test. The CWE/SANS Top 25[7] lists the most dangerous security programming errors based on prevalence and potential consequences. Our methodology includes six test types, each of which can uncover one or more common vulnerabilities from the Top 25.

Traditionally, functional requirements statements specify desired system behavior in "shall" statements [34], for example: "The system *shall* send an email message to the administrator containing the new user name and the time and date of creation when a new account is created." We demonstrate the methodology in this section using traditional functional requirements, but our methodology does not rely on requirements to be provided in "shall" format: as long as the key phrases can be identified, our methodology is applicable.

Our methodology uses *key phrases* and *supporting information* in a requirements statement to determine the type of security test that will most likely reveal vulnerabilities in the system. In these examples, the first phrase that the tester comes to after reading "The system shall provide the ability to…" contains the key action phrase and is followed by the key object phrase. We call these phrases *key* because they define the functionality the system has with respect to its environment.

[7] http://cwe.mitre.org/top25

Table 1. Example Requirements Specification Statements (from [1])

ID	Description	Results in Test Case Types (see Section 4.1)
AM 09.03	The system shall provide the ability to save scanned documents as images.	Force Exposure, Malicious File
AM 02.04	The system shall provide the ability to modify demographic information about the patient.	Force Exposure, Input Validation Vulnerability, Audit
AM 02.05	The system shall store demographic information in the patient medical record in separate discrete data fields, such that data extraction tools can retrieve these data.	*None* (due to AM 02.04)
AM 10.04	The system shall have the ability to provide access to patient-specific test and procedure instructions that can be modified by the physician or health organization. These instructions may reside within the system or be provided through links to external sources.	Force Exposure, Input Validation Vulnerability, Dangerous URL
SC 03.11	When passwords are used, the system shall use either standards-based encryption, e.g., 3DES, AES, or standards-based hashing, e.g. SHA1 to store or transport passwords.	Malicious Use of Security Functions

As seen in Table 1, requirements specifications like these typically conform to the following format: "*The system shall provide the ability to <action> a <object> <and/with/in supporting information>*." The object in these statements is most often a data store, such as a listing of users or a report regarding multiple data records for output. The action in these statements is typically an action that the system will perform on that data store, such as store, graph, view, print, or edit. The supporting information in these statements provides additional information as to how, or when the system should achieve the action. Sometimes the supporting information is a prepositional phrase in the same sentence or can extend to an additional sentence. For a detailed example of parsing requirements statements into key phrases, see Section 3.5.

3.2 Using Keywords to Identify Appropriate Test Types

This section provides additional details on which keywords elicit which type of security tests. Throughout this section, we will refer to the six test type templates defined in Section 3.3 in title case (e.g. "Audit Test types").

*3.2.1 Using **Action Phrases***

The action phrase in a requirement specification provides information about the direction the data is flowing. When using actions to elicit security tests, the object in the requirements statement indicates the target of the attack.

Actions like *record*, *enter*, *update*, *store*, or *edit* typically indicate that the data in question is flowing from user input into the system's persistent data stores. This means that Input Validation Vulnerability Test types are required. The action in AM 02.04 is *modify*, which means that the system is receiving information from the user about the patient's demographic information. One or more fields in the user interface for entering demographic information may provide an opportunity for an attacker to inject malicious input that can result in a cross-site scripting [31], buffer overflow, or SQL injection [13] in one of the fields of this data store.

Actions like *display*, *view*, *print*, or *search* typically indicate that the data in the requirements statement is flowing from the system's persistent data stores to the user interface, as in AM 10.04, where the action is *provide access to*. These requirements statements indicate that the tester should elicit a Force Exposure test. A user must be authenticated and authorized to view data or user interfaces before any access to the system's internal data is allowed.

Actions like *record*, *enter*, *update*, *store* or *edit* also necessitate a Force Exposure test case because the attacker could try to access the user interface for making changes to the data store in question. For example, based on AM 02.04, the tester knows that the system will contain some form of user interface for modifying patient demographics, and this user interface must also be protected from unauthorized access.

Action phrases like *protect*, *prevent*, or *enforce* can help a tester know that he or she is dealing with a security requirement and thus can elicit a Malicious Use of Security Features test. A tester should assert the system's ability to uphold security requirements by attempting to break or misuse the security mechanisms that are described in the requirements statements. Also, just because a security requirement is in place does not mean that the system implements that security requirement, and so a tester should elicit a test case that directly asserts that the security mechanism exists. For example, the corresponding test case for SC 03.11 should test for the possibility that a password is not stored in hashed format, and should try to sniff the password out of either an HTTP request, or by looking at the database table for user credentials.

