Information and Software Technology 55 (2013) 1279–1288

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A comparison of the efficiency and effectiveness of vulnerability discovery techniques

Andrew Austin, Casper Holmgreen, Laurie Williams *

Department of Computer Science, North Carolina State University, Raleigh 27695, USA

ARTICLE INFO

Article history:
Available online 8 December 2012

Keywords:
Security
Vulnerability
Static analysis
Penetration testing
Black box testing
White box testing

ABSTRACT

Context: Security vulnerabilities discovered later in the development cycle are more expensive to fix than those discovered early. Therefore, software developers should strive to discover vulnerabilities as early as possible. Unfortunately, the large size of code bases and lack of developer expertise can make discovering software vulnerabilities difficult. A number of vulnerability discovery techniques are available, each with their own strengths.
Objective: The objective of this research is to aid in the selection of vulnerability discovery techniques by comparing the vulnerabilities detected by each and comparing their efficiencies.
Method: We conducted three case studies using three electronic health record systems to compare four vulnerability discovery techniques: exploratory manual penetration testing, systematic manual penetration testing, automated penetration testing, and automated static analysis.
Results: In our case study, we found empirical evidence that no single technique discovered every type of vulnerability. We discovered that the specific set of vulnerabilities identified by one tool was largely orthogonal to that of other tools. Systematic manual penetration testing found the most design flaws, while automated static analysis found the most implementation bugs. The most efficient discovery technique in terms of vulnerabilities discovered per hour was automated penetration testing.
Conclusion: The results show that employing a single technique for vulnerability discovery is insufficient for finding all types of vulnerabilities. Each technique identified only a subset of the vulnerabilities, which, for the most part were independent of each other. Our results suggest that in order to discover the greatest variety of vulnerability types, at least systematic manual penetration testing and automated static analysis should be performed.

1. Introduction

Results of decades of empirical research on effectiveness and efficiency of fault and failure discovery techniques, such as unit testing and inspections, can be used to provide evidence-based guidance on the use of these techniques. However, similar empirical results on the effectiveness and efficiency of vulnerability discovery techniques, such as security-focused automated static analysis and penetration testing are sparse. As a result, practitioners lack evidence-based guidance on the use of vulnerability discovery techniques.

In his book *Software Security: Building Security In*, Gary McGraw draws on his experience as a security researcher and claims: “Security problems evolve, grow, and mutate, just like species on a continent. No one technique or set of rules will ever perfectly detect all security vulnerabilities” [1]. Instead, he advocates using a variety of vulnerability discovery and prevention techniques throughout the software development lifecycle. McGraw’s claim, however, is based upon his experience and is not substantiated with empirical evidence. *The objective of this research is to aid in the selection of vulnerability discovery techniques by comparing the vulnerabilities detected using each and comparing their efficiencies.*

In previous work [2], the first author analyzed four vulnerability discovery techniques on two electronic health record (EHR) systems. The vulnerability discovery techniques analyzed included: exploratory manual penetration testing, systematic manual penetration testing, automated penetration testing, and automated static analysis. The first author used these four techniques on Tolven Electronic Clinician Health Record (eCHR)[1] and OpenEMR.[2] These two systems are currently used within the United States to store patient records. Tolven eCHR and OpenEMR are web-based systems. This paper adds the same analysis conducted on an additional

* Corresponding author.
E-mail addresses: andrew_austin@ncsu.edu (A. Austin), cmholmgr@ncsu.edu (C. Holmgreen), williams@csc.ncsu.edu (L. Williams).

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

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.11.007

EHR, PatientOS,[3] performed by the second author. PatientOS is a custom client/server application written in Java. The new results corroborate the findings of the previous paper. Additionally, we examine the validity of the study in greater detail and offer new insights based on the PatientOS data. The second author was careful to follow the exact same procedure used by the first author and collaborated throughout the process with the first author to confirm agreement on classifications.

We classified the vulnerabilities found by these techniques as either implementation bugs or design flaws. *Design flaws* are high-level problems associated with the architecture of the software. An example of a design flaw is failure of authentication where necessary. *Implementation bugs* are code-level software problems, such as an instance of buffer overflow. Design flaws and implementation bugs occur with roughly equal frequency [1]. We then manually analyzed each discovered vulnerability to determine if the same vulnerability could be found by multiple vulnerability discovery techniques.

The contributions of this paper are as follows:

- A comparison of the type and number of vulnerabilities found with exploratory manual penetration testing, systematic manual penetration testing, automated penetration testing, and automated static analysis.
- Empirical evidence indicating which discovery techniques should be used to find implementation bugs and design flaw types of vulnerabilities.
- An evaluation of the efficiency for each vulnerability discovery technique based on the metric vulnerabilities discovered per hour.
- Additional data collected with a new EHR that served to further support previous work [2].

The rest of the paper is organized as follows; Section 2 provides background information requiring familiarity to understand the contents of the paper. Section 3 describes related work. Section 4 describes the case study and its methodology. Section 5 gives our study results. Section 6 discusses our results and provides analysis. Section 7 discusses our limitations. Section 8 summarizes our conclusions. Finally Section 9 talks about possible future work.

