On the Development of A Black Box Security Test Pattern Catalog Based on Empirical Data

Ben Smith and Laurie Williams
Department of Computer Science
North Carolina Sate University
Raleigh, NC, USA
[ben_smith, laurie_williams]@ncsu.edu

Abstract— **The United States is suffering from a shortage of software security experts. We need a vehicle with which we can capture and disseminate knowledge about how to assess whether software systems have adequate defenses against malicious users. We have adapted the notion of a software design pattern to the domain of black box security testing. *The goal of this research is to codify a process for developing a software security test pattern catalog that provides a vehicle for capturing and disseminating knowledge about software security testing based upon grounded theory analysis of empirical data*. In this paper, we present six initial software security test patterns developed via our process. The empirical data we used for our grounded theory analysis was the CWE/SANS Top 25 security vulnerabilities. We created test cases based upon these patterns using 284 functional requirements from a public specification to generate 137 black box tests. We then executed these tests on each of five electronic health record systems, which are currently used to manage the clinical records for approximately 59 million patients, collectively. Out of the 685 total test executions, 253 (37%) revealed vulnerabilities in the five systems. Our evaluation shows that our patterns target different vulnerabilities, for example specific design flaws, which automated techniques like automated penetration testing and static analysis do not typically reveal. Our study suggests that by using the patterns presented in this paper, software engineers are better able to identify commonly overlooked security vulnerabilities.**

Keywords- security; testing; black box; patterns; health care

I. INTRODUCTION

The United States is suffering from a shortage of software security experts [12]. One expert claims that there are approximately 1,000 people in the country with the skills needed for cyber defense, and goes on to say that 20 to 30 times that many are needed [19]. Another report indicates that today's graduates in software engineering are unprepared to enter the workforce because they lack a solid understanding of how to make their applications secure [21]. Due to this shortage of security expertise, we need a vehicle with which we can capture and disseminate knowledge about how to assess whether software systems have adequate defenses against malicious users.

We adapt the notion of a software design pattern as proposed by Gamma et al. [13] to the domain of black box security testing. A design pattern is a description of a recurring problem and a well-defined description of the core solution to the problem that is described such that the pattern can be used many times but never in exactly the same way [2]. A software *security test* pattern is a template of a test case that exposes vulnerabilities, typically by emulating what an attacker would do to exploit those vulnerabilities.

Capturing attacker behavior in a security test case allows the systematic, repeated assessment of a system's defenses against a particular attack. We codify a process for developing security test patterns by identifying the similarities between test cases that expose known vulnerabilities and abstracting common components to make the test strategy reusable. This development of security test patterns using empirical data can help establish the foundations for a science of security [11], where knowledge about security can be gathered and organized in the form of testable explanations and predictions. Additionally, others can use this process of developing patterns to capture and disseminate security testing knowledge and to contribute additional patterns. Just as design patterns disseminate design knowledge, expressing proven security testing techniques as patterns makes them more accessible to people who are not experts in security, and makes it easier to reuse successful testing strategies [13].

The goal of this research is to codify a process for developing a software security test pattern catalog that provides a vehicle for capturing and disseminating knowledge about software security testing based upon grounded theory analysis of empirical data. We analyzed the CWE/SANS Top 25 Most Dangerous Programming Errors[1] using a grounded theory approach [14] to produce six initial test patterns. Future studies will allow us to evolve our pattern catalog and validate our process within the context of other data sources.

We applied our initial six test patterns to the Certification Commission for Health Information Technology (CCHIT) Ambulatory Criteria [1] to develop test cases from our patterns. Specifically, we employed 284 functional requirements from the CCHIT criteria to create a black box security test plan consisting of 137

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

security tests for four open source and one proprietary electronic health record (EHR) system: OpenEMR[2], ProprietaryMed[3], WorldVistA[4], Tolven[5], and PatientOS[6]. We then executed the 137 test cases on each of these five released EHR systems that are currently used to manage the records of over 59 million patients. This resulted in a total of 685 test executions. We further evaluated the test plan by comparing it to two techniques: automated penetration testing and automated static analysis, to identify the common vulnerabilities discovered by each technique. We have also developed a tool that uses natural language processing to automate the test case generation procedure using customizable patterns and keywords. The tool, pattern catalog, test plan and test results are available from our security test patterns wiki[7].

