Radboud University of Nijmegen

Faculty of Science

Bachelor's Thesis

presented by

Geert Smelt

January 23, 2011

Programming web applications securely

Supervisors

Robert J. van Manen — Director of Forus-P B.V.
Erik Poll — Professor at Radboud University of Nijmegen

www.manaraa.com

Contents

1 Introduction 3

2 Web Applications 5

3 Code scanning 6
- 3.1 What is code scanning? 6
- 3.2 History 6
- 3.3 Why CodeSecure? 7

4 Penetration testing 11
- 4.1 What is automated penetration testing? 11
- 4.2 History 11
- 4.3 Why Hailstorm? 13

5 Comparison of code scanning and pen testing 15
- 5.1 Code scanning vs. (automated) pen testing 15
- 5.2 CodeSecure vs. Hailstorm 16

6 Experiments 18
- 6.1 General results 19
- 6.2 Cross-Site Scripting 24
- 6.3 SQL Injection 32
- 6.4 Conclusions of experimenting 36

7 Combining code scanning and automated pen testing 39

8 Future Work 41

9 Conclusions 42
- 9.1 The project in general 42
- 9.2 The tools used 43

References 46

www.manaraa.com

1 Introduction

When programming a web application it is generally a good idea to program it securely. A rule of thumb is that an application that is accessible by its users from the internet, usually is also accessible in ways you do not want as a developer. Take for instance the example of a web shop. Customers of the web shop generally do not like their credit card information being obtainable by anyone except the website's owners. As a result, the application will need a form of security to gain the trust of the would be customers.

In this thesis I am going to investigate some of the tools available for a security programmer to ensure that the application he is developing is safe enough to be placed on the web. I will do so by comparing examples of two kinds of tools:

- source code scanners (hereafter named 'code scanners')
- automated penetration testing tools (hereafter named 'automated pen testers')

Both kinds of tools focus on a different aspect of web application security. It seems useful to combine both types of tools in order to strengthen a web application's security. I will investigate the possibility of combining these two types of tools. The reason I do this is because it is currently impossible to do all of the security testing with with software only, let alone with one tool in particular. A combination of these tools could mean a developer would have less trouble in programming a secure web application, because most of the vulnerabilities present in web applications are too specific for an automated penetration tester to find by itself. These tools however do present the user with some information on the security level of his application, just not nearly enough to skip manual testing altogether. However, I will not specify all the attack possibilities on a certain web application, because you can read all about that in books like "19 Deadly Sins of Software Security"[8].

Before I can try combining the two methods of scanning for vulnerabilities in web applications, some things need an introduction or explanation. In chapter 2 I will first give a short definition of web applications. Afterwards I will be describing the tools and techniques of code scanning and automated pen testing in chapters 3 and 4 respectively. After having explained both methods I will then compare them to each other in chapter 5 to find out the advantages and disadvantages of both. Following this introduction of

www.manaraa.com

the code scanning and pen testing tools and techniques, in chapter 6 I will experiment with both to see just how effective the tools are at finding possible vulnerabilities in an open source web application written in PHP. Topics that will be addressed include the amount of false positives both tools generate, the readability of the reports generated and the difficulty of fixing an actual vulnerability. Finally, in chapter 7 I will investigate the possibility of combining both tools.

www.manaraa.com

2 Web Applications

Before we can elaborate on the security of web applications it is useful to state precisely what we mean by the term. Generally a web application is a piece of software that is accessible through a web browser, making it one of the easier ways for a cross-platform implementation. Web applications are sometimes also referred to as 'Software-as-a-Service' or 'SaaS.' Examples of web applications include services like webmail, online auctioning, retail sales and wikis. Google is a big player in this field with its products Gmail, Google Calendar, Google Docs etcetera.

In his article 'Modeling Web Application Architectures with UML' Jim Conallen gives a detailed definition of web applications. [5] He writes that there are a multiple meanings to the term web applications, for example "some believe a web application is anything that uses Java, others consider a web application anything that uses a web server. The general consensus is somewhere in between." Personally, I do not agree with this 'general consensus'; I think the key aspect of a web application is the fact that it is accessible over the internet and not limited to a single computer. As a result I would define a web application as *an application that is accessible over the internet*. According to Conallen a web application is "a web system (Web server, network, HTTP, browser) in which user input (navigation and data input) effects the state of the business." Conallen also writes that a web application is somewhat like a client/server system, only there are a few key differences. One of these differences lie in the nature of client and server communication. A web application's primary means of communicating is via HTTP, which is designed for fault tolerance and robustness. Communication between a client and server in a Web application typically revolves around the navigation of web pages, not direct communications between server side and client side objects. According to Conallen the architecture of a Web application is, generally speaking, not much different from that of a dynamic Web site. [5]

This type of interaction is another big advantage of web applications. It means the end user does not need to install anything on his computer, and thus avoid loss of disk space. This also means a user is not required to do anything in terms of patching/updating the web application, but only from the web application's server administrator. A downside to this, however, is the fact that connection interruptions usually mean a denial of service, unless some kind of caching is implemented.

www.manaraa.com

3 Code scanning

3.1 What is code scanning?

In short, code scanning is a technique for assessing a web application's security by means of a tool that analyzes the application's source code to find insecurely programmed parts. When you're developing a web application it is very important that you program them securely, especially when you're dealing with sensitive data. To make sure your web application is secure, you will need to do some testing. There are a lot of ways to check whether a web application is secure, two of which are code scanning and automated pen testing as described in the introduction on page 3. The first step that should be taken is code scanning. As you would have guessed a code scanner assesses the source code of a given application. That also means it can be used during the development phase of the web application. This is the reason why it should be done first, i.e. to detect and correct potential security flaws before the application goes live, because as we all know repairing a deployed (web) application is much more expensive than preventing bugs, and in this case security flaws. This is where the code scanners come in. Take for example an application that crashes without providing error messages. It is generally a good idea to handle errors that cause an application to crash and cause a denial of service. If this isn't properly handled a regular user usually does not know what has gone wrong. This flaw can be spotted during code review, but it is quite hard to see yourself. With the help of a code scanner a developer can find and correct the ignoring of errors before the application is fully finished. A code scanner is an automated way of checking the source code of an application for flaws that could lead to potential security breaches.

3.2 History

Initially there was little need for code review. Back then people have been more interested in network security. "Current technologies such anti-virus software programs and network firewalls comparatively secure protection at the host and network levels, but not at the application level. However, when network and host-level entry points are relatively secure, the public interfaces of Web applications become the focus of attacks."[9] Before the code scanners were available you would have to resort to books like "19 Deadly Sins of Software Security"[8] to find out what you could do to fix

www.manaraa.com

the security flaw someone had pointed out to you. With the introduction of code scanners it became a bit easier to check whether the code you had written was secure or not. If it was not secure, the program would point you in the right direction to solve the problem – note that code scanners are tools that aid a (human) code reviewer in his work. It is not a complete replacement for the way source code is reviewed, because a developer still would need a clear understanding as to why the source code was flawed. A code scanner only helped him pinpoint the exact location of a security flaw. Code scanners do also contain a lot of *false negatives*, which means the tool doesn't detect all of the flaws present in the application, and *false positives*, which means the tool detects a flaw that actually isn't one. The code scanner became more of a tool to be used during code review than a complete replacement of the process.

Figure 1: The CodeSecure Verifier

3.3 Why CodeSecure?

Examples of code scanners include CodeSecure, Pixy, Fortify 360 and PHP-SAT. I will not go into details here, but you can read up on the tools yourself to compare them. To do so you could read Nico L. de Poel's master's thesis "Automated Security Review of PHP Web Applications with Static Code Analysis" as it gives a good comparison of Fortify 360, CodeSecure, PHP-SAT and Pixy. [6] Pixy is also described in "Pixy: A static analysis tool for detecting web application vulnerabilities."[11] In my thesis I will make use of Armorize's CodeSecure tool. According to fellow students it's hard

www.manaraa.com

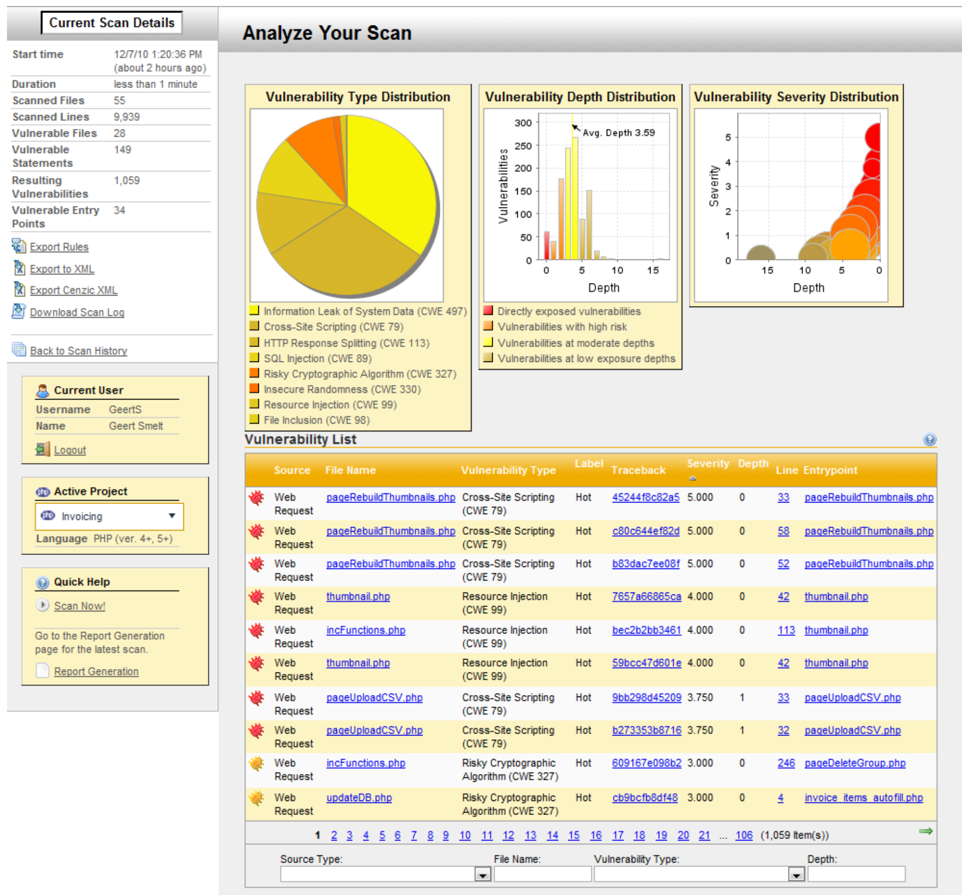


Figure 2: CodeSecure Scan Results Example

to "get access to a trial server running CodeSecure" [12]. The main reason I chose to use CodeSecure as my code scanner is that I have the benefit of working for a company, called Forus-P[1], that already has a license to use it. A thing worth mentioning is that CodeSecure comes in a device with an internet connection, for remote access, and a small hard drive, in order to store source code to be scanned. This device is called the CodeSecure Verifier by its developer Armorize and is shown in figure 1 on page 7. For a general idea about what the results of a scan performed by CodeSecure looks like, please see figures 2 and 3 on pages 8 and 9 respectively.