*3.2.2 Using **Object Phrases***

The object phrase in certain requirements statements tells the tester the nature of the data store that may be modified or viewed according to the action. When the object is something like *demographics* or *reminders for disease management*, the tester must assume these data are stored as text or an element within the system's data stores. If the object indicates a data store, then the tester can elicit an Input Validation Vulnerability Test. The fact that these object phrases also point to an Input Validation Vulnerability Test is an example of a frequent occurrence in our methodology. Sometimes two key phrases or the supporting information in a requirement statement can both point to the same test type.

In other instances, the object may be a file or a URL. For example, the object in AM 09.03 is *scanned documents as images*. Object phrases like *documents*, *files*, *forms*, and *images* elicit a Malicious File test.

The system may need to be responsible for recording insertions, deletions, and edits of sensitive information. Keywords such as *credit card numbers*, *personal identification information*, *GPAs*, and *personal correspondence* in any part of the requirements statement all indicate confidential information that a user would not want exposed to the world. Requirements that deal with viewing or editing any protected system information necessitate an Audit Test case. In AM 10.04, the fact that a healthcare provider has accessed specific test and procedure instructions can be crucial information when it comes to the determination of medical malpractice suits, and as such the system should keep a record of what a given physician or health organization has seen or not seen, as well as changed or not changed.

*3.2.3 Using **Supporting Information***

The supporting information the requirements statement provides can also give some information on the type of security test that a tester can elicit. As described in Section 3.2.1, since AM 10.04 has the verb *provide access to*, the tester can elicit a Force Exposure test case to try and gain unauthorized access to the medical test and procedure instructions mentioned in AM 10.04. However, because the requirements statement also indicates that the information may be provided through links to external resources, the tester can elicit a Dangerous URL test case. A tester cannot assume that links that any user in the system can edit will always point to safe web sites.

*3.2.4 **Exception**: Redundant Requirements*

Note that not every requirement results in an elicited security test type. For example, AM 02.04 mandates the editing and display of patient demographic data, and AM 02.05 provides more information to the tester regarding the nature of the patient's demographic data. The tester, when using the rules described in Section 3.2 would produce an Input Validation Vulnerability test, a Force Exposure test, and a Audit test based on AM 02.04. Requirement AM 02.05 would not produce any additional security test cases.

3.3 Test Types

Section 3.3.1 through 3.3.6 describe the six types of test cases that we have developed. Within the description of each test type in this section, we will provide an explanation of test type's intent and purpose. We also include a reformatted version of the test case type's template, which shows the tester how to use the strategy behind the test to attack direct object in question for the requirement under evaluation.

We chose these six test types based on the CWE/SANS Top 25, which lists the most dangerous security programming errors based on prevalence and potential consequences. We captured a set of test types that would target all the vulnerability types on the Top 25 as well as the 23 vulnerability types that CWE lists as being "on the cusp". We call this combined set of vulnerability types the "Top 25+". For a given system, the CWE/SANS Top 25+ may not uncover every security vulnerability, but we targeted the Top 25+ because they were chosen based on their prevalence among actual reported vulnerabilities. In the interest of exposing the greatest number of vulnerabilities with the fewest number of test cases [8], creating test cases that target the most frequently occurring vulnerabilities is a prudent strategy.

3.3.1 Input Validation Vulnerability Tests

Keywords: Record, Enter, Update, Create, Capture, Store, Edit, Modify, Specify, Indicate, Maintain, Customize, Query, Receive, Search, Produce

Targeted CWE/SANS Top 25+ Errors: Cross-site Scripting, SQL Injection, Classic Buffer Overflow, Path Traversal, OS Command Injection, Buffer Access with Incorrect Length Value, PHP File Inclusion, Improper Validation of Array Index, Information Exposure Through an Error Message, Integer Overflow or Wraparound, Incorrect Calculation of Buffer Size, Race Condition

Example requirement: AM 02.04 - The system shall provide the ability to modify demographic information about the patient.

Explanation: This set of test scripts is based on the idea that any input that the system obtains from any source should be sanitized before either being used or stored in the system's data stores. The tester should inject malicious input from our attack list[8] to see how the system handles input validation attacks. Unsanitized input can be used to cause several common attacks such as cross-site scripting, buffer overflow, and SQL injection.

Template (abbreviated) [9]: Authenticate as a registered user and inject malicious input from a predefined set of attack strings into the relevant field. The attack strings should be neutralized before insertion, or the input should be rejected. The data store in question should remain in tact. No error messages should occur that reveal sensitive information about the system's configuration or architecture.

3.3.2 Force Exposure Tests

Keywords: Record, Enter, Update, Create, Capture, Store, Edit, Modify, Specify, Indicate, Maintain, Customize, Query, Receive, Search, Display, View, Print, Graph, Indicate, Provide Access To, Produce, Make Available, Filter, Order

Targeted CWE/SANS Top 25+ Errors: Improper Access Control, Reliance on Untrusted Inputs in a Security Decision, Use of Hard-coded Credentials, Missing Authentication for Critical Function, Incorrect Permission Assignment for Critical Resource

Example requirement: AM 08.11 - The system shall provide the ability to filter, search or order notes by the provider who finalized the note.