2. Background

This section describes the terminology used throughout the paper and gives background information on the types of security issues one may encounter when doing security analysis. The section also discusses work related to the vulnerability discovery techniques.

2.1. Vulnerability discovery techniques

There are many different techniques available to practitioners for the discovery of software vulnerabilities. NIST defines a vulnerability to be "a flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy." [3]. Vulnerabilities are a class of software faults specific to security in that they can cause unintended program behavior, specifically in the context of the security policy of the software. In this paper, we discuss a variety of the most common vulnerability discovery techniques. These techniques are chosen from three fundamentally different approaches to vulnerability discovery: automated penetration testing, manual penetration testing, and static code analysis.

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

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

Techniques for discovering software vulnerabilities are not perfect and they sometimes incorrectly label code as containing a vulnerability when no vulnerability exists. This mislabeling is called a **false positive**. Therefore, developers must manually examine each potential fault reported by these tools to determine their validity. We call potential faults that have security implications **potential vulnerabilities.**

2.2. Vulnerability types

Vulnerability types can be classified as either implementation bugs or design flaws. In this section, we provide background on some common vulnerability types based upon their Common Weakness Enumeration (CWE) descriptions.[4] The CWE is a community maintained body of information regarding common software vulnerabilities. It attempts to formalize definitions of software security faults and provides a wealth of information on securing software.

We provide background on eight implementation bug vulnerability types found in the three projects we analyzed. **Cross-Site Scripting (XSS)** (CWE-79) vulnerabilities occur when input is taken from a user and not correctly validated, allowing for malicious code to be injected into the application and subsequently displayed to an end user. **SQL Injection** (CWE-89) vulnerabilities occur when user input is not correctly validated and the input is directly used in a database query. Not validating the input allows a malicious user to directly manipulate the data returned by the database to obtain potentially sensitive information. A **dangerous function** (CWE-242) vulnerability occurs when a method is used within code that is inherently insecure or deprecated. Such methods or functions should not be used because attackers can use common knowledge of their weakness to exploit the application. A **path manipulation** (CWE-22) vulnerability occurs when users are allowed to view files or folders outside of those intended by the

[3] http://www.patientos.org/.

[4] The Common Weakness Enumeration is a community developed dictionary of software weakness types [HYPERLINK \l "The11" 10].

application. An **error information leak** (CWE-209) vulnerability occurs when information or an error is displayed directly to a user. These errors can contain sensitive information or even authentication credentials to allow attackers greater access to the application. A failure to set the **HTTPOnly attribute** allows for non-http access to browser cookies. Such a vulnerability allows client side code to access the cookies, allowing session information or other sensitive data to be stolen in cross site scripting or phishing attacks [9]. A **hidden field manipulation** (CWE-472) vulnerability occurs when data in hidden fields are not properly validated and the field is implicitly trusted. Trusting this form of user input can lead to issues such as SQL injection and cross site scripting, or can allow inaccurate information to be inserted into the database. A **command injection** (CWE-78) vulnerability occurs when input from the user is directly executed. This vulnerability allows malicious users to directly execute commands on the host as a trusted user.

Our projects also contained vulnerability types that are considered design flaws [10]. A **nonexistent access control** (CWE-285) vulnerability occurs when access to a particular feature is not protected, granting anyone, including malicious users access to resources or functionality. A **lack of auditing** (CWE-778) vulnerability occurs when a critical event is not logged or recorded. A **trust boundary violation (**CWE-501) occurs when trusted and untrusted data is mixed in a data structure. **Dangerous file upload** (CWE-434) can occur when the system is not properly designed to handle potentially malicious files. An **uncontrolled resource consumption** (CWE-400) vulnerability exists when a system does not impose restrictions or limits on the number of resources a user is able to request. This can potentially lead to denial-of-service, as resources can be arbitrarily tied up with little effort.

3. Related work

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

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

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

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

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

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

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

4. Case study

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

4.1. Subject selection

Due to recent legislative requirements[5] in the United States, development and adoption of EHR systems has suddenly taken off. To have our case studies generalize to large, real world systems, two open-source web-applications as well as one open-source client/server application were chosen to be our subjects. The two browser-based systems we studied were Tolven eCHR and OpenEMR. The client/server application was PatientOS.

Tolven eCHR is an open-source browser-based EHR system. The project has 12 contributing developers, and commercial support is available from Tolven, Inc. **OpenEMR** is also an open-source web-based EHR system. The project has a community of 17 contributing developers and at least 23 organizations providing commercial support within the United States [20]. **PatientOS** is an open-source client/server EHR. Its development is guided primarily by a single developer, but commercial support is also available. Some additional characteristics of each of the systems are included in Table 1.

4.2. Case study methodology

We first collected the vulnerabilities found by each of the four vulnerability discovery techniques. We then classified each vulnerability as either a true or false positive. The time spent by each author during the classification phase was recorded using Gnome Time Tracker, a powerful utility for project time tracking. Finally, we analyzed whether a vulnerability was found by more than one technique. The next five subsections examine each of these steps in greater detail.