On their product web page Armorize describe their CodeSecure Verifier as

[1] http://www.forus-p.nl/

www.manaraa.com

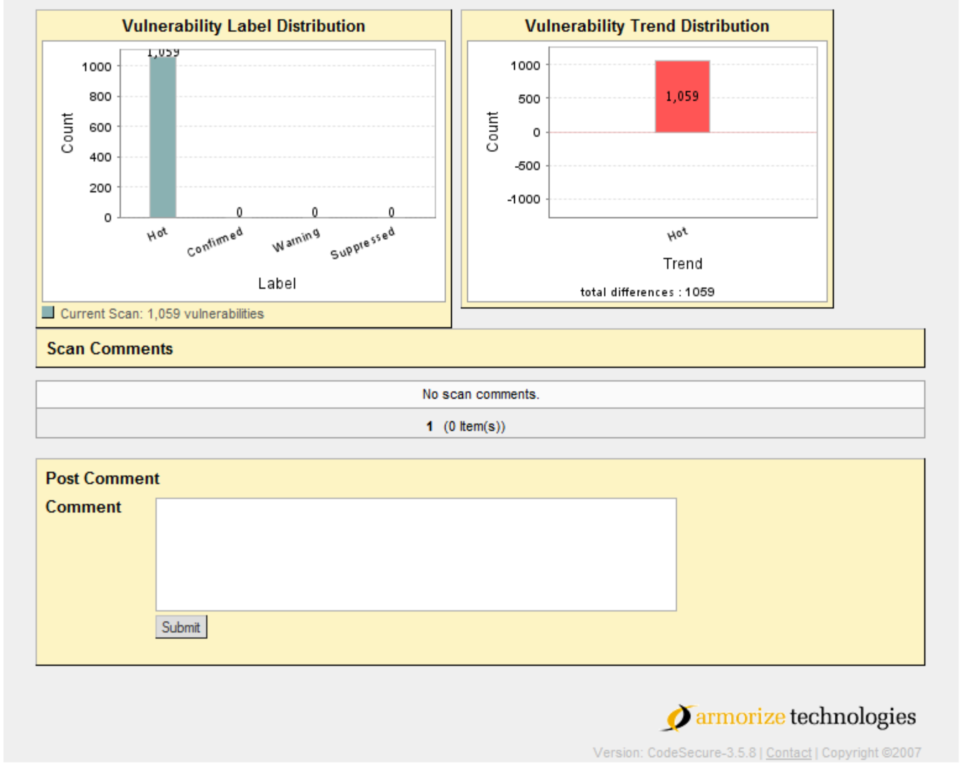


Figure 3: CodeSecure Scan Results Example (cont.)

an appliance that hosts the source code analysis and verification engine. It is being accessed via a web browser so it is a centralized source code analysis platform for developers, managers and security personnel. It can be used simultaneously by multiple users since they all connect to the same device.

Forus-P has a trial license for one of these devices with multiple versions of OWASP's WebGoat uploaded to it, which is a deliberately insecure web application intended for studying purposes, allowing me to perform scans remotely. The basic idea of this device is that you have a server you connect to (the Verifier device) and you upload the source code of the application you want to scan to it. This is being done by means of either ZIP, FTP, SVN, CVS or a Windows Share. Only after you have uploaded the web application's source code to the Verifier, you are able to scan the application for security flaws. At the moment Armorize is in an advanced stage of development of its CodeSecure software that no longer requires the user to

www.manaraa.com

make use of a piece of hardware. The Verifier will then become obsolete, because CodeSecure can then be installed locally. At the time of writing, the release of this product is scheduled to be at the end of 2010. As a result I may be able to use the newer version of their software, but for now I will use the Verifier. Another feature of the Verifier is that it is possible to download a workbench, which is in fact the well-known Eclipse IDE with an extra plugin for code scanning, and scan an application directly from your IDE after you have set up a connection between the IDE and the Verifier. The code scanning plugin is also available for Microsoft's Visual Studio. The plugin helps upload the source code to the Verifier, which in turn directly starts scanning it.

www.manaraa.com

4 Penetration testing

4.1 What is automated penetration testing?

In short, automated penetration testing is a technique for assessing a web application's security by means of a tool that performs all kinds of different attacks on it. As I have stated before in chapter 3, I will discuss two types of automated application scanners: code scanners, described in chapter 3 on page 6, and automated penetration scanners, which I will describe in this chapter. As you would have probably guessed, an automated penetration scanner is a tool that automates the work of a (human) penetration tester. This raises the question: "What is a penetration tester?" A penetration tester, from here on abbreviated as pen tester, is an application developer who tries finding security flaws in an already deployed (web) application. A penetration tester has the advantage of having testing possibilities that a 'regular' code reviewer does not have. For instance the penetration tester is able to fill out forms on a web page, but a code reviewer can only try to see if it is possible for a hacker to enter malicious code into the forms' source code.

4.2 History

In his thesis 'Automated Static Code Analysis - A tool for early vulnerability detection' Dejan Baca writes about the shift in focus during web application development.[2] They claim that during recent years software developers have changed focus from only reliability measurement to include aspects of security threats and risks. Manual audit done by experienced programmers is a time consuming but otherwise efficient method for conducting secure code revision of software. The main reasons they introduce automated auditing tools in their paper are to decrease manual audit time and to integrate automatic tools as part of a revision update. The first issue with this has its background in program checkers (Johnson 1978 [10]) followed by several generations of automated auditing tools, from rule based to more flexible context based. Code scanners use a database of keywords to find vulnerabilities and output a vulnerability report by doing a syntactic matching (Wagner et al. 2000 [13]). These tools report a large number of false positives since they lack a deepened context analysis, and therefore manual examinations to exclude the false positives are necessary. More recent global analysis tools perform an analysis of program semantics (for

www.manaraa.com

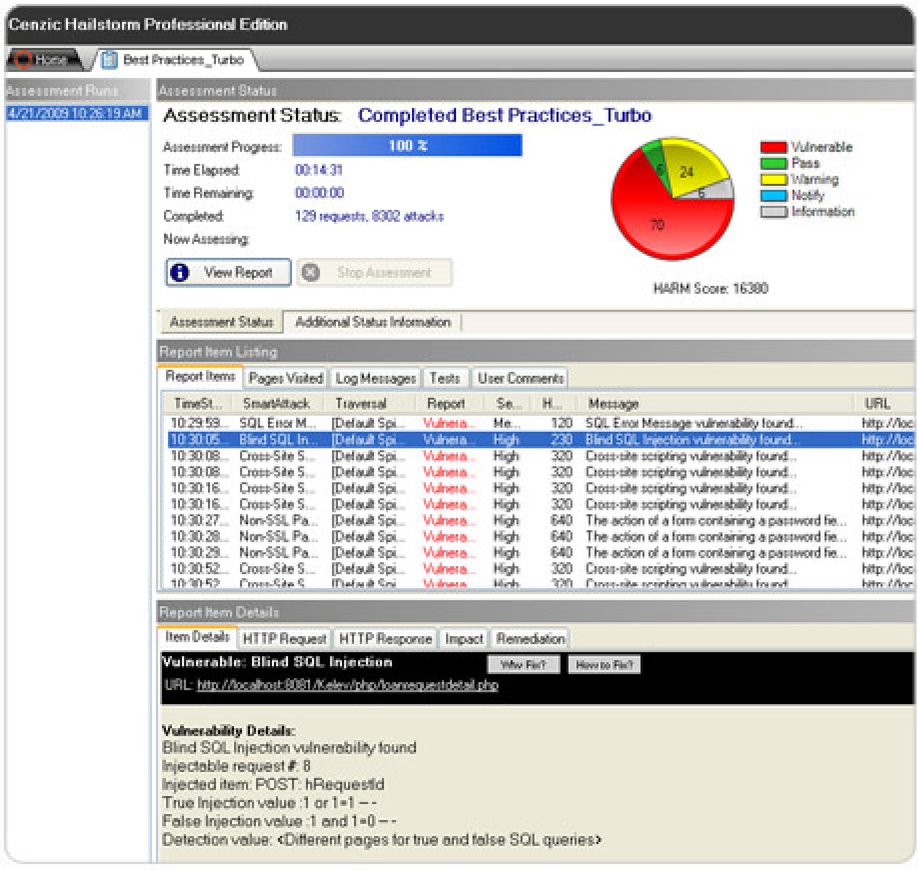


Figure 4: An example of an assessment performed by Hailstorm

an overview see (Chess and McGraw 2004a [4])) in an effort to minimize the amount of false positives. The second issue with it is the possibility to integrate automated tools into the software development life cycle as improvement and effectiveness factors. Development cost savings are part of a more general return on investment (ROI) calculation for the investigated project. For example, as you can imagine, an application that doesn't make use of sensitive data has a lower priority than one that does, in terms of money spent on security assessments.

www.manaraa.com

4.3 Why Hailstorm?

Naturally there are multiple tools that can do the penetration testing for you. Examples of these tools include Acunetix WVS, Cenzic Hailstorm Pro, HP WebInspect, IBM Rational AppScan, McAfee SECURE, N-Stalker QA Edition, QualysGuard PCI, Rapid7 NeXpose and probably a lot more. I will not write up a full comparison of these tools here. If you wish to compare the aforementioned tools you can read the paper 'State of the Art: Automated Black-Box Web Application Vulnerability Testing' by Jason Bau et al.[3] In my thesis I will make use of Cenzic's Hailstorm Pro. The reason for this is, just like is the case with the code scanner, that my employer at Forus-P possesses a license to use it. The reason why Forus-P uses it is that according to Frank Schaap, a former employee who compared HP's WebInspect and IBM's Rational AppScan tools and Cenzic's Hailstorm in an internal document, Hailstorm is a tool that generates significantly less false positives than the other alternatives and it also has more enterprise options than the other two tools.

To give you an idea what the interface looks like see figure 4 on page 12. Just like CodeSecure, Hailstorm outputs its results in a detailed report. For a detailed output of a scan performed with Hailstorm please have a look at figure 10 on page 31.

Cenzic describe Hailstorm Pro as the most accurate software product in the market, when it comes to software security. It features so called SmartAttacks that can be used to test a web application by scanning for a specific security flaw. It works by dragging and dropping them from a list to an application and then pressing the start button. They are also customizable, so you can scan very specifically. These SmartAttacks are updated weekly to stay up to date with the most abused flaws.[2]

Hailstorm also allows for entering login credentials. This allows for scanning from a specific user account, which is useful if you would like to know whether a basic user can access things he should not be able to, i.e. by means of session hijacking.