The rest of this paper is organized as follows. Section II reviews the background and related work. Section III introduces our security test pattern catalog, and illustrates how the patterns are used and how we developed them. Section IV illustrates how we applied the security test patterns to develop a black box test plan for five EHR systems. Section V illuminates a comparison we performed between test cases developed using our patterns and other security assessment techniques. Section VI describes the tool we have implemented to automatically parse natural language documents into test cases. Section VII lists the limitations of this work and this paper. Section VIII summarizes the paper.

II. Background and Related Work

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

A. Software Patterns

A *design pattern* is a description of a recurring problem and a well-defined description of the core solution to the problem that is described such that the pattern can be used many times but never in exactly the same way [2]. Design patterns were originally conceived by Alexander [3] in the field of building architecture, and tailored to software engineering by Gamma, et al. [13]. Alexander later introduced the notion of design pattern *languages* [2], which were tailored to software engineering by Coplien [9]. A pattern language is a collection of patterns that build on each other to generate a software system [2]. A pattern language is functionally complete, meaning that using one pattern creates an imbalance that is resolved by another pattern, and so on until a whole system can be developed that is balanced and well-designed [9].

We define a *software security test pattern* as a template of a test case that exposes vulnerabilities, typically by emulating what an attacker would do to exploit those vulnerabilities. Our software test pattern catalog cannot be thought of as a pattern language, in the way that Alexander and Coplien conceived of pattern languages. Instead, a *pattern catalog* is a collection of related patterns that apply to the same domain and contain the same elements (e.g. keywords, procedure template, an example of use, etc.) [13]. A pattern catalog is different than a pattern language in that a catalog is not necessarily functionally complete [13].

B. Developing Secure Software

Secure software methodologies, advocate considering security throughout the lifecycle. These methodologies indicate that development organizations should perform security-enriching processes such as the development of security requirements [25], penetration testing [4], threat modeling [16], automated static analysis [10], testing access control policies [7, 18], risk analysis and misuse cases [29], black box security testing [24], and many other techniques. 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 [23]. Each of these techniques plays a role in the prevention and removal of vulnerabilities, but none of these techniques will find every vulnerability [24].

A recent survey of information security and software development consultants indicates that although 81% of respondents were aware of formal secure development methodologies, only 30% indicated that they had adopted some methodology. Additionally, these methodologies rarely offer detailed advice on *how* to conduct black box security testing. For example, security experts use their extensive knowledge and experience to attempt attacks on an application in an exploratory and opportunistic way in a process known as penetration testing [4]. However, the success of current security assurance techniques like penetration testing that occur late in the product's lifecycle vary based on the skill, knowledge, and experience of testers [4].

The ISO defines three important concept with respect to information security, known as the "CIA Properties": confidentiality, integrity, and availability [17]. The concepts are described as follows:

- **Confidentiality:** The system shall not make the information more widely known than necessary.
- **Integrity:** The system shall not allow the information to be tainted. This does not guarantee the accuracy of the information, but guarantees that the same information that a user puts in will be the information that a user gets out.
- **Availability:** The system shall make its information available to the user at all times or as frequently as the user shall need it.

C. Security, Requirements, and Testing

Before developers can mitigate the risks of security threats, they must know the requirements for the system's

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

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 [33]. Functional requirements statements often specify desired system behavior in "shall" statements [33], 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." Several techniques have been constructed for developing and analyzing adequate security requirements [20], as well as improving the traceability of non-functional requirements to help maintain critical system qualities throughout a system's lifetime [8]. Several researchers have already proposed the use of requirements-based testing for creating a black box test plan [26], but we propose the first methodology we are aware of to use requirements-based testing to create a black box security test plan. 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 [31]. 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 [20]. 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 [30].

D. Grounded Theory

In grounded theory, first proposed by Glaser [14], theory is developed from data. Key points of the data are anchored with *codes*—labels that highlight frequently recurring properties of the data. Data from codes are then organized into *categories*. The core aspects of category that are coded and then established become a *concept.* Grounded theory also makes heavy use of the *constant comparative method*, where concepts that are recorded in *codes* are repetatively compared to other concepts and codes to see if the data should be restructured, while continuously reevaluating preconceived notions or existing theories [14]. To follow Glaser's dictum, "all is data". In the context of this paper, every bit of information about a vulnerability could be important for understanding the relationship between systems and attackers.

III. Development Process for a Security Test Pattern Catalog

This section provides our pattern catalog for developing black box software security tests as well as the procedure we used to develop the catalog based on empirical data.

A. Pattern Template

This section provides our template for the patterns in our pattern catalog.

Pattern Name (based on targeted vulnerability)

Keywords: The list of words that can be found within natural language documents that signal the need for the application of the pattern.