4.2.1. Exploratory manual penetration testing

To keep other discovery techniques from biasing our exploratory manual penetration testing, we conducted exploratory manual penetration testing prior to conducting vulnerability discovery with other techniques. To perform exploratory manual penetration testing, we manually attempted to exploit various components of the test subjects opportunistically, in an ad hoc manner. The exploratory manual penetration testing was conducted by authenticating with the target application and manually navigating through each page trying various attacks. We used supplemental

[5] http://www.recovery.gov/.

Table 1
Characteristics of Tolven eCHR and OpenEMR (adapted from [2]).

	Tolven eCHR	OpenEMR	PatientOS
Language	Java	PHP	Java
Version evaluated	RC1 (5/28/2010)	3.1.0 (8/29/2009)	0.99 (1/17/2010)
Lines of code (counted by CLOC1.08)	466,538	277,702	487,437

tools such as web browsers, JavaScript debuggers (e.g. Firebug[6]) and http proxies (e.g. WebScarab[7]) for viewing raw http requests. BackTrack 5[8] and its large collection of tools specifically made for penetration testing was used when attempting to exploit PatientOS. In particular, WireShark[9] proved to be very useful for analyzing traffic sent between the client and the server sides of the application.

4.2.2. Static analysis

To perform static analysis, we used Fortify 360 v.2.6.[10] Fortify 360 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". Fortify 360 generated a list of potential vulnerabilities when scanning was complete.

4.2.3. Automated penetration testing

To conduct automated penetration testing, we used IBM Rational AppScan 8.0.[11] Rational AppScan conducts a black box security evaluation of a website by crawling a web application and attempting to perform a variety of attacks. To use AppScan, we provided authentication credentials to the systems so that the tool could login to both of our browser-based test subjects. We left the default scanning options selected for our automated penetration tests. AppScan generated a list of potential vulnerabilities when scanning was complete.

The ease with which the DOM-elements of a browser-based interface can be manipulated is unmatched by traditional client-side software. Differences in GUI libraries and operating systems complicate matters when dealing with custom client software. Basic tools exist for very specific analyses, but none of them offer the same end-to-end analysis offered by the web-application tools, particularly AppScan. Thus, PatientOS does not include results for automated penetration testing.

4.2.4. Systematic manual penetration testing

One vulnerability discovery technique, proposed by Smith and Williams [21], suggests using the functional requirements specifications of the software system to systematically generate security tests to surface security vulnerabilities. The authors create these tests by breaking the systems functional requirements statements into distinct phrase types such "Action Phrase" and "Object Phrase." Using these two phrase types, the authors then propose a systematic method of generating security tests using common patterns. Since the authors have provided a detailed test plan [21] and have run their test plans against our subjects, we will use the results they obtained for our study.

4.2.5. False positive classification

Automated security tools are far from perfect. Both static analysis and automated penetration testing tools generate long lists of potential vulnerabilities that must be manually classified as either true or false positives. To perform this classification, we manually examined each individual vulnerability. For automated penetration testing, false positive classification was performed by looking at the raw HTTP requests generated and confirming if the attempted exploit was actually visible in the raw output or accepted as trusted input. For static analysis, we examined the line of code classified as vulnerable and also examined related methods. 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.

4.2.6. Comparison of vulnerabilities found

Comparing the results of each technique on each subject allowed us to make conclusions regarding their effectiveness in particular situations. By comparing one technique against another, specific comments on strengths and weaknesses could be made, as well as conclusions regarding the best approach to security testing. Because static analysis discovered the most vulnerabilities, it was used as the benchmark to be compared with all other techniques. We looked at the number of vulnerabilities found by one technique against another, but we also looked at the specific vulnerabilities found by each technique to see if it was discovered by more than one technique.

5. Results

In this section we present the results of our case study. We include analyses of the results of applying the four vulnerability discovery techniques against the three EHR systems. We include false positive rates with the automated tools. Finally, a metric intended to reflect overall efficiency, vulnerabilities discovered per hour, is discussed for each. This metric only takes the time spent by the authors reviewing potential vulnerabilities into account; it does not include time spent setting up or running the tools.

5.1. Exploratory manual penetration testing

For Tolven eCHR, the first author spent approximately 15 man-hours performing exploratory manual penetration testing. After 15 h of testing, we were unable to find any security issues in Tolven eCHR. Since we discovered no vulnerabilities in Tolven eCHR with exploratory manual testing, the vulnerabilities per hour metric is 0.

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

Since each vulnerability discovered was exploited, they are all considered true positives. We classified all of the bugs found with exploratory manual penetration testing to be implementation bugs with the exception of the malicious file upload, which was a design flaw. All of the vulnerabilities found with exploratory manual penetration testing were caused by lack of input validation. Since

[6] http://getfirebug.com/.
[7] http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.
[8] http://www.backtrack-linux.org/.
[9] http://www.wireshark.org/.
[10] https://www.fortify.com/products/fortify360/index.html.
[11] http://www-01.ibm.com/software/awdtools/appscan/.