Something worth noting is the difficulty of obtaining a trial license to see if the product fits you. My employer has been in contact with Cenzic for multiple months, but I have only recently received my trial version download link from them. Until then I was a little worried about it, because I feared

[2] http://www.cenzic.com/products/cenzic-hailstormPro/

www.manaraa.com

I would not get my license in time to use Hailstorm for this thesis. Luckily I did get the license and am now able to scan my web applications with it. The results of these scans can be viewed in chapter 6 on page 18.

www.manaraa.com

5 Comparison of code scanning and pen testing

5.1 Code scanning vs. (automated) pen testing

In this section I will describe the advantages of using code scanning over pen testing and vice versa. The differences listed below are not my personal experience, but come from various sources. After performing experiments with both tools (see chapter 6 on page 18), I will detail my own experience with the two scanning methods, and specifically with Hailstorm and CodeSecure.

1. As you may or may not have guessed from reading chapters 3 and 4, when asked about fundamental differences between the two techniques, the first thing that should come to mind is the way the tools set up a 'connection' to the web applications to be scanned. When performing a code scan, the web application that will be scanned does not need to be up and running for it to be assessed. The reason for this is the code scanning technique solely scans the application's source code for any anomalies in terms of security flaws. Compared to pen testing this is an advantage, since pen testing, be it automated or manual, requires you to have a fully functioning (part of a) web application running on your web server before you are able to do any pen testing whatsoever. This means you can integrate code scanning very early into the development process whereas a pen tester cannot start until at least a portion of the application is indeed running on a server.

2. Code scanning is harder to do. The reason for this is that a code scanner cannot enter values into form fields and submit them. Surely there are things you are able to scan for before putting the web application out there, such as commented login credentials, however it is not fully impossible for a code scanner, either human or tool, to detect possible security flaws in a form. The issue here is not that you cannot find any flaws, but merely the level of difficulty of finding such a flaw by performing a code review. Code reviewing has to cope with finding out where in the source code there is a function that sanitizes requested input before processing it. An automated pen tester has the advantage here, as it can indeed enter values into form fields and observe the rendered responses, without having to look for a function that should sanitize input. However, it is just as impossible for a pen tester to try out every possible input as it is for a code scanner. Take

www.manaraa.com

for example the security flaw of SQL injection. In their paper 'Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks' Fonseca et al. write that to detect SQL injection the code scanner uses the source code to follow all the possible paths and the changes it may go through due to the process of the SQL query text and finally parses the result. However, this technique will probably not find all security flaws because of the source code is generally too complex. In these situations it is preferable to use the pen testing approach as it doesn't look towards the source code, but merely generates input and then checks whether the returned output indicates a security flaw.[7]

3. Another difference between the techniques is the (automated) pen tester's inability to check for things like page name guessing. Usually it is quite hard to find a page when you do not know its name. You could try page names like `/admin.php`, `/login.php` and so on, but you will probably never find all pages you can tamper with. In this case you will not find anything with a pen tester, either human or automated, but a code scanner should pick up on these things as it is able to scan for the files that have been uploaded instead of just guessing their names.

As you see, both code scanning and pen testing have their advantages, thus it seems like there should be no reason to not use them both when you are developing a web application. It is even recommended to use a code scanner during development, as "A late lifecycle penetration testing paradigm uncovers problems too late, at a point when both time and budget severely constrain the options for remedy. In fact, more often than not, fixing things at this stage is prohibitively expensive." [1]

5.2 CodeSecure vs. Hailstorm

Besides the differences between both methods of scanning for vulnerabilities in web applications, there are also a few differences between both tools I have used to experiment with. In this section I will describe those differences, however not as thorough as I did in the previous section, as the differences in scanning methods obviously also apply to the tools that automate the methods.

1. To start off, CodeSecure comes in its own device. The reason for this,

www.manaraa.com

according to a member of Armorize's support staff with whom I spoke, is to prevent pirating. According to him pirating was a very large concern for them, especially because they do much of their business in Asia, where intellectual property rights are not as strongly upheld as for instance Europe or North America. Armorize is also very close to releasing a new version of CodeSecure, only this time as software bundle. Unfortunately I cannot make use of that version, meaning it will have to be saved for future work.

2. The advantage of having a device instead of software is that it is then possible to access the software remotely. The only, although very small, disadvantage of this is that you will need to upload source files to the device every time you want to perform a vulnerability scan. This is not as tedious of a task as you think, due to the IDE plugins that exist (see chapter 3 on page 6).

3. Hailstorm only runs locally or by means of a VMware virtual machine, which in turn allows for a portable install. The reason for this is that it is possible to create a virtual machine with a pre-installed version of Hailstorm on it, and then transfer it to another employee so he can perform scans as well.

4. CodeSecure offers online joint vulnerability reviewing, due to being remotely accessible. This is being done by labeling possible vulnerabilities. Four labels exist: `HOT` (the default setting), `CONFIRMED`, `WARNING` and `SUPPRESSED`. Hailstorm's only way to do vulnerability reviewing is by a single user, and only allows for omission of false positives before generating reports.

As it is impossible to say, for example, how many *true positives* both tools generate before doing any scanning, I will come to that later on in chapter 6 on page 18. There I will discuss things like

- It could be possible that one of the tools is especially good at detecting Cross-Site Scripting flaws and the other does not detect any of those.
- It could also be possible that the tools are especially good at detecting Cross-Site Scripting and SQL Injection, but not much else.
- How hard is it to verify that the security flaw reported by a tool is in fact a flaw?
- How hard is it to fix a reported flaw with the feedback of the tools?

www.manaraa.com

6 Experiments

I will experiment only with two of the most commonly known and most dangerous security flaws – a list of which is also maintained by the OWASP organization [14] – namely SQL Injection and Cross-Site Scripting. This is being done to reduce the amount of work significantly and to prevent this thesis from being too superficial by trying, and failing, to explain every single flaw in detail. In this section I will describe the experiments I have conducted on the Simple Invoicing web application I have found on the internet. It was developed by BigProf with their own standard called 'AppGini' and resulted in around 15,000 lines of code. On their website BigProf makes the bold claim that AppGini generates secure code. According to them you needn't worry whether your code is vulnerable to SQL injection, Cross-Site Scripting, brute force attacks etc. because 'they have done the hard work for you'.[3] I have decided to put this latter claim to the test by using CodeSecure and Hailstorm. The results of these tests can be found in the following subsections.

Testing setup

In the tests that I will describe in the following subsections I have used the hard- and software setup listed below:

- A Dell Inspiron 9400 laptop (hosting the web application that will be scanned).
- XAMPP for Windows (the server software suite).
- Simple Invoicing by BigProf (the web application to be scanned).
- CodeSecure 3.5.8 trial version (the code scanner).
- Hailstorm v6.5 (build 5267) trial version (the automated pen tester).

Security flaws within Simple Invoicing that require administrator rights to abuse them are relatively less interesting than ones that do not, as it is a tad harder to obtain an administrator's username and password and start hacking away than it is to register your own account. Therefore most of the tests are conducted using a non-administrator account, in order to find the more vulnerable entry points into the application. I will however also scan

[3] `http://www.bigprof.com/appgini/`

www.manaraa.com

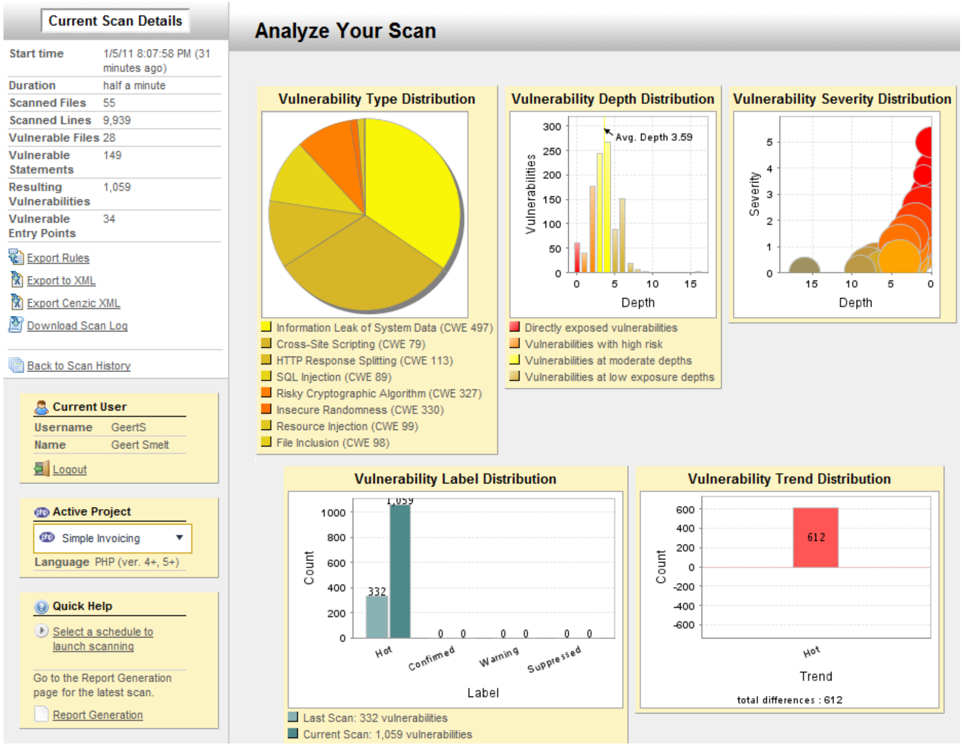


Figure 5: CodeSecure General Scan Results

using an administrator account, but I will just not go into specifics regarding the results of those scans.

6.1 General results

Before going into specifics regarding Cross-Site Scripting and SQL Injection (listed in sections 6.2 and 6.3 respectively), I will first present some general results of scanning Simple Invoicing with Hailstorm and CodeSecure. This is done to give you an idea how flawed the application really is.

Results from CodeSecure

I scanned Simple Invoicing with a 'Default Scan' using CodeSecure. After that scan CodeSecure reported 1059 vulnerabilities, all of which were labeled `HOT`. According to the user manual however, this label only means that

www.manaraa.com

CodeSecure has found a "possible vulnerability, but no manual reviewer has confirmed or discarded it yet." I will come to this in a little bit. For a breakdown please refer to figure 5 on page 19.

The 'Default Scan' performed with CodeSecure includes the following types of vulnerabilities:

- Reflection Injection
- Cross-Site Scripting (322 possible flaws)
- HTTP Response Splitting (120 possible flaws)
- XPath Injection
- Resource Injection (12 possible flaws)
- File Inclusion (1 possible flaw)
- SQL Injection (115 possible flaws)
- Command Injection
- Hard-Coded Password
- Information Leak of System Data (366 possible flaws)
- LDAP Injection
- Open Redirect
- Code Injection
- Risky Cryptographic Algorithm (99 possible flaws)
- Insecure Randomness (14 possible flaws)

The number of possible security flaws found is displayed between brackets. Where there are no values CodeSecure has not found any security flaws. Bear in mind these are only possible flaws and I will not discuss them further, except for Cross-Site Scripting and SQL Injection, in the following sections.