Targeted Vulnerability Types: The type of vulnerability that the test template (found below) is designed to expose or uncover. In this paper, we targeted vulnerabilities from the CWE/SANS Top 25. Future patterns contributed by us and others are not restricted to the CWE/SANS Top 25 or any vulnerability list.

CIA Properties: A list of the CIA properties (see Section II.B) that this pattern helps to uphold and how.

Test Procedure Template: A generalized form of the test case steps that should expose the targeted vulnerability type.

Expected Results Template: The generalized expected results for a system that is *not* vulnerable to this vulnerability type. Test failures in this context signify vulnerabilities that are present.

Example Natural Language Artifact: A natural language artifact that this pattern can be successfully applied to.

Example Test Procedure and Expected Results: The result of applying the template from the pattern to the example natural language artifact.

B. A Process for Developing a Security Test Pattern Catalog

The input to this process is a set of known or existing vulnerabilities. To develop a security test pattern catalog from empirical data, follow these steps:

1. Examine the first (or next) vulnerability.
2. Create a set of systematic, repeatable, black box test cases that target this vulnerability.
3. Compare this test set to the existing test plan and organize all tests into categories based on the similarities between the test procedures and expected results. Also consider reorganizing existing categories.
4. If there are more vulnerabilities to consider, return to Step 1.
5. Once the organization of the categories is established, extract the parts from the test cases in each category that are different, and keep the parts that are same.
6. These repeating parts in each category become the Procedure Template and Expected Results Template of a new pattern. Add this pattern to the catalog.

To develop the keywords for the template that are applicable for each pattern, we first consider the Test Procedure Template for when the test case would be applicable based on certain implied design characteristics found in natural language specifications that related to a security test case. For example, tests from the Input Validation Tests pattern (see Section III.C.1) are most relevant when input is involved. Then, we decompose natural language artifacts into their key phrases, as described in Section III.C. Next, we examine these decomposed key phrases and search for keywords that would imply these design characteristics. For example, the keywords *enter*, *store*, or *update*, will most often indicate the presence of some form of input field (though not necessarily) that a tester can exploit using the Input Validation Test Pattern.

C. *Security Test Pattern Catalog*

Sections III.C.1 through III.C.6 describe the six test case patterns that we have developed. We do *not* intend to indicate that this list of test case patterns is complete or sufficient for detecting all types of vulnerabilities. In the future, we plan to expand the test pattern catalog to include those contributed by the community and validated with our grounded theory approach. With the use of our automated natural language parsing tool (see Section VI), a security expert or tester can customize the test templates used as well as the keywords that signify the appropriate test pattern.

The parts of the patterns in braces (e.g., <*insert object phrase*>) indicate instructions to the user on how to apply the pattern. To apply a test pattern from our catalog, testers need a natural language artifact, such as a requirements statement. The structure of a statement in a natural language artifact contains certain key phrases that relate to the system's functionality. A statement can be broken into key action phrases, key object phrases, and supporting information. The object phrase 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 phrase 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 example, consider a requirements specification that states, "The system shall provide the ability to modify demographic information about the patient". This statement can be broken down as follows:

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

To apply a pattern to this requirement, a tester fills the component parts (action, object, and supporting information) into the italicized portion of the test template. We provide the results of applying the pattern in an example after each template.

1) *Pattern: Input Validation Vulnerability Tests*

Keywords: *Record*[8], *Enter, Update, Create, Capture, Store, Edit, Modify, Specify, Indicate, Maintain, Customize, Query, Receive, Search, Produce*

Targeted Vulnerability Types: 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, Uncontrolled Format String, NULL Pointer Dereference, Incorrect Conversion between Numeric Types, Untrusted Search Path, Use After Free, External Initialization of Trusted Variables or Data Stores, Missing Initialization

CIA Properties: *Integrity* – input validation attacks most commonly alter or destroy information that is contained within the system. *Confidentiality* – some input validation attacks, like SQL injection, reveal information in the database by tricking the system into executing a query that was unintended by its developers. *Availability* – some input validation attacks force the system into an endless loop, or bring the system to a malfunctioning state.

Test Procedure Template:

1. Authenticate as <*insert a registered user name*>.
2. Open the user interface for <*insert action phrase*>ing an <*insert object phrase*>.
3. Inject one random attack from the attack list[9] into a field of the <*insert object phrase*>.
4. Repeat the previous step for five attacks[10] from the attack list.
5. Repeat the previous two steps for five fields from the <*insert object phrase*>.