Explanation: This set of test scripts is based on the idea that a user should be authenticated and authorized before they are able to interact with the protected information contained within the system. Authentication alone is not enough to ensure the protection of sensitive system data. An attacker will try to access web pages in browser histories, or to directly force the exposure of a user interface screen by guessing the series of steps required to get there.

---

[8] This attack list was obtained from http://neurofuzz.com.

[9] The templates have been formatted differently in the interest of space. A more thorough listing of templates can be found at: http://realsearch.com/healthcare/doku.php?id=public:security_evaluation_plan

Template: Authenticate as a registered user and access the user interface for performing the action in the requirement. Record the steps that were required to perform this action. Log off and attempt to repeat the recorded steps. The interface should be inaccessible, and the user should be denied access.

3.3.3 Malicious File Tests

Keywords: File, Save, Upload, Receive, Image, Document, Scanned

Targeted CWE/SANS Top 25+ Errors: Unrestricted Upload of File with Dangerous Type, Download of Code Without Integrity Check

Example requirement: AM 09.01 - The system shall provide the ability to capture and store external documents.

Explanation: This set of test scripts is based on the idea that uploaded files, when not scanned for viruses or protected by file type filters, can contain malicious code that can be used to exploit the system, produce a Denial of Service attack, or to send revealing information back to the attacker.

Template: Authenticate as a registered user and access the user interface for storing a file. Select and upload a malicious file. View or download the malicious file. The file should be rejected upon selection or should not be allowed to be stored.

3.3.4 Malicious Use of Security Functions Tests

Keywords: Protect, Enforce, Prevent, Authorized, Detect, Authenticate, Allowed, Support, Prohibit, Password, Require, Allow, Encryption

Targeted CWE/SANS Top 25+ Errors: Could apply to any, depending on what the security feature is meant to prevent.

Example requirement: SC 03.01 - When passwords are used, the system shall support the ability to protect passwords when transported or stored through the use of cryptographic-hashing with SHA1, SHA 256 or their successors and/or cryptographic encryption with Triple Data Encryption Standard (3DES), Advanced Encryption Standard (AES) or their successors.

Explanation: This set of test scripts is based on the idea that having security functions (e.g. authentication, encryption, or password-hashing) is not enough to ensure that a system is secure. Even the most stringent use of standard security practices can still leave critical vulnerabilities in a system. Encryption algorithms fail, authentication mechanisms can be broken, and one way to ensure these security functions work correctly is to "think like an attacker" and try to break them.

Template: There is no template for this test type. The pattern for these tests is to break the security mechanism that the security requirement describes or to test to see that the security mechanism actually fulfills the functions it was designed to fulfill.

3.3.5 Dangerous URL Tests

Keywords: Links, External resource, URLs, Addresses, External documents

Targeted CWE/SANS Top 25+ Errors: Open Redirect, Cross-Site Request Forgery

Example requirement: AM 10.03 - The system shall provide the ability to provide access to patient-specific test and procedure instructions that can be modified by the physician or health organization; these instructions are to be given to the patient. These instructions may reside within the system or may be provided through links to external sources.

Explanation: This set of test scripts is based on the idea that for places where the requirements specify that the user can insert a link to external reference, attackers can insert links to dangerous websites that may contain pop-ups, spyware, adware, or malware for the user to download. The system should not allow the user to store (either intentionally or unintentionally) a link that goes to a dangerous website.

Template: Authenticate as a registered user. Open the user interface for inserting a URL that links to an external resource. Insert a URL that points to a known malicious or dangerous website. The link should be rejected as malicious. An error message should indicate to the user that the link points to a dangerous website.

3.3.6 Audit Tests

Keywords: patient record, demographics, credit card information, GPA, personal identification information

Targeted CWE/SANS Top 25+ Errors: None. *Insufficient Logging* is the CWE classification for vulnerabilities that these test cases can expose.

Example requirement: AM 06.01 - The system shall provide the ability to capture, store, display and manage patient history.

Explanation: The ability of the system to track what happened and when helps keep a clear record of provenance for the records the system maintains. This is also useful for verifying their authenticity, and protecting the people who depend on them. What this means, in effect, is that any change to the data stores within the system as well as for the administration of the security features of the system should be recorded. If something is added to the system's data and then deleted, both the *add* and the *delete* operations should be recorded and auditable.

Template: Authenticate as a registered user and access the user interface for performing the action in the requirement. Logout as the registered user. Login as the system administrator. The audit records should show that the registered user performed the action. The audit record should be clearly readable and easily accessible.