Results from Hailstorm

After performing a 'Best Practices' scan, which includes the seventeen what they call 'SmartAttacks' (see section 4.3 on page 13 for an explanation) listed

www.manaraa.com

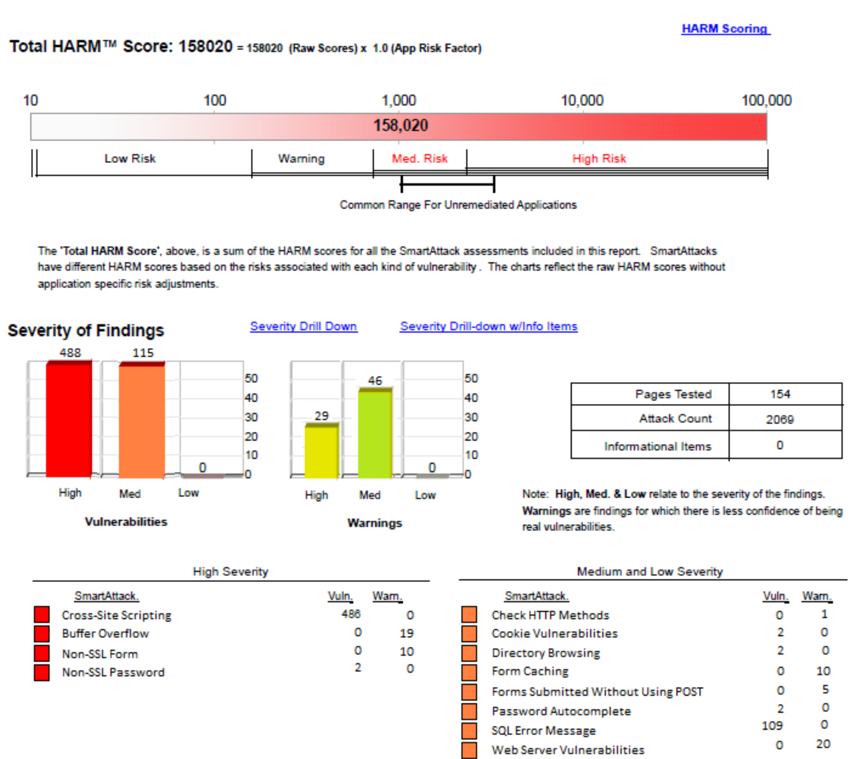


Figure 6: Hailstorm's Best Practices Scan Results Breakdown

below, Hailstorm reported a total number of 678 possible security flaws, of which 603 were listed as 'High Severity' and the other 75 as 'Medium or Lower Severity.' You can see a breakdown of the results in figure 6 on page 21. Most of these flaws have fallen into the categories 'Cross-Site Scripting' and 'SQL Error Message.' The other vulnerabilities and warnings, which add up to around 10% of the total potential vulnerabilities, include 'Form Caching,' 'Non-SSL Forms,' 'HTML & JavaScript Comments' and 'Web Server Vulnerabilities.' The last, however, is merely found because I didn't take my time to set up my XAMPP software suite to protect itself from attacks. I have chosen not to do this, because it would go beyond the scope of this thesis, because I have no real knowledge of such topics and because the application would only be available locally. Detailing all these potential flaws would be too much, so I shall zoom in on the two most common flaws according to OWASP: Cross-Site Scripting and SQL Injection.[14]

The 'Best Practices' scan includes the following SmartAttacks:

www.manaraa.com

- Application Exception
- Blind SQL Injection
- Check HTTP Methods (1 warning, medium severity)
- Cookie Vulnerabilities (2 vulnerabilities, medium severity)
- Cross-Site Scripting (486 vulnerabilities, high severity)
- Directory Browsing (2 vulnerabilities, medium severity)
- File Directory Discovery
- Form Caching (10 warnings, medium severity)
- Forms Submitted without using POST (5 warnings, medium severity)
- HTML JavaScript Comments
- Non-SSL Form (10 warnings, high severity)
- Non-SSL Password (2 vulnerabilities, high severity)
- Open Redirect
- Password Autocomplete (2 vulnerabilities, medium severity)
- SQL Disclosure
- SQL Error Message (109 vulnerabilities, medium severity)
- Web Server Vulnerabilities (20 warnings, medium severity)

Again, between brackets the number of possible security flaws is noted, followed by the severity of the detected flaw.

These SmartAttacks are comparable to what CodeSecure call 'Vulnerabilities.' Since both tools use very differing terms for the possible flaws they find, reading both their outputs may become very confusing. That is why, from here on, I will use the term 'Vulnerabilities' when talking about possible *severe* (i.e. with relatively high impact) security flaws. When either tool reports a vulnerability with a significantly lower impact I will make use of the term 'Warnings.' Hailstorm first distinguishes between high severity, medium and lower severity vulnerabilities. In both of those categories it then distinguishes between vulnerabilities and warnings, resulting in six possible categories to place a possible security flaw in. This is represented in figure 6 on page 21. Like Hailstorm, CodeSecure also uses levels of severity for the reported security flaws. Besides that CodeSecure also uses a 'Vulnerability

www.manaraa.com

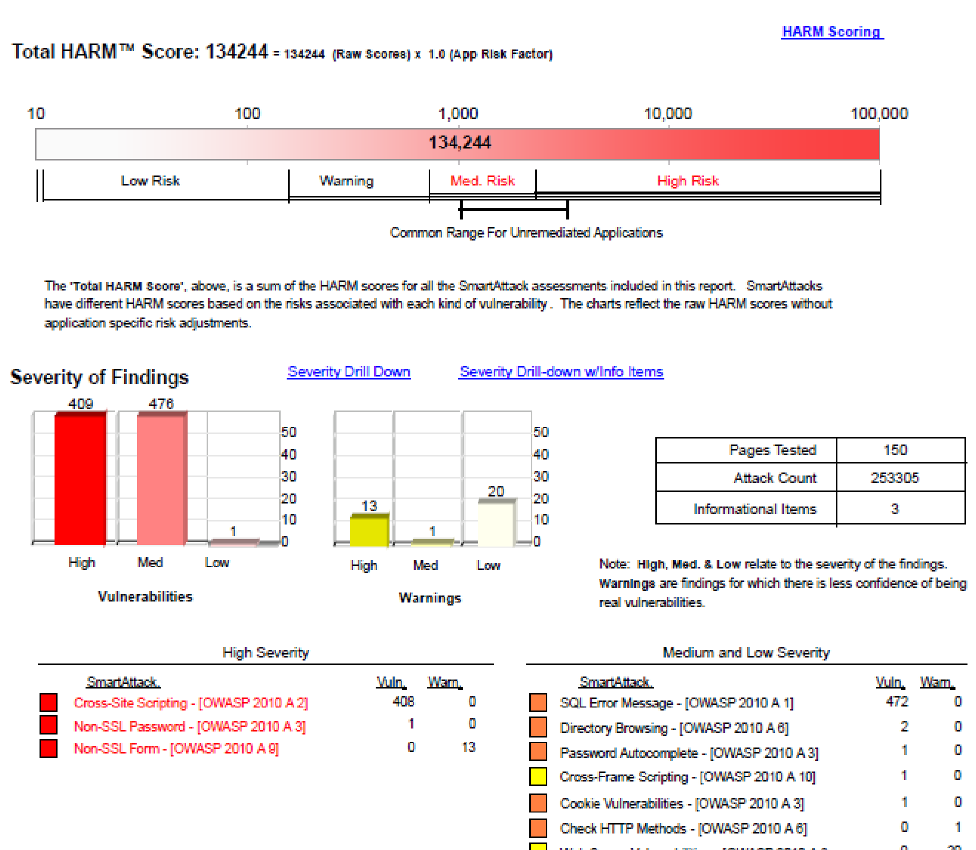


Figure 7: Hailstorm's OWASP Pen Test Results

Depth' metric, which describes the possibility for an attacker to exploit the possible security flaw that was found. As you may suspect, a flaw found deeply inside the administrator's control panel is less deep than one found at the index page of a web application. This is a feature that Hailstorm doesn't have. In addition to this depth metric, CodeSecure also uses labels like `HOT` and `WARNING`. However, this labeling is done by the engineer(s) who review(s) CodeSecure's output. CodeSecure labels all possible security flaws as `HOT` by default. It is then up to the engineer(s) to find out which of the reported items are actual vulnerabilities (labeled `CONFIRMED`), warnings (labeled `WARNING`) and false positives (labeled `SUPPRESSED`). An overview of these metrics can be seen in figure 5 on page 19.

Besides running this Best Practices scan I have also run a scan with all SmartAttacks that OWASP deems vulnerable.[14] This set is different from

www.manaraa.com

the set of attacks performed by the Best Practices scan, as the Best Practices are the opinion of Hailstorm and the OWASP scan focuses on different attacks. For example, in the best practices scan Cross-Frame Scripting is not included, whereas OWASP does include it in their list of most severe attacks (compare figures 6 and 7 on pages 21 and 23 respectively). However, listing these would be superfluous, so I will refrain from doing that. The reason I have also run a full OWASP scan is that I wanted to see how well Simple Invoicing would do against all kinds of possible attacks, rather than just one or two. Another reason is the fact that both tools could be performing well on SQL Injection or Cross-Site Scripting, but very bad on any other attacks. The overview of the OWASP scan's results can be found in figure 7 on page 23. I am not going into detail on this scan's results, because that would go beyond the scope of this thesis. After taking a look at the report, one can clearly see both tools do not *just* find SQL Injection or Cross-Site Scripting flaws, but also find a lot of other things.

Something worth mentioning is the HARM score you see in the aforementioned Hailstorm results. HARM is an abbreviation for Hailstorm Application Risk Metric which is a means of scaling the severity of the vulnerabilities found in the scanned web applications. The higher the HARM-score the graver the vulnerability found. In a way it is comparable to CodeSecure's vulnerability depth metric, but they cannot be translated one on one.

6.2 Cross-Site Scripting

Results from CodeSecure

As you can see in figure 8 on page 29, CodeSecure has found many possible Cross-Site Scripting vulnerabilities in Simple Invoicing. Because there are so many, I will not describe all of them but just highlight one. Although the majority of the flaws have been found in administrator pages, I will not pick one of these flaws, due to the reason stated at start of this chapter, but I will describe one that can be accessed by any user.