Expected Results Template:

- The system should gracefully inform the user that the input is invalid.
- The data store for the <*insert object phrase*> should remain intact.
- The system shall not reveal data that is not a part of this <*insert object phrase*>.
- No error messages should occur that reveal sensitive information about the system's configuration or architecture.

Example Natural Language Artifact: Requirement AM 02.04 - The system shall provide the ability to modify demographic information about the patient.

Example Test Procedure:

1. Authenticate as Dr. Robert Alexander.
2. Open the user interface for entering patient demographic information and create a new patient.

[8] Keywords that appear in more than one pattern are italicized.

[9] Any attack list can be used, but for this paper we used a list of common attacks from http://neurofuzz.com.

[10] The choice of the number of tries for attacks is admittedly arbitrary. A security tester could execute as many attacks in as many fields as he or she desires. Some limit on the number of attacks will help in situations where testing a product is time-limited.

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.

Example 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 intact.
- No data should be revealed that is not a part of this patient's demographic information.
- No error messages should occur that reveal sensitive information about the system's configuration or architecture.

2) Pattern: Force Exposure Tests

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

Targeted Vulnerability Types: Improper Access Control, Improper Authorization, Reliance on Untrusted Inputs in a Security Decision, Use of Hard-coded Credentials, Missing Authentication for Critical Function, Incorrect Permission Assignment for Critical Resource, Improper Cross-boundary Removal of Sensitive Data, Link Following, Exposed Dangerous Method or Function, Improper Control of Interaction Frequency

CIA Properties: *Confidentiality* – force exposure tests assert that the system does not reveal information to users that it cannot identify or users that do not have the proper authorization to view that information.

Test Procedure Template:

1. Authenticate as <*insert registered user name*>.
2. Open the user interface for <*insert action phrase*>ing a <*insert object phrase*>.
3. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
4. Logout as <*insert a registered user name*>.
5. Repeat the actions recorded in the earlier step, either by entering the stored URL, or by repeating the interface actions.

Expected Results Template:

- The interface should not be visible or accessible to an unauthorized user.
- Upon forcing the page, the user is denied access without authorization.

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

Example Test Procedure:

1. Authenticate as Dr. Robert Alexander.
2. Open the user interface for searching the notes of patient Ellen Thompson for a provider.
3. Observe the method of accessing this interface, either by recording the URL or the series of user interface actions to reach this interface.
4. Logout.
5. Repeat the actions recorded in the earlier step, either by entering the stored URL, or by repeating the interface actions.

Example Expected Results:

- The interface should not be visible or accessible to an unauthorized user.
- Upon forcing the page, the user is denied access without authorization.

3) Pattern: Malicious File Tests

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

Targeted Vulnerability Types: Unrestricted Upload of File with Dangerous Type, Download of Code Without Integrity Check

CIA Properties: *Availability* – malicious files often render the system or the user's machine dysfunctional.

Test Procedure Template:

1. Authenticate as <*insert registered user name*>.
2. Open the user interface for <*insert action phrase*> a <*insert object phrase*>.
3. Select and upload a malicious file.
4. View or download the malicious file.

Expected Results Template:

- The file should be rejected upon selection or should not be allowed to be stored.

Example Natural Language Artifact: Requirement AM 09.01 - The system shall provide the ability to capture and store external documents.

Example Test Procedure:

1. Authenticate as Dr. Robert Alexander.
2. Open the user interface for storing an external document for patient Ellen Thompson.
3. Select and upload a malicious file.
4. View or download the malicious file.

Example Expected Results:

- The file should be rejected upon selection or should not be allowed to be stored.

4) Pattern: Malicious Use of Security Functions Tests

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

Targeted Vulnerability Types: Missing Release of Resource After Effective Lifetime, Improper Restriction of Excessive Authentication Attempts, Operation on a Resource after Expiration or Release, Guessable CAPTCHA, Missing Encryption of Sensitive Data, Improper Check for Unusual or Exceptional Conditions, Allocation of Resources without Limits or Throttling, Use of a Broken or Risky Cryptographic Algorithm, Use of Insufficiently Random Data, could apply to many others, depending on what the security feature is meant to prevent.

CIA Properties: *Confidentiality* – most attacks on security functions will allow a user unauthorized access to the system and the records it contains.

Test Procedure 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.

Expected Results Template: Same as above.

Example Natural Language Artifact: Requirement SC 03.02 - When passwords are used, the system shall support password strength rules that allow for minimum number of characters, and inclusion of alpha-numeric complexity.