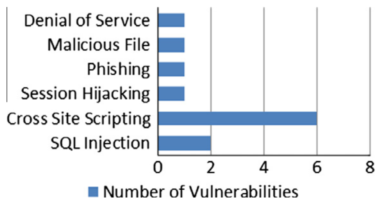

Fig. 1. Vulnerabilities found in OpenEMR with exploratory manual penetration testing (as reported in [2]).

exploratory manual penetration testing found 12 vulnerabilities in 30 h, the efficiency metric is 0.40 vulnerabilities per hour.

In the exploratory penetration testing effort against PatientOS, the second author spent 14 h examining the client-side of PatientOS. The only vulnerability found was the transfer of unencrypted traffic between client and server that contained sensitive data. Specifically, when a nurse created a new office visit, a JavaBean containing the patient's name, SSN, insurance policy information, diagnoses, drug allergies, etc. was transmitted in plain-text to the server. If the server was physically separated from the client machine (as it likely would be), then this sensitive data would likely have to cross untrusted networks, from which anybody with a packet sniffer could collect it. In a man-in-the-middle attack, a malicious user on the client-side network could potentially alter the data before forwarding it to the server via ARP-cache poisoning or a similar technique. The single vulnerability found in the 14 h yields a vulnerabilities discovered per hour value of 0.07.

5.2. Systematic manual penetration testing

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

OpenEMR failed 63 of 137 tests. Fig. 2 breaks down the vulnerabilities found in OpenEMR with systematic manual penetration testing. Since the authors found 63 vulnerabilities in 67 h, the vulnerabilities per hour metric is 0.94. Also note that this number maybe be low as the 60 h number given by the authors included time for the evolution of their methodology.

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

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

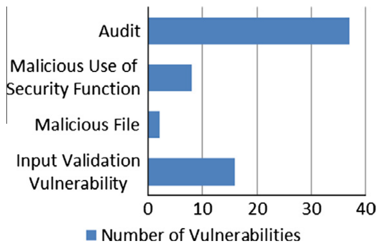

Fig. 2. Vulnerabilities found in OpenEMR with systematic manual penetration testing (as reported in [2]).

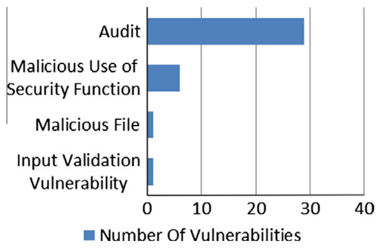

Fig. 3. Vulnerabilities found in Tolven eCHR with systematic manual penetration testing (as reported in [2]).

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

PatientOS also failed 37 of the 137 tests. This gives it the same vulnerability discovery rate as Tolven: 0.55. Fig. 4 shows the vulnerabilities found broken up by categories.

Almost all of the vulnerabilities found with systematic manual penetration testing in PatientOS were audit related. PatientOS failed to properly log critical events in the system. PatientOS also failed to maintain good password policy, especially in regards to the admin account. All of the vulnerabilities found were classified as design flaws.

In all subjects, systematic manual penetration testing found mostly design flaws, but it also found several implementation bugs. Since both types of vulnerabilities occur with roughly equal frequency, having a technique that finds both could be useful [1]. Ultimately, however, the use of just one technique will not be enough.

5.3. Automated penetration testing

Running AppScan on Tolven eCHR resulted in 37 security issues after roughly 8 h of unattended scanning. Approximately 1 h was spent going through the 37 potential vulnerabilities. Only 22 of these 37 issues were true positives, giving a 40% false positive rate. Since we found 22 true positives in 1 h of evaluation, the vulnerabilities per hour metric is 22.00. Table 2 provides our results.

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

AppScan found 735 potential vulnerabilities in OpenEMR after six and a half hours scanning. We spent roughly 10 h going through all of these issues and classifying them as either true or false positives. After classification, 710 true positives remained from the 735 potential vulnerabilities, giving a false positive rate of 3%. We found 710 true positive vulnerabilities in 10 h of evaluation, giving us a vulnerabilities per hour metric of 71.00.

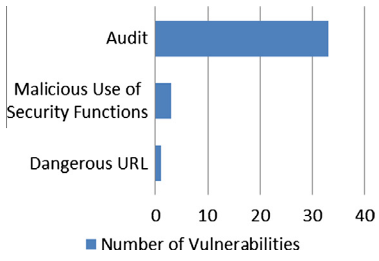

Fig. 4. Vulnerabilities found in PatientOS with systematic manual penetration testing (as reported in [2]).

Table 2
automated penetration test vulnerabilities in Tolven eCHR (as reported in [2]).

Type	True positives	False positives	False positive rate (%)
Session identifier not updated	0	3	100
Cross site request forgery	0	1	100
Cacheable SSL page	0	9	100
Missing HttpOnly attribute	5	0	0
System information leak	17	0	0
Email address pattern	0	2	100
Total	22	15	40

Table 3
Automated penetration testing vulnerabilities in OpenEMR (as reported in [2]).