For example, one of these flaws has been found in the `invoices_view.php` file. Besides listing the invoices vulnerability, CodeSecure also lists a detailed step by step trace of a way this vulnerability can be exploited. You can see an example of this (shortened) trace in figure 9 on page 30. As you can see there are multiple rows (as many as there are steps in the execution of a function) and three columns. The left column reports the type of

www.manaraa.com

change that occurs by executing the line of code directly below it, the middle column specifies the file in which the line of code is written and executed and the right column lists the function that was invoked in order to get to the line of code currently being examined. This would be ideal information to be able to review directly from your IDE, as it saves you a lot of time switching back and forth. I have tried to set up a connection between the CodeSecure Verifier, which hosts the trial version I was allowed to use, and the Eclipse IDE, unfortunately to no avail. After following the procedure listed in the user manual, a connection from the IDE to the Verifier kept getting refused.

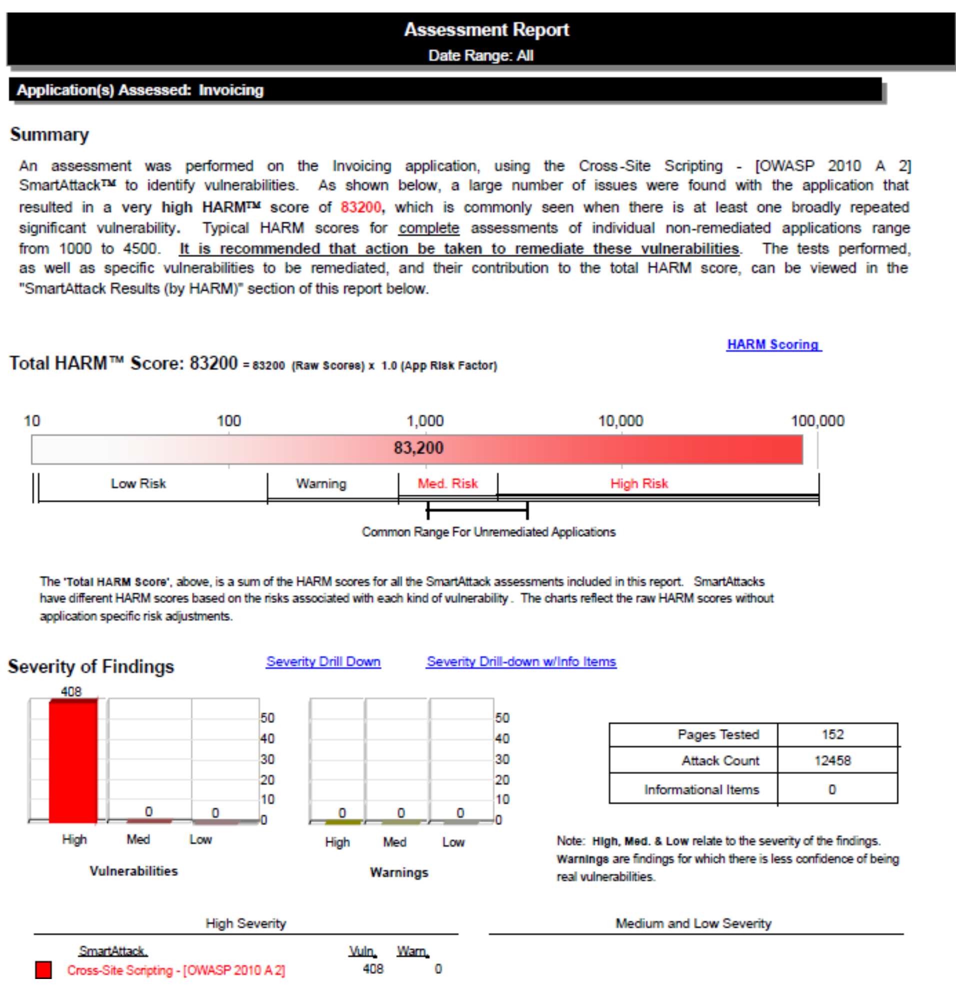
Using these results I set out to look if CodeSecure had reported a false positive, or whether the possible flaw reported was indeed a vulnerability, i.e. a *true positive*. The trace shown in figure 9 has helped me in finding that out. Due to the fact that the trace is so detailed, especially compared to Hailstorm's information, I have been able to confirm that the potential flaw found in the invoices example is indeed a true positive. In the thirteen-step-long trace, of which only the first six are presented, you are given enough information in order to pinpoint the exact location of the possible flaw. It is then up to you to grab a book, or search the internet, in order to find out how to correct this vulnerability, because CodeSecure does not provide you with the details on how to fix it. In case the reported possible flaw is in fact a false positive, an engineer is able to label this item as `SUPPRESSED`, like I previously detailed on page 22. It will then no longer show up in a rescan of the same web application.

Results from Hailstorm

After scanning Simple Invoicing, Hailstorm reported a total of 408 possible Cross-Site Scripting vulnerabilities (see the executive summary in figure 11 on page 26). By looking at the HARM score (mentioned at the end of section 6.1), you will notice that this particular web application is immensely flawed, with just one means of exploiting it. As you can see in figure 10 on page 31 most of these vulnerabilities are of the same kind. This is due to the web page's layout. Most of the web pages with which you add clients, invoices or invoice items contain multiple forms. Due to Simple Invoicing being created with a tool that generates the code instead of a programmer writing it (recall the AppGini standard mentioned at the start of this chapter), most if not all of these forms are programmed in the exact same way, resulting in the

www.manaraa.com

exact same Cross-Site Scripting vulnerability being exploitable in each and every one of them.


Assessment Report
Date Range: All

Application(s) Assessed: Invoicing

Summary

An assessment was performed on the Invoicing application, using the Cross-Site Scripting - [OWASP 2010 A 2] SmartAttack™ to identify vulnerabilities. As shown below, a large number of issues were found with the application that resulted in a very **high** **HARM™** score of 83200, which is commonly seen when there is at least one broadly repeated significant vulnerability. Typical HARM scores for complete assessments of individual non-remediated applications range from 1000 to 4500. **It is recommended that action be taken to remediate these vulnerabilities**. The tests performed, as well as specific vulnerabilities to be remediated, and their contribution to the total HARM score, can be viewed in the "SmartAttack Results (by HARM)" section of this report below.

HARM Scoring

Total HARM™ Score: 83200 = 83200 (Raw Scores) x 1.0 (App Risk Factor)

The '**Total HARM Score**', above, is a sum of the HARM scores for all the SmartAttack assessments included in this report. SmartAttacks have different HARM scores based on the risks associated with each kind of vulnerability. The charts reflect the raw HARM scores without application specific risk adjustments.

Severity of Findings Severity Drill Down Severity Drill-down w/Info Items

Pages Tested	152
Attack Count	12458
Informational Items	0

Note: **High, Med. & Low** relate to the severity of the findings. **Warnings** are findings for which there is less confidence of being real vulnerabilities.

High Severity

SmartAttack	Vuln.	Warn.
Cross-Site Scripting - [OWASP 2010 A 2]	408	0

Medium and Low Severity

Figure 11: Executive summary of Hailstorm's Cross-Site Scripting pen test

Take for instance the page `clients_view.php`. According to Hailstorm, this page alone contains 128 different ways to exploit Cross-Site Scripting flaws. The first six results of the scan Hailstorm performed can be found in figure 10 on page 31. As can be seen there, the same method has been applied in multiple fields. Hailstorm injects multiple instances of the fields `SortField`, `SortDirection`, `SelectedID` and `FirstRecord` with (variants

المنارة للاستشارات
www.manaraa.com

of) the following string:

```
>'><script>alert([insert random string here])</script>
```

If this string is not being escaped by the web application, it could result in a Cross-Site Scripting vulnerability. As you may expect this could have serious consequences for a web application of this type, i.e. one that concerns money transactions.

The flaws Hailstorm reported were indeed actual vulnerabilities and not false positives. All in all it was not really that simple to find out whether the possible vulnerabilities Hailstorm reported were false positives or true positives. To be able to reproduce this vulnerability I had to get help from a colleague of mine, since Simple Invoicing uses JavaScript in order to submit forms. Because I am not really familiar with JavaScript, but more importantly because I am fairly new to the hacking field, I had a lot of trouble reproducing the Cross-Site Scripting vulnerabilities Hailstorm reports. Once I had figured out that this particular vulnerability was not a false positive I tried to find out how an engineer could repair it. Unfortunately, the so called 'assessment reports with remediation' that Hailstorm generates are not detailed enough to be able to pinpoint the location of the reported flaw. The remediation given, does point you in the right direction, but it is not nearly enough to be able to fix the vulnerability on your own with just this information. For example, for PHP, you get the general tip to escape the reserved characters, but not the location where you should do this.

Just like CodeSecure, Hailstorm also reports a list of Cross-Site Scripting flaws in the admin pages of Simple Invoicing. Again, these are less relevant than flaws in the publicly accessible pages of the application, so I will direct you to figure 12 on page 28 if you would like to have a look at the results. Please note that these are the results obtained from scanning only administrator pages. The potential vulnerabilities found in Simple Invoicing when accessed with a regular user's account still apply, even for an administrator.

www.manaraa.com

Cross-Site Scripting - [OWASP 2010 A 2] Vulnerable Findings

Assessment: Interactive XSS, **Run:** 07-01-11 13:58:21, **Traversal:** InteractiveJob

Cross-Site Scripting - [OWASP 2010 A 2]

1. Vulnerable (High, HARM: 320) at: http://localhost/invoicing/admin/pageEditGroup.php?

Message: Cross-site scripting vulnerability found
Injected item: POST: description
Injection value: </textarea><script>alert(12944051.32337)</script>
Detection value: 12944051.32337
This is a reflected XSS vulnerability, detected in an alert that was an immediate response to the injection.

Cross-Site Scripting - [OWASP 2010 A 2] Warning Findings

Assessment: Interactive XSS, **Run:** 07-01-11 13:58:21, **Traversal:** InteractiveJob

Cross-Site Scripting - [OWASP 2010 A 2]

1. Warning (High, HARM: 96) at: http://localhost/invoicing/admin/pageViewGroups.php?page=1&searchGroups=testval

Message: Cross-site scripting vulnerability found
Injected item: GET: searchGroups
Injection value: >'><script>alert(12944051.21947)</script>
Detection value: >'><script>alert(12944051.21947)</script>
Detected in attack response.
Detected with other text in: ONCLICK attribute, found inside: INPUT tag.