Example Test Procedure:

1. Open the change password screen.
2. Enter the appropriate identifying information for Dr. Green.
3. Attempt to change Dr. Green's password to an empty string.
4. Attempt to change Dr. Green's password to the letter 'a'.

Example Expected Results:

- The system should disallow all password changes attempted in this test.

5) Pattern: Dangerous URL Tests

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

Targeted Vulnerability Types: Open Redirect, Cross-Site Request Forgery

CIA Properties: *Availability* – dangerous URL attacks are often used to prevent a user from being able to access his or her data. *Confidentiality* – some dangerous URL attacks are meant to intercept or steal a user's identity or their personal information.

Test Procedure Template:

1. Authenticate as <*insert a registered user name*>.
2. Open the user interface for <*insert action phrase*> an <*insert object phrase*>.
3. Create a new record for <*insert object phrase*>.
4. Insert an attack string from the malicious URLs list for the <*insert object phrase*>.

Expected Results Template:

- The link should be rejected as malicious.
- An error message should indicate to the provider that the link points to a dangerous website.
- The data store for the links should remain intact.
- No data should be revealed that is not a part of this <*insert object phrase*>.
- No error messages should occur that reveal sensitive information about the system's configuration or architecture.

Example Natural Language Artifact: Requirement AM 08.13 - 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.

Example Test Procedure:

1. Authenticate as Dr. Robert Alexander.
2. Open the user interface for adding patient-specific instructions for patient Ellen Thompson.
3. Create a new record for patient-specific instructions.
4. Insert an attack string from the malicious URLs list for the patient-specific instructions.

Example Expected Results:

- An error message should indicate to the provider that the link points to a dangerous website.
- The link should be rejected as malicious.
- The data store for the links should remain intact.
- No data should be revealed that is not a part of these patient-specific instructions.
- No error messages should occur that reveal sensitive information about the system's configuration or architecture.

6) Pattern: Audit Tests

Keywords: patient record, demographics, credit card information, grade point average (GPA), personal identification information

Targeted Vulnerability Types: None. *Insufficient Logging* is the CWE classification for vulnerabilities that these test cases can expose.

CIA Properties: *Confidentiality* – without the deterrent effect of a record of viewing personal information, insider attackers will view anyone's information without consequence.

Test Procedure Template:

1. Authenticate as <*insert a registered user name*>.
2. Open the user interface for <*insert action phrase*>ing an <*insert object phrase*>.
3. Logout as <*insert a registered user name*>.
4. Authenticate as <*insert an administrator's user name*>.
5. Open the audit records for today's date.

Expected Results Template:

- The audit records should show that *registered user* <*insert action phrase*>ed an <*insert object phrase*>.
- The audit records should be clearly readable and easily accessible.

Example Natural Language Artifact: Requirement AM 03.08.01 – The system shall provide the ability to associate orders and medications with one or more codified problems/diagnoses.

Example Test Procedure:

1. Authenticate as Dr. Robert Alexander.

2. Open the user interface for adding an association between Theodore S. Smith's Hypertension diagnosis and Zantac.
3. Logout as Dr. Robert Alexander.
4. Authenticate as Denny Hudzinger.
5. Open the audit records for today's date. If necessary, focus on patient Theodore S. Smith.

Example Expected Results:

- The audit records should show adding and removing the association of Theodore S. Smith's Hypertension diagnosis and Zantac, both linked to Dr. Robert Alexander, and with today's date.
- The audit records should be clearly readable and easily accessible.

IV. Applying the Security Test Patterns

We applied our six test patterns (see Section III.C) to create a black box security test plan, which we executed on for four open source and one proprietary electronic health record (EHR) systems.

A. Choosing the Appropriate Test Pattern

For this paper, we chose patterns from our catalog using a functional requirements specification. However, our pattern catalog does not rely on functional requirements statements to function: as long as the key phrases can be identified in a natural language text, our pattern catalog is applicable.

The structure of a requirements statement, as well as certain keywords, can guide the tester to choose an appropriate test pattern. We used *key phrases* and *supporting information* in a requirements statement to determine the relevant security test pattern that will most likely reveal vulnerabilities in the system. 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.

Requirements specifications typically conform to the following format: "*The system shall provide the ability to <action> a <object> <and/with/in supporting information>*." For example, 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 [32], SQL injection [15] 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. The pattern Input Validation Tests, which our pattern catalog includes, contains the keyword *modify* and will attempt to tamper with or reveal information from the demographic information object.