Type	True positives	False positives	False positive rate (%)
Cross site scripting	7	0	0
SQL injection	214	0	0
System information leak	467	0	0
Directory traversal	18	0	0
Email address patterns	0	5	100
Missing HTTP only attribute	4	0	0
HTML information leak	0	3	100
JavaScript cookie manipulation	0	6	100
Phishing through frames	0	8	100
Session ID not updated	0	1	100
Unencrypted login	0	2	100
Total	710	25	3

Table 3 shows the breakdown of the type of vulnerabilities found. Automated penetration testing did particularly well at finding input validation vulnerabilities such as SQL injection, XSS, and Error Information Leak vulnerabilities. Of these three types of vulnerabilities, the false positive rate was 0%.

We were unable to run AppScan against the PatientOS client because PatientOS was not a web-application. The DOM elements of a web-application user interface can easily be parsed and modified, unlike the user interface of a custom client. Many different GUI libraries exist for custom clients, resulting in no single interface with which to access or manipulate the components of the user interface across applications.

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

5.4. Automated static analysis

Automated static analysis for Tolven eCHR generated a list of 3765 potential vulnerabilities. Despite only scanning for security issues, there were 1450 issues reported had no security implications. For example, Fortify 360 reported "J2EE Bad Practices" and "Code Correctness" issues even after explicitly scanning only for security issues. Removing the non-security issues reported by Fortify 360 resulted in a total of 2315 potential vulnerabilties.

We spent about 18 h manually classifying these potential vulnerabilities as either true or false positives. Speed of classification was greatly enhanced by the Fortify 360 user interface. Lines containing potential vulnerabilities could be viewed with a single click and vulnerabilities in a single file could also be grouped. The speed of classification was also influenced by the similarity and quantity of false positives. For example, many XSS vulnerabilities had similar structure and layout, so the analysis involved checking for differences in a common pattern and determining how those differences influenced the potential vulnerability. Because of these similar issues, it could take 5–10 s to evaluate a line of code in some cases, or up to several minutes for more complicated issues. After pruning for false positives, 50 true positive vulnerabilities were identified giving a 98% false positive rate. We found 50 true positives in 18 h of testing, for a vulnerabilities per hour measurement of 2.78. Table 4 breaks down the vulnerability types discovered and their false positive rates.

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

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

Table 4
Static analysis vulnerabilities in Tolven eCHR (as reported in [2]).

Type	True positives	False positives	False positive rate (%)
SQL injection	5	24	83
Cross site scripting	28	182	87
System information leak	13	441	97
Header manipulation	2	1	33
File upload abuse	2	0	0
Weak cryptography or randomness	0	111	100
Weak access control	0	225	100
Command injection	0	2	100
Denial of service	0	57	100
J2EE misconfiguration	0	19	100
LDAP issues	0	28	100
HTTP verb tampering	0	2	100
JavaScript hijacking	0	39	100
Log forging	0	114	100
Deprecated method	0	18	100
Misused authentication	0	2	100
Password management	0	337	100
Path manipulation	0	151	100
Poor logging	0	218	100
Privacy violation	0	31	100
Race condition	0	31	100
Resource injection	0	9	100
Setting manipulation	0	21	100
Trust boundary violation	0	10	100
Unsafe reflection	0	19	100
Weak XML schema	0	173	100
Total	50	2265	98

Table 5
Static analysis vulnerabilities in OpenEMR (as reported in [2]).

Type	True positives	False positives	False positive rate (%)
SQL injection	984	12	1
Cross site scripting	171	3138	95
System information leak	29	56	66
Hidden fields	119	15	11
Path manipulation	7	86	92
Dangerous function	7	0	0
HTTPOnly not set	1	0	0
Dangerous file inclusion	2	110	98
File upload abuse	1	8	88
Command injection	0	44	100
Insecure randomness	0	23	100
Password management	0	36	100
Header manipulation	0	17	100
Other	0	170	100
Total	1321	3715	74

Fortify 360 generated a list of 5036 potential vulnerabilities when used to analyze OpenEMR. We spent approximately 40 man hours going through all the potential vulnerabilities classifying them as either a true positive or a false positive. After pruning false positives, 1321 true positive vulnerabilities were identified giving a false positive rate of 74%. With static analysis we found 1321 true positives vulnerabilities in 40 h. This gives us a vulnerabilities discovered per hour metric of 32.40. Table 5 summarizes the vulnerability types discovered and their false positive rates.

Static analysis was able to find 984 SQL injection vulnerabilities in OpenEMR. OpenEMR uses a custom method that has insufficient input validation to execute all database queries. To determine if an invocation was vulnerable, we only had to look at the method invocation parameters. This significantly sped up the time to evaluate SQL injection potential vulnerabilities. Static analysis also reported 3309 XSS issues in OpenEMR. While 171 of these issues were true positives, the vast majority of them were not. The tool failed to correctly understand that input validation prevented the XSS.