Figure 12: Hailstorm's Cross-Site Scripting pen test results of admin pages

www.manaraa.com

Web Request

Label: Hot

Cross-Site Scripting (CWE 79)

Vulnerable Files	Vulnerable Line(s)
invoicing.zip/invoicing/admin/incFunctions.php	303 , 314 , 315 ,
invoicing.zip/invoicing/admin/pageEditGroup.php	131 , 144 , 159 ,
invoicing.zip/invoicing/admin/pageEditMember.php	108 , 115 , 123 , 153 , 164 , 196 , 206 , 216 , 226 , 235 ,
invoicing.zip/invoicing/admin/pageEditOwnership.php	48 , 56 , 67 , 95 , 96 , 102 , 135 , 137 ,
invoicing.zip/invoicing/admin/pageMail.php	120 , 139 , 139 ,
invoicing.zip/invoicing/admin/pageRebuildThumbnails.php	33 , 52 , 58 ,
invoicing.zip/invoicing/admin/pageUploadCSV.php	32 , 33 , 51 , 144 , 146 ,
invoicing.zip/invoicing/admin/pageViewRecords.php	69 , 73 , 125 , 126 , 128 , 129 , 130 , 133 , 142 , 142 , 142 , 145 , 148 , 148 , 148 ,
invoicing.zip/invoicing/membership_passwordReset.php	73 , 98 ,

Database

Label: Hot

Cross-Site Scripting (CWE 79)

Vulnerable Files	Vulnerable Line(s)
invoicing.zip/invoicing/admin/incFunctions.php	303 , 314 , 315 ,
invoicing.zip/invoicing/admin/pageAssignOwners.php	77 ,
invoicing.zip/invoicing/admin/pageEditMember.php	108 , 115 , 123 , 153 , 164 , 196 , 206 , 216 , 226 , 235 ,
invoicing.zip/invoicing/admin/pageEditOwnership.php	48 , 56 , 67 , 95 , 96 , 102 , 135 , 137 ,
invoicing.zip/invoicing/admin/pageHome.php	56 , 56 , 75 , 75 , 94 , 95 , 110 , 114 , 121 , 125 , 129 , 153 ,
invoicing.zip/invoicing/admin/pageMail.php	120 , 139 , 139 ,
invoicing.zip/invoicing/admin/pagePrintRecord.php	49 , 65 , 67 ,
invoicing.zip/invoicing/admin/pageTransferOwnership.php	49 , 63 , 74 , 100 , 114 , 147 , 159 , 160 , 184 , 198 , 213 , 237 , 275 , 289 ,
invoicing.zip/invoicing/admin/pageUploadCSV.php	32 , 33 , 51 , 144 , 146 ,
invoicing.zip/invoicing/admin/pageViewGroups.php	64 , 68 , 75 , 75 , 78 , 81 , 82 , 84 , 85 , 99 ,
invoicing.zip/invoicing/admin/pageViewMembers.php	106 , 140 , 141 , 141 , 156 , 159 , 161 , 165 , 167 , 181 ,
invoicing.zip/invoicing/admin/pageViewRecords.php	69 , 73 , 125 , 126 , 128 , 129 , 130 , 133 , 142 , 142 , 142 , 145 , 148 , 148 , 148 ,
invoicing.zip/invoicing/datalist.php	568 ,
invoicing.zip/invoicing/invoices_autofill.php	25 , 26 , 27 , 28 , 29 , 30 ,
invoicing.zip/invoicing/invoices_view.php	199 ,
invoicing.zip/invoicing/membership_passwordReset.php	73 , 98 ,
invoicing.zip/invoicing/membership_signup.php	142 ,

System Environment

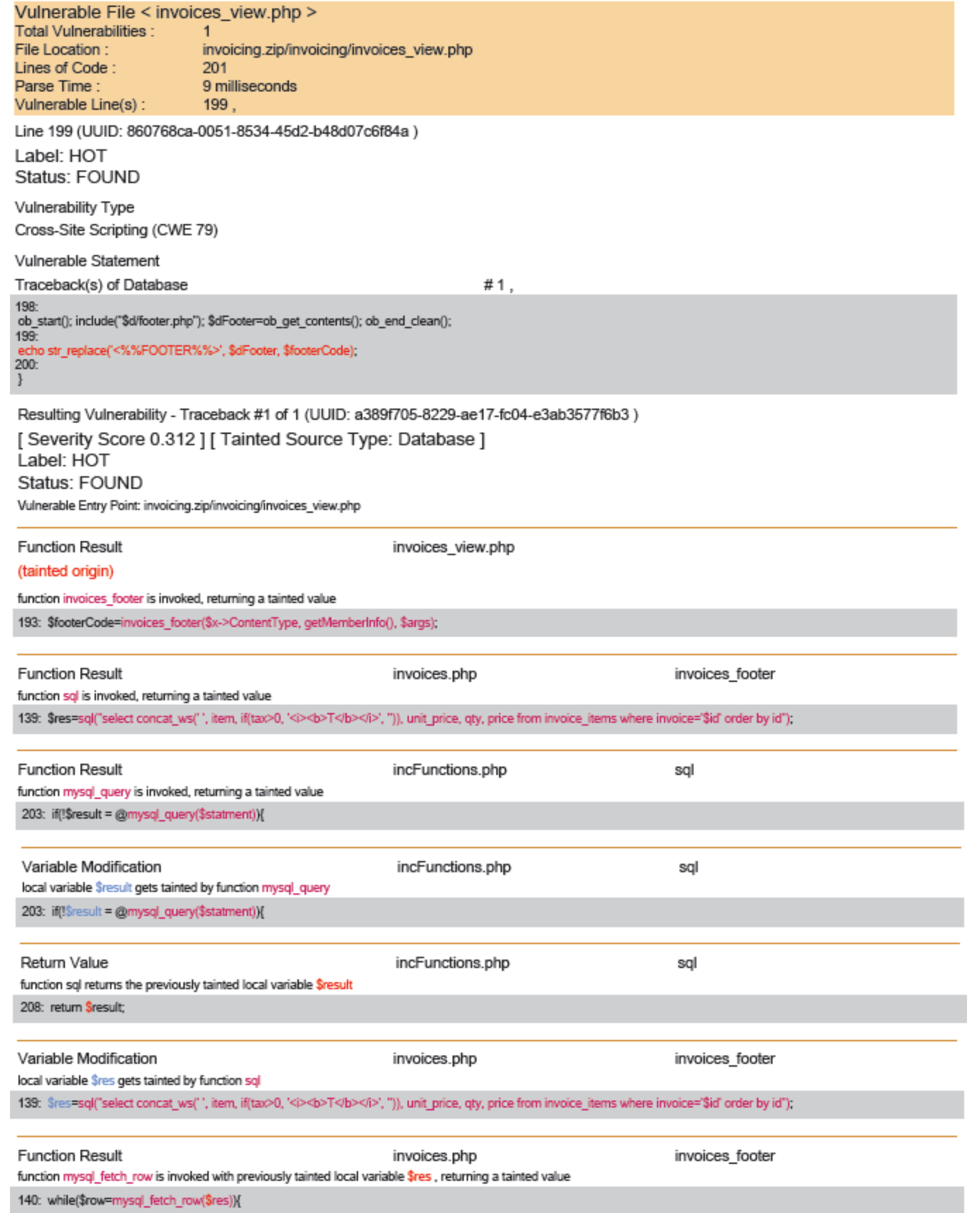
Label: Hot

Cross-Site Scripting (CWE 79)

Vulnerable Files	Vulnerable Line(s)
invoicing.zip/invoicing/admin/incFunctions.php	303 , 314 , 315 ,

Figure 8: CodeSecure's Cross-Site Scripting Results Overview

www.manaraa.com



Vulnerable File < invoices_view.php >

Total Vulnerabilities :	1
File Location :	invoicing.zip/invoicing/invoices_view.php
Lines of Code :	201
Parse Time :	9 milliseconds
Vulnerable Line(s) :	199 ,

Line 199 (UUID: 860768ca-0051-8534-45d2-b48d07c6f84a)

Label: HOT
Status: FOUND

Vulnerability Type
Cross-Site Scripting (CWE 79)

Vulnerable Statement
Traceback(s) of Database # 1 ,

```
198:
 ob_start(); include("$d/footer.php"); $dFooter=ob_get_contents(); ob_end_clean();
199:
 echo str_replace('<%%FOOTER%%>', $dFooter, $footerCode);
200:
 }
```

Resulting Vulnerability - Traceback #1 of 1 (UUID: a389f705-8229-ae17-fc04-e3ab3577f6b3)
[Severity Score 0.312] [Tainted Source Type: Database]
Label: HOT
Status: FOUND
Vulnerable Entry Point: invoicing.zip/invoicing/invoices_view.php

---

Function Result (tainted origin) — invoices_view.php
function invoices_footer is invoked, returning a tainted value

```
193:  $footerCode=invoices_footer($x->ContentType, getMemberInfo(), $args);
```

Function Result — invoices.php — invoices_footer
function sql is invoked, returning a tainted value

```
139:  $res=sql("select concat_ws(' ', item, if(tax>0, '<i><b>T</b></i>', '')), unit_price, qty, price from invoice_items where invoice='$id' order by id");
```

Function Result — incFunctions.php — sql
function mysql_query is invoked, returning a tainted value

```
203:  if(!$result = @mysql_query($statment)){
```

Variable Modification — incFunctions.php — sql
local variable $result gets tainted by function mysql_query

```
203:  if(!$result = @mysql_query($statment)){
```

Return Value — incFunctions.php — sql
function sql returns the previously tainted local variable $result

```
208:  return $result;
```

Variable Modification — invoices.php — invoices_footer
local variable $res gets tainted by function sql

```
139:  $res=sql("select concat_ws(' ', item, if(tax>0, '<i><b>T</b></i>', '')), unit_price, qty, price from invoice_items where invoice='$id' order by id");
```

Function Result — invoices.php — invoices_footer
function mysql_fetch_row is invoked with previously tainted local variable $res , returning a tainted value

```
140:  while($row=mysql_fetch_row($res)){
```

Figure 9: CodeSecure's Cross-Site Scripting feedback for the invoices vulnerability

www.manaraa.com

Cross-Site Scripting - [OWASP 2010 A 2] Vulnerable Findings

Assessment: Interactive XSS, **Run:** 06-01-11 13:18:33, **Traversal:** InteractiveJob

Cross-Site Scripting - [OWASP 2010 A 2]

1. Vulnerable (High, HARM: 320) at: http://localhost/invoicing/clients_view.php?

Message: Cross-site scripting vulnerability found
Injected item: POST: SortDirection
Injection value: >'><script>alert(12943163.6177)</script>
Detection value: 12943163.6177
This is a reflected XSS vulnerability, detected in an alert that was an immediate response to the injection.

Cross-Site Scripting - [OWASP 2010 A 2]

2. Vulnerable (High, HARM: 320) at: http://localhost/invoicing/clients_view.php?

Message: Cross-site scripting vulnerability found
Injected item: POST: SortField
Injection value: >'><script>alert(12943163.6187)</script>
Detection value: 12943163.6187
This is a reflected XSS vulnerability, detected in an alert that was an immediate response to the injection.

Cross-Site Scripting - [OWASP 2010 A 2]

3. Vulnerable (High, HARM: 320) at: http://localhost/invoicing/clients_view.php?

Message: Cross-site scripting vulnerability found
Injected item: POST: SelectedID
Injection value: "><script>alert(12943163.6277)</script>
Detection value: 12943163.6277
This is a reflected XSS vulnerability, detected in an alert that was an immediate response to the injection.