3.4 Security Test Plan Creation Methodology

To create the black box security test cases, a tester should execute the following steps:

1. Examine the next (or first) requirements statement from the requirements specification document. If there are no more statements to evaluate, then stop.
2. Break this requirements statement into its phrases using the procedure described in Section 3.1 and list each key action phrase, key object phrase, and supporting information separately.
3. Use the mappings provided in Section 3.3 to determine the type of security test case to elicit based on the key phrases provided in the phrases from Step 2.
4. Check if the new test case would be unnecessary due to an existing test case (see Section 3.1). If the new test would be unnecessary, return to Step 1.

5. For each type of security test case, substitute the phrases from the requirements statement into the test case type template and add the test case to the test plan.
6. The test case receives a prefix according to its type, as listed in Section 3.3. Append an integer number (counting numbers are acceptable) to this prefix to form the test's unique identifier. For example, the third test that was created to test for Dangerous URLs is called DU3.
7. Return to Step 1.

The initial version of our methodology was based on STRIDE threat modeling (see Section 2.1). However, the threat categories of STRIDE [14] do not map directly to test types because a single test type can lead to multiple threats. For example, an Input Validation Vulnerability Test can be used to Spoof, Tampering, Information Disclosure, Denial of Service, and Elevate one's Privilege.

3.5 Example Application of Methodology

In this section, we present an example with a single requirement through to a test case. Consider AM 02.04 in Table 1: "The system shall provide the ability to modify demographic information about the patient." This requirement can be broken down as follows:

- **Key Action Phrase:** modify
- **Key Object Phrase:** demographic information about the patient
- **Supporting Information:** *none*

As shown in Table 1, AM 02.04 signals the need for Force Exposure, Input Validation Vulnerability, and Audit test case types. In this example, we discuss the Input Validation Test Case type in detail. In AM 02.04, the phrase *modify* is the key action phrase. This key action phrase indicates that an attacker has the opportunity to input malicious strings that can take the form of a cross-site scripting [31], SQL injection [13] or many other input validation vulnerabilities. These attacks, if properly executed, have the potential to tamper with or reveal information from the *demographic information* object. Input Validation Tests, which our methodology includes, will attempt to tamper with or reveal information from the demographic information object. We provide the resultant Input Validation test case, developed by analyzing AM 02.04 here.

Based on Requirement: AM 02.04.

Assumptions: The system has been pre-initialized with the reusable set of test data.

Description:

1. Authenticate as Dr. Robert Alexander.
2. Open the user interface for entering patient information and create a new patient.
3. Inject one random attack from the attack list into a field of the demographic information.
4. Repeat the previous step for five attacks from the attack list.
5. Repeat the previous two steps for five fields from the patient demographic information.

Expected Results:

- The attack strings should be neutralized or sanitized before insertion, or the attack strings should be rejected and the user gracefully informed that their input is invalid.
- The data store for the demographic information should remain in tact.
- Data should not be revealed that unless it is a part of this patient's demographic information.
- No error messages should occur that reveal sensitive information about the system's configuration.

4. EVALUATION

This section describes the procedure we used to conduct the evaluation of our methodology. We evaluated our methodology by using a requirements specification to create a black box security test plan for four open source and one proprietary electronic health record (EHR) systems. We then executed the test plan to evaluate its ability to reveal vulnerabilities.

4.1 Requirements: CCHIT Criteria

In 2006, through a consensus-based process that engaged stakeholders, the Certification Commission of Healthcare IT (CCHIT)[10] defined certification criteria focused on the functional capabilities that should be included in ambulatory (outpatient) and inpatient EHR systems [21, 22, 25]. We consider the CCHIT certification criteria as a functional requirements specification because all statements in the certification criteria take the form of "shall" statements and because the individual criteria express behavior that an EHR must exhibit in order to be certified [6]. The 286 CCHIT ambulatory certification criteria primarily relate to the functional capability that must be present in an EHR system (see [1]). The criteria are categorized into different areas of functionality, such as ambulatory (numbered AM), interoperability (IO-AM), and security (SC). For instance, in Table 1, the criteria AM 09.03, AM 02.04 and AM 10.04 are certification criteria for ambulatory EHR systems, and criteria SC 03.11 applies to the security of the EHR system under evaluation. Most requirements that pertain to security in the CCHIT certification criteria are prefixed with an SC, but others are not. For example, AM 36.05 states: "The system shall provide the ability to prevent specified user(s) from accessing a designated patient's chart." The currently existing security criteria primarily focus on features like encryption, hashing, and passwords.

4.2 Evaluated Systems

We chose five EHR systems for our study that are responsible for managing the records for over 59 million patients. EHR systems provide a good test bed for evaluating our methodology because all five systems implement the same functional requirements, meaning we could evaluate our resultant test plan multiple times. Table 3 presents a summary of the facts for each study subject.