B. Developing the First Six Security Test Patterns

Following a grounded theory process as described in Section III.B, we developed patterns based on these CWE/SANS Top 25+ and the keywords contained within a requirements specification. We describe both of these artifacts briefly in this section. The CWE/SANS Top 25, lists the most dangerous security programming errors based on prevalence and potential consequences. We did not tailor our test case patterns based on the vulnerabilities we have seen reported in the systems we evaluated. We captured a set of general test cases 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.

The Certification Commission of Healthcare IT (CCHIT)[11] defined certification criteria focused on the functional capabilities that should be included in ambulatory (outpatient) and inpatient EHR systems [28] in 2006, through a consensus-based process that engaged stakeholders. In this paper, we chose to apply our test cases to the CCHIT certification criteria since these criteria express behavior that an EHR must exhibit in order to be certified [6].

C. Targeted Systems

We chose five EHR systems for our testing that are responsible for managing the records for over 59 million patients. Deploying and configuring the open source systems in this paper (all except ProprietaryMed) was a time-intensive task that required much expertise and effort to complete. EHR systems provide a good test bed for applying our pattern catalog because all five systems implement the same functional requirements, meaning we could evaluate our resultant test plan multiple times. Table 1 presents a summary of the facts for each system.

D. Test Case Results

We used our test patterns on the 284 CCHIT functional requirements statements and created 137 tests, which can be found on our test patterns wiki. Table 2 lists the overall test case results for the system under test described in Section IV.C. 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

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

Table 1. Summary of the Systems under Test

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.[12]	305,944 / 1,643[13]	GPL	34 [14]	31 million[15]
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

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 test patterns wiki, asserts that the tester should attempt unencrypted HTTP (i.e. not HTTPS/SSL connections) access to the EHR system if the 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[16], 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 does not implement 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.

Overall, our test plan launched 253 (see the cell with the * in Table 2) 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. We developed the security test plan in approximately 60 person hours. Executing the test plan manually on each of the system under test consumed approximately six to eight person hours per project. An undergraduate student with minimal security experience also executed the test plan on the systems in this paper and achieved similar results, indicating that non-expert software testers can use the test plan. We also alerted developers to the vulnerabilities we found by posting respective healthcare IT communities' bug report pages.

V. Comparison to Other Techniques

In light of the numerous techniques that one can apply to develop secure software (see Section II.B), we asked how our technique would compare to using existing security assessment techniques. To answer this question, we performed a comparative evaluation between our black box test plan and two automated security assessment techniques [5]: automated static analysis using Fortify 360 v2.6.5, and automated penetration testing using IBM Rational AppScan v8.0.0.0. Our goal in performing this evaluation was to see the commonalities, if any, in the vulnerabilities discovered by the respective techniques. For comparison, we used two of our systems under test, Tolven and OpenEMR, described in Section IV.C. Section V.A discusses our methodology for using the two security analysis tools. Section V.B summarizes these results.

A. *Security Assessment Techniques*

This section describes the details of how we gathered the data from two security assessment techniques.

[12] https://sourceforge.net/projects/openemr/files/stats/timeline
[13] Calculated using CLOC v1.08, http://cloc.sourceforge.net
[14] http://sourceforge.net/project/memberlist.php?group_id=60081
[15] http://www.openmedsoftware.org/wiki/
[16] 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 systems in paper and demonstrate that our test plan could function well on a thin client-based system.

Table 2. Test Results for the Five Systems under Test

	Pattern	Input Validation Vuln.	Malicious File	Dangerous URL	Force Exposure	Security Features	Audit	Total
	Prefix	IV	MF	DU	FE	SF	AU	
Overall (in five EHRs)	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**

Automated Static Analysis. Fortify 360[17] supports analysis of 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".

Automated Penetration Testing. Rational AppScan[18] conducts a black box security evaluation of the website by crawling the web application and attempting a variety of attacks. We left the default scanning options selected for our automated penetration testing.

Classifying False Positives. 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 vulnerability. For static analysis, we examined the line of code classified as vulnerable and also examined related methods. For automated penetration testing, we performed false positive classification 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 we had to attempt to manually recreate the attack through the application to confirm whether the potential vulnerability was a true positive.

B. Results

Fortify reported 5,036 issues in OpenEMR, of which we determined 3,715 to be false positives. In Tolven, Fortify reported 2,315 issues, of which we determined 2,265 to be false positives. However, we did not exclude false positives from our analysis of Fortify, because many times our test plan found an exploit that corresponded to an alert that we had classified as false positive.