Cross-Site Scripting - [OWASP 2010 A 2]

4. Vulnerable (High, HARM: 320) at: http://localhost/invoicing/clients_view.php?

Message: Cross-site scripting vulnerability found
Injected item: POST: SortField
Injection value: >'><script>alert(12943163.7527)</script>
Detection value: 12943163.7527
This is a reflected XSS vulnerability, detected in an alert that was an immediate response to the injection.

Cross-Site Scripting - [OWASP 2010 A 2]

5. Vulnerable (High, HARM: 320) at: http://localhost/invoicing/clients_view.php?

Message: Cross-site scripting vulnerability found
Injected item: POST: SortDirection
Injection value: >'><script>alert(12943163.7597)</script>
Detection value: 12943163.7597
This is a reflected XSS vulnerability, detected in an alert that was an immediate response to the injection.

Cross-Site Scripting - [OWASP 2010 A 2]

6. Vulnerable (High, HARM: 320) at: http://localhost/invoicing/clients_view.php?

Message: Cross-site scripting vulnerability found
Injected item: POST: SelectedID
Injection value: "><script>alert(12943163.7637)</script>
Detection value: 12943163.7637
This is a reflected XSS vulnerability, detected in an alert that was an immediate response to the injection.

Figure 10: Hailstorm's Cross-Site Scripting pen test results

www.manaraa.com

6.3 SQL Injection

In this section I will present the results of the scans performed by CodeSecure and Hailstorm after scanning for SQL Injections in the Simple Invoicing web application.

Results from CodeSecure

After scanning Simple Invoicing for SQL Injection vulnerabilities, CodeSecure reported a total of 115 'Resulting Vulnerabilities.' At first, when looking at the 'vulnerability specification' part of the executive summary of the scan (figure 13 on page 34), I thought that there was only one statement in the entire source code that contained all 115 of these vulnerabilities. Later it became clear to me that the term 'statement' as chosen by CodeSecure is not a statement from the source code, but a string that can be injected via these 115 vulnerabilities. CodeSecure also summarizes its findings in a list of 'Identified Entry Points.' According to this list the most vulnerable files are the most visited pages of the web application, i.e. `clients_view.php`, `invoice_items_view.php` and `invoices_view.php`, all three of them containing eighteen vulnerabilities.

I believe most of the possible flaws CodeSecure reports are actual vulnerabilities, i.e. true positives. I came to this conclusion because I have not been able to find any sanitation functions anywhere in the entire source code. This has made CodeSecure believe all of the SQL query invoking functions are flawed. However, due to the fact that all of these SQL queries are being invoked by pressing a row in a table and not via text-based user input, a user will probably not be able to exploit them. It is thus unclear whether the possible flaws CodeSecure reports are true or false positives. As I said before, I believe these are true positives, because it might not be possible to exploit them right now, but possibly a future update of Simple Invoicing will add functionality for manually performing SQL queries, thus leading to an exploitable SQL Injection vulnerability.

Results from Hailstorm

Hailstorm didn't find any of the SQL Injection flaws CodeSecure reported. At first I thought this was due to Hailstorm not scanning the page at all, but when I looked up the pages Hailstorm had visited I noticed this was not

www.manaraa.com

the case. Then it raised the question whether CodeSecure reports a false positive in this case. This also doesn't seem likely as I have not been able to find any input sanitation functions in the code. My conclusion is that Hailstorm wasn't able to input a query to be executed by Simple Invoicing. As I said in the section above about the results from CodeSecure, these SQL queries are invoked by pressing a record in a table, instead of entering strings like

$$x' \texttt{ OR } 1 = 1; --$$

into search fields. Since all this functionality is automated there is no possibility of exploiting it, at least not by using only SQL Injection attacks. Maybe after using a few other attacks it would be possible to alter this behavior, but that is beyond the scope of this thesis.

www.manaraa.com

Executive Summary

Scan Summary

Project	Invoicing
Project Manager	admin, GeertS
Scheduled	Dec 15, 2010 19:41:57 Europe/Amsterdam
Started	Dec 15, 2010 19:41:58 Europe/Amsterdam
Finished	Dec 15, 2010 19:42:24 Europe/Amsterdam
Duration	26 seconds, 714 milliseconds
Scanned Files	55
Scanned Lines	9,939
Vulnerable Entry Points	32
Resulting Vulnerabilities	115
Vulnerable Statements	1
Vulnerable Files	1

Scan Settings

Scan Type	Immediate Scan (requested by admin)
Policy Name	SQL Injection
Tainted Source Type(s)	Web Request , Property Configuration , XML , File , Network Input , Web Service , Database , Command Line Arguments , Private , Hard-coded , System Environment , Exception Handling , Session
Vulnerability Modules	Dataflow | SQL Injection (CWE 89)
Traceback Setting	One most vulnerable traceback
Merge Unknown	No

Vulnerability Density

Metrics	Vulnerability Density
Per File	2.091
Per Thousand Lines of Code	11.571

Vulnerability Specification

Vulnerability Type	Vulnerable Statement	Resulting Vulnerability
SQL Injection (CWE 89)	1	115

Vulnerability Distribution

Vulnerability Type Distribution

SQL Injection (CWE 89)

Category Risk Score

Low Moderate High

Risk Score

100 90 80 70 60 50 40 30 20 10 0

SQL Injection (CWE 89)

Figure 13: Executive Summary of CodeSecure's SQL Injection Scan

www.manaraa.com

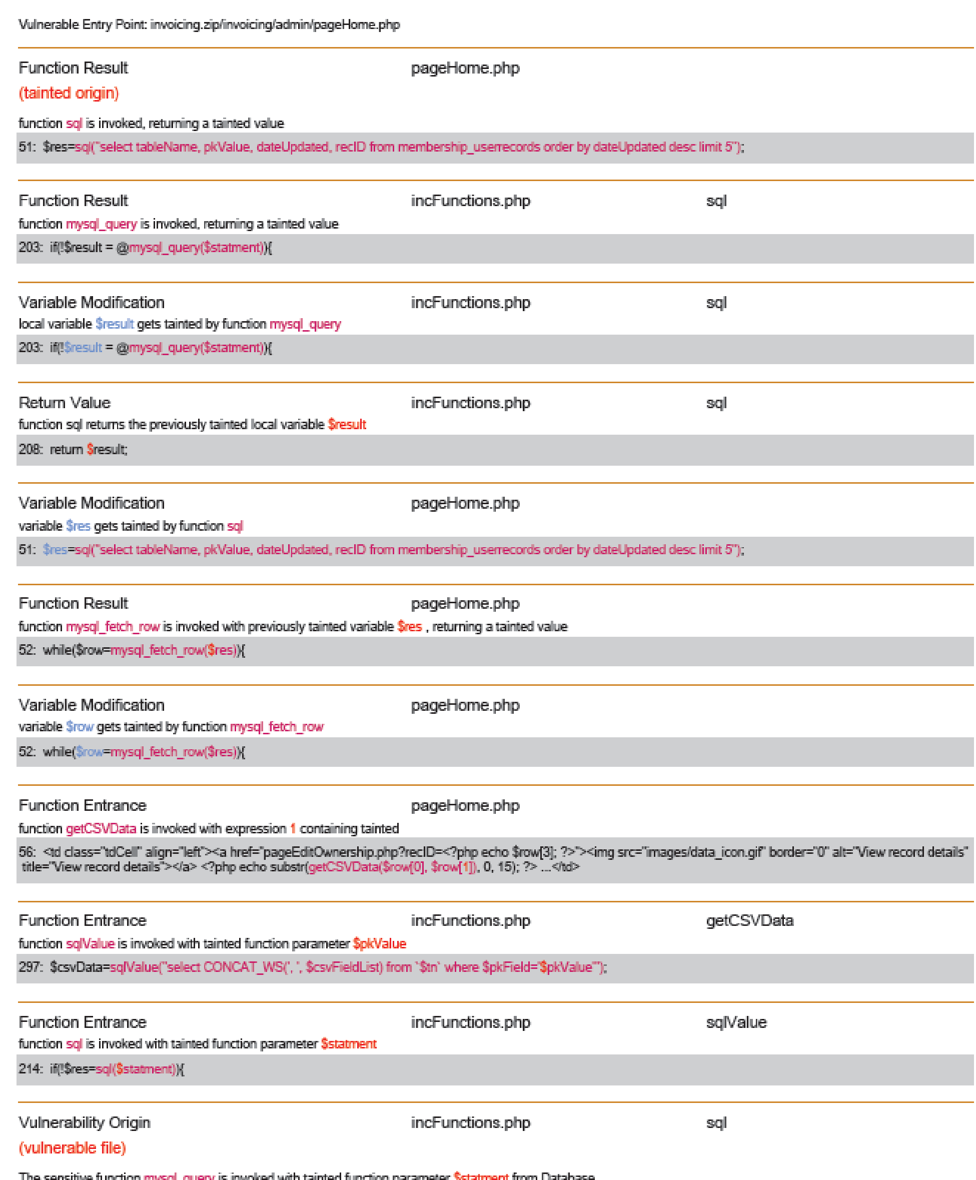


Figure 14: CodeSecure SQL Injection Trace

www.manaraa.com

6.4 Conclusions of experimenting

After scanning Simple Invoicing with Hailstorm it became clear that BigProf's claim of not having to worry about SQL Injection, Cross-Site Scripting and other flaws, was indeed false. As you can see in the results presented in the previous sections, Simple Invoicing really does contain lots of Cross-Site Scripting and SQL Injection vulnerabilities. After taking a close look at the reports both tools produce, I found out that they report roughly the same vulnerabilities. However I have found a couple occurrences where either one of the tools did not detect a flaw while the other did, e.g. when scanning for SQL Injection vulnerabilities. They also do not detect the same flaws. In figures 5 and 7 on pages 19 and 23 respectively, you can see CodeSecure detects 1059 Cross-Site Scripting vulnerabilities, whereas Hailstorm only finds 408. This seems due to the fact that CodeSecure can only guess what the effects would be, for reasons stated in chapter 5 on page 15, while Hailstorm can directly verify a potential flaw by observing the output. All in all, both tools have a big overlap in the vulnerabilities reported, but they also both report vulnerabilities that the other does not. Some reasons for this are described in chapter 5 on page 15.

I have also run an OWASP Scan with Hailstorm, to see how well Hailstorm does on finding vulnerabilities that aren't Cross-Site Scripting or SQL Injection. The results of these scans showed that Simple Invoicing is not only very leaky when it comes to Cross-Site Scripting and SQL Injection, but it also does not do such a good job at preventing multiple other types of attack scenarios. For instance, as we have seen in figure 7 on page 23, Hailstorm has also found hundreds of SQL Error Message occurrences. This allows for an attacker to guess field names from your database in order to construct specific (malicious) queries. Besides that there are, among others, also Cookie Vulnerabilities, Password Autocomplete and Cross-Frame Scripting vulnerabilities present in the application.