- **OpenEMR** is an open source EHR system with at least 20 companies providing commercial support within the United States[11]. OpenEMR is actively pursuing CCHIT certification[12]. The accessibility of the source code for

[10] http://www.cchit.org

[11] http://www.openmedsoftware.org/wiki/OpenEMR_Commercial_Help

[12] http://www.openmedsoftware.org/wiki/OpenEMR_Certification

Table 3. Summary of the Study Subjects

System	Version / Release Date	Language / Platform	Install Base / Usage	LoC / Files	License	# Contributors	Estimated Records (Patients)
OpenEMR	3.2 / February 16th, 2010	PHP / web-based	1,563 downloads /mo.[14]	305,944 / 1,643[15]	GPL[16]	34 [17]	31 million[18]
ProprietaryMed	1.0 / March 31st, 2010	ASP.NET / web-based	17 physician practices	120,000 / 900	Proprietary	12	30,000
Astronaut WorldVistA	0.9.9.6 / April 30th, 2010	MUMPS / thin-client	529 downloads / mo.	1,646,655 / 25,474	GPL	~37	28 million
Tolven	RC1 / May 28th, 2010	Java / web-based	151 downloads / mo.	466,538 / 4,169	LGPL	12	10 million
PatientOS	0.981 / November 15th, 2009	Java / thin-client	1492 downloads / mo.	478,547 / 2,828	GPL	6	n/a

OpenEMR, as well as its active contributing open source community makes the application an ideal candidate for our evaluation.

- **ProprietaryMed** is a web-based EHR created for use in primary care practices. The development team that developed ProprietaryMed would like to keep the product's name confidential. ProprietaryMed is a strong candidate for our evaluation because of its contrast with the other open source projects. PropreitaryMed is closed-source, is a paid product, and uses a different architecture of frameworks than the other projects.
- **VistA** was developed in the 1970's using the MUMPS programming language. VistA is comprised of several key modules, including the VistA server, the command-line interface, and the CPRS (Computerized Patient Record System). The CPRS acts as a thin-client or desktop application graphical user interface for interacting with the VistA server. There are more than 68 private-sector healthcare facilities in the United States that are running various configurations of VistA as their EHR system. Additionally, there are more than 1,607 federal government installations of VistA that manage over 24 million patient records [30]. According to its developers, the open source contributors to VistA are difficult to count, since VistA does not use typical version control mechanisms, but the number of known developers for WorldVistA alone is more than 37.
- **Tolven** is a recently developed health care platform that includes a clinical record system and a personal health record system. Currently, 30 clients worldwide are evaluating Tolven [18]. The developers of Tolven have indicated that there are a number of deployments of Tolven underway in the US, Europe and Asia at the moment that should support in the excess of 10 million patient records.
- **PatientOS** is an open source EHR application that uses a thin-client for viewing patient data. We contacted the developers of PatientOS for an estimate of the number of patient records are currently being managed using their product and have yet to receive a reply.

4.3 Test Case Development and Execution

We used our methodology on the 284 CCHIT functional requirements statements and created 137 tests, which can be found on our healthcare wiki[13]. An undergraduate student with minimal security experience also executed the test plan on our study subjects and achieved the similar results, indicating that non-expert software testers can use the test plan. Table 4 lists the test case results for each of the case study subjects described in Section 4.2. We use the following legend to help describe the results:

- **Pass:** The system met the test case's specified preconditions, and the actual results matched the expected results. The test case did not reveal any security issue.
- **Fail:** The system met the test case's specified preconditions, but one or more results did not match the expected results. The test case revealed a security issue.
- **PNM:** Precondition not met. We could not execute the test case due to constraints in the system's configuration or setup, or perhaps because the test case makes an assumption about the system that simply is not true.
- **N/A:** The test case could not be executed because we could not find the functionality specified in the requirements. These systems are not CCHIT-certified, with the exception of Astronaut WorldVistA, and so a missing requirement is understandable.

We consider PNM results as providing flexibility for the test plan to cover potential vulnerabilities that may have an opportunity to exist in some systems but not for others. For example, test SF10, available on the healthcare evaluation wiki, asserts that the tester should attempt unencrypted HTTP (i.e. not HTTPS/SSL connections) access to the EHR system if the

[13] http://realsearchgroup.com/healthcare/doku.php?id=public:requirements_based_security_evaluation