AppScan reported 735 issues in OpenEMR, of which we determined 25 to be false positives. In Tolven, AppScan reported 37 issues, of which the we determined 15 to be false positives. In this analysis, our test plan never found a vulnerability that we had marked as a false positive, and so false positives were excluded from our analysis.

As Table 3 shows, Fortify and AppScan both uncovered many issues that our test plan failed to identify. However as Table 3 shows, these automated security assessment techniques missed 80-90% of our discovered vulnerabilities as well. Using static analysis and automated penetration testing tools to guide testers towards efficient, sweeping changes to a system's input validation mechanisms is preferable unless cost-prohibitive. Development organizations that are unable to afford expensive proprietary automated security assessment tools can still use our approach. The majority (84% in Tolven and 68% in OpenEMR) of the issues that our test plan discovered and the tools did not were in the audit category. With the prevalence of insider attacks [27] and the attacks a malicious user could perpetrate on patients' records with the lack of a sufficient audit mechanism, we find that a security assessment of EHR systems should address these issues.

VI. AUTOMATION

We have implemented the Security Test Pattern Instantiator (STPI), a requirements parsing tool using the Stanford Parser libraries [22]. A running copy of the STPI web application is available from our security test patterns website. STPI uses the natural language processing engine within the Stanford Parser to extract the key phrases described in Section III.C. Using the mapping of keywords to test types described in Section IV.A, the tool automatically generates an HTML file containing the systematic black box security test plan. We used the CCHIT functional requirements statements as well as the manual test plan described in Section IV to evaluate the level of agreement between the STPI and the manual parsing of the 284 requirements described in Section IV.B.

The tool automatically parsed action phrases from the natural language requirements statement that agreed with the manual analysis for 73% of the requirements. For 15% of the requirements, the action phrase the tool parsed

Table 3. Discovered and Missed Issues for OpenEMR and Tolven

	OpenEMR	Tolven
Test Plan Total	**63**	**35**
AppScan Discovered	6 (9.5%)	2 (5.7%)
AppScan Missed	57 (90.5%)	33 (94.3%)
Fortify Discovered	12 (19.9%)	4 (11.4%)
Fortify Missed	51 (80.1%)	31 (88.5%)
AppScan Total	**710**	**22**
Tests Discovered	6 (0.8%)	2 (9%)
Tests Missed	704 (99%)	20 (91%)
Fortify Total	**1321**	**50**
Tests Discovered	12 (0.9%)	4 (8%)
Tests Missed	1309 (99%)	46 (92%)

[17] https://www.fortify.com/
[18] http://www-01.ibm.com/software/awdtools/appscan/

agreed with the manual analysis, and the object phrase the STPI parsed did not. For 11% of the requirements, the tool parsed neither action phrase nor object phrase in agreement with the manual analysis. The tool agreed with the test case patterns selected by the manual analysis for 73.3% of the requirements. STPI allows manual intervention at each phase of the test generation process.

We also asked three graduate students to compare the key phrases selected by the tool and those selected by the manual evaluation and determine whether the selections were equivalent. Additionally, when the students thought the phrases were not equivalent, we asked them which selection was incorrect. On average, the students indicated that the key action phrases selected by the tool and manual analysis were the same for 73% of the requirements, and the key object phrases were the same for 72% of the requirements. When the students indicated that the phrases were not equivalent, they stated that the tool was wrong for 24% of the requirements.

VII. Threats to Validity

Internal validity: 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 pattern catalog could discover, but different architectures and platforms offer different security challenges. 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

systems 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. The assessment tools we chose for this study do not specifically target the domain and type of security vulnerabilities that we aim to discover with our approach, and this may have biased the results.

External validity: Grounded theory is valid for the data involved in its development, but we will conduct further work before establishing any generalizations [14]. Our results may only apply to open source or non-industrial systems in health care. 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 pattern catalog, 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 pattern catalog is no exception to this rule. Similarly, other security assessment tools besides IBM's Rational AppScan and Fortify 360 may produce different results that may share more commonalities with our approach.

Construct validity: Our use of automated static analysis and penetration testing is subject to human error. The inspection of alerts may have missed more vulnerabilities that could have been shared between our test plan and the automated security assessment tools. Additionally, the application and evaluation of the test plan itself is subject to human error. A tester may have executed a test incorrectly, or incorrectly characterized a test result.