Fortify 360 reported 12,333 issues in total after scanning PatientOS. The majority of them were unrelated to security and could be removed. For example, 10,029 were broad or narrow error-handling blocks; after verifying that no security-critical functions relied on catching errors, the entire group was able to be dismissed. Filtering for only the security-related potential vulnerabilities left 1643 issues to be analyzed. We spent 13 h classifying the results, leaving 145 true positives. This gives Fortify a 92% false positive rate against PatientOS.

Table 6 contains the vulnerability types and false positive rate of the issues discovered by Fortify 360 in PatientOS.

Of the 145 issues that turned out to be true positives, most were related to JRE and client/server JNI injection vulnerabilities. Also, a number of denial-of-service possibilities were highlighted by Fortify. 145 true positives in 13 h gives a vulnerabilities discovered per hour metric of 11.15.

6. Analysis and discussion

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

Table 6
Static analysis vulnerabilities in PatientOS.

Type	True positives	False positives	False positive rate (%)
Unsafe mobile code	46	29	39
Password management	0	330	100
Privacy violation	33	255	89
Redundant null check	14	4	22
Denial of service	0	126	100
Path manipulation	11	139	93
Unsafe JNI	2	0	0
Weak cryptographic hash	0	0	100
Missing check for null parameter	1	0	0
Missing XML validation	1	0	0
Access control	0	1	100
Command injection	0	3	100
Dead code	0	235	100
Insecure randomness	0	15	100
Log forging	4	3	43
Object model violation	0	2	100
Obsolete	0	1	100
Misused authentication	0	3	100
Poor logging practice	18	14	44
Race condition	0	5	100
Resource injection	0	14	100
Setting manipulation	0	3	100
SQL injection	0	23	100
System information leak	0	23	100
Unchecked return value	0	16	100
Unreleased resource: streams	15	107	88
Unsafe reflection	0	4	100
Weak XML schema	0	288	100
Total	145	1644	92

6.1. Comparing vulnerabilities discovered

In this section, we provide results that aggregate the specific vulnerabilities discovered in the EHR systems. To gain a better understanding of when to use each type of discovery tool, we compare how effective one discovery tool was at detecting the specific vulnerabilities found with other tools. We compared each vulnerability found with the other discovery techniques to every vulnerability we found with static analysis. We chose to compare everything to static analysis because it reported the greatest number of true positives.

First, we compared the vulnerabilities we discovered with static analysis to every vulnerability found with the other discovery techniques. A breakdown vulnerabilities found with static analysis compared to all the other discovery techniques can be found in Table 7. The second column represents the unique number of vulnerabilities found of each particular class using static analysis, while the third through fifth columns represents how many of those vulnerabilities were discovered with each corresponding vulnerability discovery technique.

The details of Table 7 will be discussed in each of the following subsections.

6.1.1. Exploratory manual penetration testing

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

Static analysis was able to find all the SQL injection vulnerabilities found by exploratory manual penetration testing. However, static analysis was able to only find three of the XSS vulnerabilities out of six. Other types of vulnerabilities found with manual testing were not discovered with static analysis. Other static analysis tools would not likely be able to find these issues either; they occur due

Table 7
Vulnerabilities found in static analysis compared to all other discover techniques (adapted from [2]).

Vulnerability type	Static analysis	Manual testing	Systematic test plans	Automated testing
SQL injection	989	2	1	0
Cross site scripting	199	3	5	5
System information leak	42	0	0	0
Hidden fields	119	0	0	0
Path manipulation	18	0	0	0
Dangerous function	7	0	0	0
No HTTPOnly attribute	1	0	0	0
Dangerous file inclusion	2	0	0	0
File upload abuse	3	0	0	0
Header manipulation	2	0	0	0
Password management	55	0	0	0
Privacy violation	33	1	0	0
Redundant null check	14	0	0	0
Unsafe JNI	2	0	0	0
Total	1486	6/1486	6/1486	5/1486

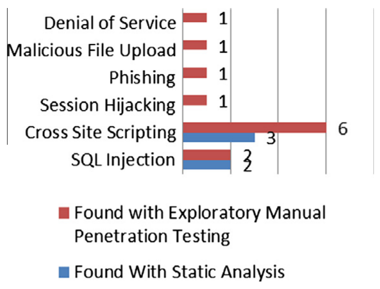

Fig. 5. Vulnerabilities found in both exploratory manual penetration testing and static analysis (as reported in [2]).

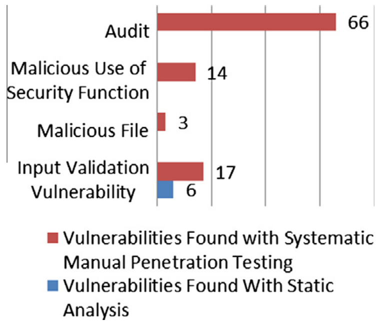

Fig. 6. Vulnerabilities found in both systematic manual penetration testing and static analysis (as reported in [2]).

to the interaction between application components (e.g. browser, server configuration, etc.). These results suggest that only doing static analysis and not some form of black box testing potentially leaves many types of vulnerabilities undiscovered. Similarly, automated penetration testing was unable to find any of the issues discovered by static analysis. Fortify found a number of privacy violations in PatientOS, but none of them were related to the one found during the exploratory manual penetration test.