[14] https://sourceforge.net/projects/openemr/files/stats/timeline

[15] Calculated using CLOC v1.08, http://cloc.sourceforge.net

[16] http://www.gnu.org/licenses/gpl.html

[17] http://sourceforge.net/project/memberlist.php?group_id=60081

[18] http://www.openmedsoftware.org/wiki/Open_Source_EHR_in_Practice

Table 4. Test Results for the Five Case Study Subjects

	Type	Input Validation Vuln.	Malicious File	Dangerous URL	Force Exposure	Security Features	Audit	Total
	Prefix	**IV**	**MF**	**DU**	**FE**	**SF**	**AU**	
OpenEMR	**Pass**	0	0	0	17	3	3	**23**
	Fail	16	2	0	0	8	37	**63**
	N/A	13	2	4	10	0	18	**47**
	PNM	1	1	0	1	1	0	**4**
Proprietary Med	**Pass**	7	0	0	17	6	6	**36**
	Fail	5	5	1	0	3	42	**56**
	N/A	10	0	3	11	1	10	**35**
	PNM	8	0	0	0	2	0	**10**
Astronaut WorldVistA	**Pass**	13	0	0	28	3	4	**51**
	Fail	6	0	3	0	5	46	**58**
	N/A	8	0	1	0	2	7	**17**
	PNM	3	5	0	0	2	1	**11**
Tolven	**Pass**	10	0	0	12	3	0	**25**
	Fail	1	1	0	0	5	29	**37**
	N/A	17	0	4	15	2	27	**65**
	PNM	2	4	0	1	1	2	**10**
PatientOS	**Pass**	14	0	0	14	1	1	**30**
	Fail	0	0	1	0	3	33	**37**
	N/A	13	5	3	12	1	18	**52**
	PNM	3	0	0	2	7	6	**18**
Overall (totals across all five systems)	**Pass**	45	0	0	88	16	13	**162**
	Fail	26	8	5	0	25	189	**253***
	N/A	62	7	15	48	6	79	**217**
	PNM	17	10	0	4	13	9	**53**
Total		**150**	**25**	**20**	**140**	**60**	**290**	**685**

system is a web application. When the system is not configured to allow web access, as in the case of our installation of Astronaut WorldVistA[19], test SF10 received the result PNM. This logic allows us to enable our test plan to include the testing for unencrypted HTTP access for the other three web applications, OpenEMR, ProprietaryMed, and Tolven.

Test cases of type N/A should be considered as allowing us to evaluate the completeness of an EHR system. The test case should exist in the test plan for each and every requirement that is possible, regardless of whether the system implements the requirement. For example, IV24 states that the tester should assign a task to a user in the EHR system and insert an attack string for the description of the task. When we executed this test case on OpenEMR, we found no user interface for assigning a task to another user. We searched OpenEMR's user manuals and found no reference to task assignment. As such, we assume that OpenEMR fails CCHIT requirement AM 24.01, which requires the system to be capable of assigning messages, and test IV24 received a result of N/A for OpenEMR.

[19] Some installations of WorldVistA allow the configuration of web-based access to the VistA server for the manipulation of EHRs. We chose not to enable this configuration to help contrast VistA with the other subjects in our case study and demonstrate that our test plan could function well on a thin client-based system.

The final section of Table 4 lists the overall results, which is a sum of the test results for the five study subjects. That is, the cell for PNM with a *fail* test result is calculated by adding the number of *fail* test results for all five systems for PNM test case types. These data can also be found on the healthcare wiki, including a detailed list of each test case's actual results in addition to its summary.

Overall, our test plan launched 253 (see the * in Table 4) successful attacks in the five EHR systems that consisted of both implementation-level defects, such as cross-site scripting, and design-level issues, such as the lack of encryption on the backup copy of system data. Audit test case failures were the most prevalent, with the five study subjects failing 67% of these test cases overall. Malicious Use of Security Function test cases were the next most prevalent at 46%. Dangerous URL test cases were most likely to result in an N/A result at 75%. Malicious File test cases were the most likely to result in a PNM result at 50%. Our study subjects passed 66% of the Force Exposure test cases and failed 0% of them. The remaining test cases in the Force Exposure category resulted in a PNM or N/A result, meaning the functionality an attacker would try to expose does not exist in the system, either because the system failed to meet the requirement, or because of some issue with the way the system was configured. Perhaps Force Exposure test cases are not the best test case for revealing vulnerabilities.

We developed the security test plan in approximately 60 person hours. Executing the test plan manually on each of the case study subjects consumed approximately six to eight person hours per project. We also took time to alert developers to the vulnerabilities we found by posting respective healthcare IT communities' bug report pages.

5. LIMITATIONS

This section presents the limitations of this paper.

5.1 Methodology

Attackers often use functions, procedures, or interfaces in their target systems that are not specified by the requirements. Even if a system passes all of the test cases elicited using our methodology, the system can still exhibit software vulnerabilities. No fault or vulnerability detection technique can identify every problem with a complex, industrial-scale software system, and our methodology is no exception to this rule. Specifically, developers often use utility functions or auxiliary technologies to enable the functionalities of the systems they develop, and attackers have been known to exploit the holes left open by these components.