VIII. Summary

In this paper, we codified a process for developing security a security test pattern catalog that uses a grounded theory approach to identify the similarities between test cases that expose known vulnerabilities and abstracting common components to make the test strategy reusable. We used this process to develop six black box security test patterns that target the CWE/SANS Top 25+. We then used our pattern catalog to create 137 test cases based on the 284 CCHIT ambulatory certification criteria and running the test plan on each of 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. Additionally, our comparison to other techniques shows that two automated security assessment tools missed 80-90% of our discovered vulnerabilities. Finally, our automation of this technique shows promising results for using a tool to help non-experts in security to create and use black box security testing in a systematic fashion.

Acknowledgment

We would like to thank the Realsearch research group for their helpful comments and suggestions on this 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.

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] C. Alexander, *A Pattern Language: Town, Buildings, Construction*, Oxford, UK: Oxford University Press, 1977.

[3] C. Alexander, *The Timeless Way of Building*, Oxford, UK: Oxford University Press, 1979.

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

[5] A. Austin, and L. Williams, "One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques," in

Emperical Software Engineering and Measurement (ESEM), Banff, Alberta, Canada, 2011, to appear.

[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] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff, "An approach to modular and testable security models of real-world health-care applications," in ACM Symposium on Access Control Models and Technologies (SACMAT), Innsbruck, Austria, 2011.

[8] 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.

[9] J. Coplien, *Software Patterns*, New York, NY, USA: SIGS Books & Multimedia, 2000.

[10] D. Evans, and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software,* vol. 19, no. 1, pp. 42-51, 2002.

[11] D. Evans, and S. Stolfo, "Guest Editor's Introduction: The Science of Security," *IEEE Security & Privacy,* vol. 9, no. 3, pp. 16-17, 2011.

[12] K. Evans, and F. Reeder, *A Human Capital Crisis in Cybersecurity: Technical Proficiency Matters*, Center for Strategic and International Studies, http://csis.org/files/publication/101111_Evans_HumanCapital_Web.pdf, 2010.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, MA: Addison Wesley Longman Publishing Company, 1995.

[14] B. G. Glaser, and A. L. Strauss, *The Discovery of Grounded Theory; Strategies for Grounded Research*, New York, NY: Aldine de Gruyter, 1967.

[15] 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.

[16] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack, "Uncover Security Design Flaws Using the STRIDE Approach," *MSDN Magazine*, http://msdn.microsoft.com/en-us/magazine/cc163519.aspx, 2006].

[17] ISO, *ISO/IEC IS 17799: Information technology - Code of practice for information security management.*, 2005.

[18] J. Julliand, P.-A. Masson, and R. Tissot, "Generating security tests in addition to functional tests," in Automation of Software Testing (AST), Leipzig, Germany, 2008, pp. 41-44.

[19] M. L. Kelly, *Cyberwarrior Shortage Threatens US Security*, NPR Morning Edition, http://www.npr.mobi/templates/transcript/transcript.php?storyId=128574055, 2010.

[20] 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.

[21] R. Lemos, "Security lessons still lacking for computer science grads," *InfoWorld, http://www.infoworld.com/t/application-security/security-lessons-still-lacking-computer-science-grads-769*, 2011.

[22] M. P. Marcus, M. A. Marcinkeiwicz, and B. Santorini, "Building a large annotated corpus of Enligh: The Penn Treebank," *Journal of Computational Linguistics,* vol. 19, no. 2, pp. 313-330, 1993.

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

[24] G. McGraw, and B. Potter, "Software Security Testing," *IEEE Security and Privacy,* vol. 2, no. 5, pp. 81-85, 2004.

[25] 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.

[26] G. Mogyorodi, "Requirements-based testing: an overview," in Technology of Object-oriented languages and systems, Santa Barbara, CA, 2001, pp. 286-295.

[27] A. P. Moore, D. M. Cappelli, and R. F. Trzeciak, *The "Big Picture" of Insider IT Sabotage Across U.S. Critical Infrastructures*, Carnegie Mellon Software Engineering Institute. CERT Program. , 2008.

[28] 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.

[29] G. Sindre, and A. Opdahl, "Eliciting security requirements with misuse cases," *Requirements Engineering,* vol. 10, no. 1, pp. 34-44, 2005.

[30] 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.

[31] H. H. Thompson, and J. A. Whittaker, "Testing for software security," *Dr. Dobb's Journal,* vol. 27, no. 11, pp. 24-34, 2002.

[32] G. Wassermann, and Z. Su, "Static detection of cross-site scripting vulnerabilities," in International Conference on Software Engineering, Leipzig, Germany, 2008, pp. 171-180.

[33] K. E. Wiegers, *Software Requirements, 2nd Edition*, Redmond, WA: Microsoft Press, 2003.