Static analysis can only find vulnerabilities that exist in the source code (errors of commission), not code that is missing (errors of omission). Other techniques are better suited to finding design flaws, such as errors of omission.

6.1.2. Systematic manual penetration testing

A comparison in the types of vulnerabilities from the EHR systems found with systematic manual penetration testing and static analysis can be found in Fig. 6.

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

Systematic manual penetration testing also found more vulnerabilities than exploratory manual penetration testing. Systematic manual penetration testing found all of the vulnerabilities discovered by exploratory manual penetration testing in OpenEMR. The systematic manual test plan also found vulnerabilities in Tolven eCHR even though exploratory manual penetration testing did not. None of the tools or the systematic manual test plans found the privacy violation in PatientOS data transmittal.

6.1.3. Automated penetration testing

A breakdown in vulnerabilities found with automated penetration testing compared to static analysis can be found in Fig. 7. With automated penetration testing, we discovered seven XSS vulnerabilities. Using static analysis we were only able to find five of these seven vulnerabilities. No other vulnerabilities were found by both automated penetration testing and by static analysis. Static analysis did find many vulnerabilities of the same type, but they were not the same vulnerabilities as automated penetration testing and often not even in the same file. One example of this would be the SQL injection class of vulnerabilities. Static analysis was able to find 989 of these vulnerabilities, but they were not the same individual vulnerabilities found with automated penetration testing. These results suggest that using just static analysis or automated penetration testing would be insufficient in discovering the vast majority of vulnerabilities.

6.2. Vulnerabilities per hour

To understand the efficiency of the vulnerability discovery techniques, we calculated the time it took, on average, to discover a vulnerability with each technique. Where tools are involved, we only account for the time spent evaluating the results of the tool. The installation time of a tool should not reflect on its ability to detect vulnerabilities. Additionally, the cost of computer time is

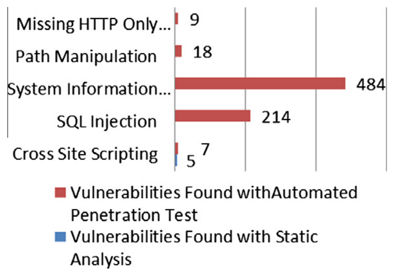

Fig. 7. Vulnerabilities found in both automated penetration testing (as reported in [2]).

Table 8
Efficiency of vulnerability discovery techniques (adapted from [2]).

Discovery technique	Vulnerabilities per hour		
	Tolven eCHR	OpenEMR	PatientOS
Exploratory manual penetration testing	0.00	0.40	0.07
Systematic manual penetration testing	0.94	0.55	0.55
Automated penetration testing	22.00	71.00	N/A
Static analysis	2.78	32.40	11.15

negligible compared to that of a paid man-hour, so we chose not to include the time spent running a tool, only evaluating its output.

Table 8 lists the efficiency calculations for each vulnerability discovery technique. Since Smith and Williams [21] only provided a range we took the average of the times for each evaluation. Also it should be noted that the majority of the time spent for systematic manual penetration testing is spent on the creation of the test plan, rather than the actual testing of the application.

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

7. Limitations

Runeson and Höst outline four primary threats to validity that can help to determine the extent to which the results of a case study might be influenced by researcher bias [24]. **Construct validity** pertains to whether or not the measured results really serve to describe the constructs that they are intended to describe. We believe that our vulnerability data is perfectly descriptive of what we intended. **Internal validity** is threatened when chains of causality are hypothesized (e.g. A causes B), but the possibility of invisible factors (e.g. C might also affect B) is not addressed. Our results are not concerned with causality, so this is not a concern for this study. **External validity** describes the validity of any generalizations made from the data. All of our systems were open-source EHRs, but we do not believe this prevents us from generalizing our results. **Reliability** is concerned with how dependent the data is on individual researchers. All of the potential vulnerabilities were discovered by tools, but their classification was performed by two separate authors.

When comparing vulnerability discovery techniques, the logical construct to use is number of vulnerabilities. We distinguish between false positives and real vulnerabilities to account for the inability of automated tools to actually determine whether a vulnerability is exploitable. We believe that our vulnerabilities per hour metric accurately describes the number of exploitable vulnerabilities discovered per man-hour. Hence, we dismiss construct validity as a significant concern.

Having taken a top-down approach to comparing different vulnerability discovery techniques, we were not concerned with any form of causality. In a bottom-up approach, however, exploring *why* different techniques discover different vulnerabilities, internal validity would be a serious concern.

Our case study subjects were chosen to represent the population of developed code bases. Our choices were limited to three open-source EHR systems. This is a threat to external validity, but we believe that EHR systems generalize to the intended population. The systems we analyzed were written in a variety of widely used languages not specific to health record systems.