Since our test plan is confined to what is described in the requirements, a tester would have no way of knowing about holes that may be exposed by auxiliary technologies or third-party components. Other techniques, such as static analysis and traditional penetration testing are more tuned to identifying and removing these types of defects. Also, the test plan developed using our methodology is only as good as the requirements specification used to develop it. Many software systems do not even have an explicit requirements specification and still others have a requirements specification that is vague or unclear. Future work can examine how our methodology would perform in these instances.

The CCHIT ambulatory certification criteria may not be representative of requirements specification statements for other domains, and using our methodology on these specifications—even if they are precise and clear—may prove to be infeasible. There are other types of security tests that could be elicited from requirements specifications. We chose to develop these test types and their templates based on the CWE/SANS Top 25+ to maximize the amount of potential vulnerabilities that test plans written using our methodology could discover, but different architectures offer different security challenges.

5.2 Case Study

Our results may only apply to open source or non-industrial systems in health care. Some of the test results may have been different given a different test environment for each of the systems we evaluated. We often configured these study subjects in the simplest way possible, to maximize the efficiency of our evaluation. Some of the systems' documentation suggests that with an unknown amount of setup time, these systems may be capable of achieving more of the requirements, thus producing not as many N/A or PNM results.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented a methodology that uses a software system's functional requirements specification to formulate a set of security test cases that assesses the system's ability to protect itself and its data from malicious intruders. We evaluated this methodology by creating 137 test cases based on the 284 CCHIT ambulatory certification criteria and running the test plan on five EHR systems. We discovered 253 individual security vulnerabilities in five released applications. These vulnerabilities ranged from cross-site scripting attacks, to phishing attempts, to the ability to upload a dangerous file, to the ability to impersonate another user. These vulnerabilities could be catastrophic with respect to the objective of protecting patients' medical records.

In future work, we will evaluate the ability of our methodology to perform in other domains, with different types of requirements specifications. Additionally, analyzing requirements statements may reveal missing or ambiguous issues to the requirements engineers. Future work will include an investigation and analysis of our methodology in conjunction with the requirements engineering process. Finally, we plan to develop a tool that employs natural language parsing to partially automate this methodology.

Secure software development should continue to make the use of security activities at every stage of software development to build security in. Software testers can augment existing security development processes by using our methodology on any system that has a functional requirements specification. The requirements can help develop a set of security test cases before the system is written that can be executed in a black box test plan to reveal mission-critical vulnerabilities that emanate from the established functionality of the software system. The development team can benefit from a *pattern-based* approach to codifying current security knowledge (i.e. the CWE/SANS Top 25+) into black box tests, so that experts and non-experts alike can test attacks in a systematic fashion.

7. ACKNOWLEDGMENTS

We would like to thank the North Carolina State University Realsearch group for their helpful comments on the paper. The National Science Foundation under CAREER Grant No. 0346903 supports this work. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Additionally, this work is supported by the United States Agency for Healthcare Research Quality.

8. REFERENCES

[1] *CCHIT Certified 2011 Ambulatory EHR Certification Criteria*, The Certification Commission for Health Information Technology, http://www.cchit.org/sites/all/files/CCHIT%20Certified%202011%20Ambulatory%20EHR%20Criteria%2020100326.pdf, 2010.

[2] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy,* vol. 3, no. 1, pp. 84-87, 2005.

[3] J. Bach, "Risk and Requirements-Based Testing," *IEEE Computer Society Press,* vol. 32, no. 6, pp. 113-114, 1999.

[4] B. Beizer, *Black-Box Testing*, Chichester: J. Wiley, 1995.

[5] B. Beizer, *Softwar Testing Techniques. 2nd Edition.*, London: International Thomson Compute Press, 1990.

[6] K. M. Bell, "A Statement from Karen M. Bell, M.D., Chair, Certification Commission for Health Information Technology. Press Release. http://www.cchit.org/media/news/2010/06/statement-karen-m-bell-md-chair-certification-commission-health-information-technology," 2010.

[7] S. Bellovin, “Security by checklist,” *IEEE Security and Privacy,* vol. 6, no. 2, pp. 88, 2008.
[8] A. Bertolino, “Software Testing Research: Achievements, Challenges, Dreams,” in International Conference on Software Engineering (ICSE 07), Minneapolis, MN, 2007, pp. 85-103.
[9] J. Cleland-Huang, “Toward improved traceability of non-functional requirements,” in International workshop on Traceability in emerging forms of software engineering, Long Beach, California, 2005, pp. 14-19.
[10] D. Evans, and D. Larochelle, “Improving Security Using Extensible Lightweight Static Analysis,” *IEEE Software,* vol. 19, no. 1, pp. 42-51, 2002.
[11] J. Gregoire, K. Buyens, B. D. Win *et al.*, “On the Secure Software Development Process: CLASP and SDL Compared,” in International Conference on Software Engineering. Software Engineering for Secure Systems (SESS 2007), Minneapolis, MN, USA, 2007, pp. 1-7.
[12] C. B. Haley, R. Laney, J. D. Moffett *et al.*, “Security Requirements Elicitation: A Framework for Representation and Analysis,” *IEEE Transactions of Software Engineering,* vol. 34, no. 1, pp. 133-153, 2008.
[13] W. Halfond, and A. Orso, “AMNESIA: Analysis and monitoring for NEutralizing SQL injection attacks,” in International Conference on Automated Software Engineering, Long Beach, CA, 2005, pp. 174-183.
[14] S. Hernan, S. Lambert, T. Ostwald *et al.*, "Uncover Security Design Flaws Using the STRIDE Approach," *MSDN Magazine*, http://msdn.microsoft.com/en-us/magazine/cc163519.aspx, 2006].
[15] A. v. Lamsweerde, “Elaborating Security Requirements by Construction of Intentional Anti-Models,” in International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, 2004, pp. 148-157.
[16] S. Lipner, “The Trustworthy Computing Security Development Lifecycle,” in 20th Computer Security Applications Conference, Tuscon, Arizona, 2004, pp. 2-13.
[17] R. C. Martin, and G. Melnik, “Test and Requirements, Requirements and Tests: A Möbius Strip,” *IEEE Software,* vol. 25, no. 1, pp. 54-59, 2008.
[18] R. A. Marudo, *VistA & Open Healthcare News. http://www.tolvenhealth.com/documents/VistANews2008Jan-Feb.pdf*, 2008.
[19] G. McGraw, *Software Security: Building Security In*: Addison-Wesley Professional, 2006.
[20] N. R. Mead, and T. Stehney, “Security quality requirements engineering (SQUARE) methodology,” in Software Engineering for Secure Systems (SESS), St. Louise, Missouri, USA, 2005, pp. 1-7.
[21] H. P. Office. "HHS Announces Project to Help 3.6 Million Consumers Reap Benefits of Electronic Health Records," 6/25/2010, 2010; http://www.hhs.gov/news/press/2007pres/10/pr20071030a.html.
[22] H. P. Office, "ONC Issues Final Rule to Establish the Temporary Certification Program for Electronic Health Record Technology. Press Release. http://www.hhs.gov/news/press/2010pres/06/20100618d.html," 2010.
[23] W. Perry, *Effective methods for software testing*, Third ed., New York, NY, USA: John Wiley & Sons, 2006.
[24] G. Sindre, and A. Opdahl, “Eliciting security requirements with misuse cases,” *Requirements Engineering,* vol. 10, no. 1, pp. 34-44, 2005.
[25] E. Singer, "A Big Stimulus Boost for Electronic Health Records," *Technology Review*, MIT, 2009.
[26] B. Smith, L. Williams, and A. Austin, “Idea: Using System Level Testing for Revealing SQL Injection-Related Error Message Information Leaks,” *Lecture Notes in Computer Science,* vol. 5965, Engineering Secure Software and Systems (ESSoS 2010), pp. 192-200, 2010.
[27] F. Swiderski, and W. Snyder, *Threat Modeling*, Redmond, WA: Microsoft Press, 2004.
[28] R. Telang, and S. Wattal, “Impact of Software Vulnerability Announcements on the Market Value of Software Vendors - An Empirical Investigation,” in Workshop on the Economics of Information Security, Pittsburgh, PA, 2005, pp. 1-34.
[29] H. H. Thompson, and J. A. Whittaker, “Testing for software security,” *Dr. Dobb's Journal,* vol. 27, no. 11, pp. 24-34, 2002.
[30] I. Valdes, *Free and Open Source Software in Healthcare, Open Source Working Group White Paper*, American Medical Informatics Association, https://www.amia.org/files/Final-OS-WG%20White%20Paper_11_19_08.pdf, 2008.
[31] G. Wassermann, and Z. Su, “Static detection of cross-site scripting vulnerabilities,” in International Conference on Software Engineering, Leipzig, Germany, 2008, pp. 171-180.
[32] J. H. Weber, “Security evaluation and assurance of electronic health records,” *Studies in health technology and informatics,* vol. 143, pp. 290-296, 2009.
[33] M. W. Whalen, A. Rajan, M. P. Heimdahl *et al.*, “Coverage metrics for requirements-based testing,” in International Symposium on Software Testing and Analysis (ISSTA 06), Portland, Maine, USA, 2006, pp. 25-36.
[34] K. E. Wiegers, *Software Requirements, 2nd Edition*, Redmond, WA: Microsoft Press, 2003.