The open-source component present in each of our cases poses the question: can open-source software be generalized to the same superset of code-bases as corporately developed software, or is it a different breed of code-base entirely? The open-source community has proven many times that it is capable of producing quality and secure software since the first days of the Free Software Foundation.[12] The three EHRs analyzed are in use all over the United States and corporate entities exist solely for the purpose of providing commercial support for them.

However, the tools we selected to represent automated static analysis and automated penetration testing may not be representative of the other tools available which perform similar services. Also, the results of two tools performing the same technique may differ on the same code-base. Our case study only utilized a single tool to represent each vulnerability discovery technique. Additionally, the architectural differences between the web-applications and the custom client prevented us from being able to run AppScan on all of our subjects.

In any case study, reliability of data collection is always a significant threat to the validity of the subject. There is the potential for misclassification of a true/false positive when analyzing the results of automated static analysis. The sheer number of potential vulnerabilities identified by Fortify360 lessened the statistical impact of any single misclassification for the purposes of this study. Additionally, human error could significantly have affected the reliability of the results. In an effort to minimize these concerns, the first and second authors collaborated to verify agreement on samples of potential vulnerabilities.

Finally, the authors performing the manual exploratory penetration testing were not professionals. The authors were experienced in application security; however, a team of professional penetration testers likely would have uncovered more of the vulnerabilities that were missed in the manual exploratory penetration tests of our case study.

8. Conclusion

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

[12] http://www.fsf.org/.

tion testing and finally manual penetration testing. These results do not refute the opinions of McGraw discussed in the introduction. However, the results do suggest that if development teams have limited time, they should conduct automated penetration testing to discover implementation bugs and systematic manual penetration testing to discover design flaws. All identifiable vulnerabilities should be removed from a system, regardless of their nature; based on this we conclude that no one technique is enough for the identification of software vulnerabilities.

9. Future work

There are several areas branching from our work that could benefit from further study. This paper approached the question of vulnerability discovery with a top-down approach, ultimately allowing us to draw comparisons between the techniques. A bottom-up approach could reveal more about the ability of techniques to detect the exact same vulnerabilities. Additionally, it would be beneficial to know why different types of vulnerabilities were discovered by one technique and not another. Even within a particular type of vulnerability (e.g. XSS), static analysis and penetration testing found different vulnerabilities. Future work could look to determine why different vulnerabilities of the same type were found with automated penetration testing and static analysis. Further study could also examine a greater variety of discovery techniques available for desktop applications.

In addition to researching new ideas and directions, the threats to the validity of the original research could be reduced by performing a systematic review of the decisions made against a sample of potential vulnerabilities. Additionally, the original developers of the systems could be contacted to help verify the true positives identified.

Acknowledgment

This work is supported by the Agency for Healthcare Research Quality. We also would like to thank the members of the Realsearch group for their invaluable feedback on our research and this paper.

References

[1] G. McGraw, Software Security: Building Security, Pearson Education, Boston, USA, 2006.

[2] Andrew Austen, Laurie Williams, One technique is not enough: an empirical comparison of vulnerability discovery techniques, in: International Symposium on Empirical Software Engineering and Measurement, 2011, pp. 97–106.

[3] G. Stoneburner, A. Goguen, A. Feringa, Risk management guide for information security systems. NIST Special Publication 800-30. July 2002. <http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>.

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

[5] B. Chess, G. McGraw, Static analysis for security, IEEE Security and Privacy 2 (6) (2004) 76–79.

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

[7] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs, in: IEEE Software, vol. 25, no. 5, pp. 22–29, 2008.

[8] T. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Software verification with BLAST, in: Proceedings of the 10th International Conference on Model Checking Software (SPIN'03), Springer-Verlag, Berlin, Heidelberg, 2003, pp. 235–239.

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

[10] The MITRE Corporation, Common Weakness Enumeration, March 2011. <http://cwe.mitre.org/>.

[11] N. Antunes, M. Vieira, Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services, in: 15th IEEE Pacific Rim International Symposium on Dependable Computing, Shanghai, 2009, p. 301.

[12] A. Doupè, M. Cova, G. Vigna, Why Johnny Can't Pentest: an analysis of black-box web vulnerability scanners, in: Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Bonn, 2010.

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

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

[15] D. Baca, K. Petersen, B. Carlsson, L. Lundberg, Static code analysis to detect software security vulnerabilities – does experience matter?, in: International Conference on Availability, Reliability and Security (ARES '09), Fukuoka, 2009, p. 804.

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

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

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

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

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

[21] B. Smith, L Williams, Systematizing security test planning using functional requirements phrases, North Carolina State University, Raleigh, Technical Report TR-2011-5, 2011.

[22] B. Smith et al., Challenges for protecting the privacy of health information: required certification can leave common vulnerabilities undetected, in: Security and Privacy in Medical and Home-care Systems (SPIMACS 2010) Workshop, Chicago, 2010, pp. 1–12.

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

[24] Per Runeson, Martin Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164.