Hailstorm clearly is quite an all-round tool, i.e. it also detects a lot of other vulnerabilities. However, CodeSecure is not a bad tool at all either. As shown in CodeSecure's general scan results (figure 5 on page 19), it also detects a lot more than just these two vulnerabilities. The question here is if all those items are not for the most part false positives, but it looks like that is not the case. Only about half of the possible flaws seem to be false positives. CodeSecure really only has some trouble with guessing the resulting output after entering values into form fields, but that was to be

www.manaraa.com

expected. Sometimes it guesses correctly, while at other times the guess is wrong.

As I already mentioned in the section 6.3 on page 32, Hailstorm does not report any vulnerabilities of this particular type. In this case CodeSecure does a better job, but this is mainly because CodeSecure can only see what *would* happen if the input was ‘infected’ as they call it. Hailstorm can verify this directly, because it need not guess the output. This does not mean Hailstorm is faulty in not finding any of these vulnerabilities. In this case, I prefer to think of not being able to find it as it not being exploitable (yet). This may change in the future, when an update to Simple Invoicing is released changing the current SQL query behavior.

It seems like CodeSecure is somewhat more prone to false positives, due to the fact that it has to guess output instead of observing it. Hailstorm reports significantly less false positives, but also manages to miss the SQL Injections, although that is not entirely due to Hailstorm but more due to the injections not being exploitable (yet).

Regarding the difficulty of fixing detected flaws, in my opinion CodeSecure provides a user with much more feedback than Hailstorm does, at least when reviewing Cross-Site Scripting vulnerabilities. CodeSecure gives you a detailed trace of the vulnerability. Unfortunately I have not been able to get the IDE plugin for CodeSecure up and working, because somehow the connection kept getting refused. It would have been interesting to see to what extent it were possible to use the IDE to perform a scan, but more importantly, to get the results directly back into the IDE.

In contrast to CodeSecure’s reporting capabilities, Hailstorm merely provides you with its location and it will also give you some steps towards remediation. However these remediation steps are constrained to giving a few basic tips for each programming language.

All in all I have found a few situations where Hailstorm picked up on a vulnerability while CodeSecure did not, and also a few where the roles were reversed. Because the list is quite long considering the number of Cross-Site Scripting and SQL Injection vulnerabilities present in Simple Invoicing, I have not listed those, apart from the SQL Injection vulnerabilities. That is why it seems to be useful to be able to combine both CodeSecure and Hailstorm into one process. This way it would require less effort and time from software engineers to sufficiently test their products. It would also be less tedious for the engineers to go through all of CodeSecure’s false positives,

www.manaraa.com

since most of them would be filtered by overruling them with Hailstorm's results. If it were possible to combine them, an engineer would build a part of the web application and have it scanned with the press of a button. In the meantime he could go get himself a cup of coffee and when he returns the tests would be done, allowing him to analyze the results.

www.manaraa.com

7 Combining code scanning and automated pen testing

After experimenting with both tools it became clear that for a really good software development lifecycle it is necessary to integrate code scanning and automated pen testing into one process. The reason for this is that when using only automated pen testing tools like Hailstorm, you sometimes will not be able to find all of the vulnerabilities in a given web application, as we have seen in the experiments performed on Simple Invoicing. As described in section 6.3 on page 32, CodeSecure finds a lot of SQL Injection vulnerabilities, but Hailstorm doesn't find any of those, due to them not being exploitable (yet). This is a specific example of a reason why it is useful to combine both code scanning and (automated) pen testing. The same is also true for Cross-Site Scripting, as I described in sections 6.2 and 6.4 on pages 24 and 36 respectively, since a combination of both tools finds more vulnerabilities in Simple Invoicing than either tool does on its own.

So, as I had anticipated at the end of section 5.1 on page 15, it seems to be quite useful to combine both code scanning and (automated) pen testing when performing a code review of your web application. One way to do this is via the export option as can be seen on the left hand side in figure 5 on page 19. Unfortunately this doesn't work. In fact it will be removed upon the release of version 4 of CodeSecure. I spoke with a member of the CodeSecure support staff regarding this topic, and he told me that "it was never fully implemented within the Hailstorm product.". The intended use for this function was to be able to download an XML file that you can import into Cenzic's tools, including Hailstorm, in order to combine both tools' report generating functions. Strangely enough when you download this XML file, all you get is a blank file instead of the output you would have liked to insert into Hailstorm, confirming the theory of the member of the support team.

Since it was impossible to incorporate CodeSecure's results into Hailstorm's, a different approach is needed. Ideally an engineer would write a new function, upload it to the code scanner to see if it is securely programmed, and then continue his work. After a little more of the web application is complete it is then possible to perform a quick automated pen test to see if the code scanner was correct in detecting, or not detecting, a given flaw. This way both tools' *false negatives*, i.e. flaws that remain undetected, get scanned twice to increase the chance that at least one of the tools to find it. As we

www.manaraa.com

have seen after experimenting, it appeared that a lot of the vulnerabilities both tools report are in fact the same. This indicates an overlap in the tools. The vulnerabilities that go undetected by one tool but not by the other are the most interesting aspect. Recall the SQL Injection vulnerabilities that are found by CodeSecure, but not by Hailstorm. Obviously it is not possible to find each and every one of the flaws in a given web application. The program code usually is too complex for that and it requires a great deal of processing power to be able to exhaustively generate inputs.

Ideally you would probably like to use a programming IDE with a connection to CodeSecure via the plugin I mentioned near the end of section 3.3 on page 7 (or in the near future a locally installed tool), upload your files for a security review, and then have CodeSecure (or the IDE) initiate an automated pen test by means of Hailstorm after it completes the code scanning, sending its results along with the pen test request. Part of the fully automated solution is already there, i.e. the IDE with plugin, but for initiating scans with Hailstorm more effort is required. Cenzic does provide access to Hailstorm by means of a command line interface and an API. These could then be used for starting the scan. The downside is that I have been unable to find out where this API is described. The help function of Hailstorm doesn't provide the answer, and neither does their website. Since I am now somewhat familiar with Hailstorm, I had expected it not too be too big of a problem to start a scan using the command line interface, but then again the real trouble would be combining both tools' produced reports.

www.manaraa.com

8 Future Work

In this thesis I haven't been able to explore all the areas of interest I would have wanted to. Below I will detail each of these areas in a single paragraph.

As I mentioned in chapter 3 on page 6 for example, I spoke with a member of Armorize's support team who told me they have a release planned within the coming months for version 4.0 of CodeSecure, which will no longer be shipped in an appliance model. I do not know how much has changed since the version I used, since I cannot test it at this time.

Besides using a newer version of the tools I used to write this thesis, it would also be interesting to have a look at some of the other types of flaws that OWASP deems critical in their Top Tens, such as Cross-Site Request Forgery, Insufficient Transport Layer Protection, Insecure Cryptographic Storage etcetera. I have performed an automated pen test with the most vulnerable types of attacks as presented by OWASP, but I have only included a general overview in section 6.1 on page 19, as there are simply too many topics to discuss.

Sadly, when trying to export the results CodeSecure produced, I got presented with a blank XML file. Had it been a file that actually contained results it might have been possible to import those into Hailstorm. I have contacted Armorize about this possible bug and they told me the feature was getting removed in their upcoming version of CodeSecure. Please refer to chapter 7 on page 39 for more details. It would have been interesting to see what would have happened once I did manage to import some results into Hailstorm. I had hoped for a combined report featuring both CodeSecure's and Hailstorm's scan results, but since that will never be possible it might be better to combine both tools before they do any scanning.

Unfortunately, as I mentioned in chapter 7 as well, it was not yet possible for me to write a script that would be able to initiate a scan in CodeSecure followed by a pen test in Hailstorm. This is due to Hailstorm's API not being publicly available, and since my trial has expired I could not gain access to both Hailstorm and its API anymore. Future studies could further explore this topic, in order to find out if it is possible to combine both tools once access to the API has been obtained.

www.manaraa.com

9 Conclusions

During my research I found several interesting answers. I listed them in the coming sections. Some, if not most of them, I have already explained in section 6.4 on page 36, but for purposes of completeness I have decided to mention those here as well.

9.1 The project in general

Since I am a new employee at Forus-P my employer suggested I start learning the basics of Hailstorm. I could then learn how to use it and also it would give me a subject to write this thesis about. However, when I first started writing this thesis, I had not yet obtained a license to use Hailstorm. My employer then sent a request for a trial license to Cenzic, but I only received this fourteen day trial license in the first week of December. After trying out the various options I started performing tests on Simple Invoicing, but when I wanted to create vulnerability reports the next day, I was locked out of my trial version with no way to get back in, precisely during the Christmas holidays. During that time I found myself unable to perform scans as my contact at Cenzic was not at his desk very often during that period. Had I known my trial started at the time of the install instead of the time I inserted a license, I probably also would have started the experimenting sooner. Luckily just before New Year's Day I received another fourteen day trial license, so I could extract the results and include them in this thesis.

At the start of the project I thought it was more important to show my supervisors that I was at least producing text so they could see I was doing something. Now I see I should have started experimenting a couple of months earlier, as it appeared I was quite stressed for time in the end because I had to do all experimenting and refining of the text in just a few weeks time. Maybe that way I would have received the second trial before the Christmas holidays instead of during them, enabling me to do multiple reruns of the scans and compare them.

A difficulty related to Hailstorm's trial is the trouble I had getting a web server up and running with a J2EE application. Since I am not used to setting up a server I spent countless hours trying to get a web server up and running with a J2EE application installed on it so I could start doing automated pen tests with Hailstorm. The reason I kept trying to install this

www.manaraa.com

application was because at first I had chosen to scan OWASP's WebGoat web application and it was written in J2EE. In order not to get too mixed results, I wanted to perform the same scans on an application written in the same language running on the same server. After a while I then decided to give up on trying to install a J2EE web application and installed one written in PHP instead. This was a lot easier, as all server software needed came in one package, i.e. XAMPP.

I have also attempted to do a detailed analysis of OWASP's WebGoat with both of the tools. Later on I decided to drop this, deliberately insecure, web application, since OWASP has not implemented more than one of each vulnerability, making a detailed analysis of the application too simple. Both tools could then either find the flaw or not find it, and there was not much room for false positives either.

9.2 The tools used

CodeSecure was quite easy to get up and running compared to Hailstorm. I had almost no trouble understanding how to set up a scan with CodeSecure, whereas I had real difficulty understanding what to do with Hailstorm. Besides the ease of working with CodeSecure, my trial license lasted until I wanted it to, as the support personnel was very helpful and were willing to renew my license no questions asked. With Hailstorm I had some trouble understanding how to set up a scan at first. Sometimes the interface also is not all that clear, as there are a lot of windows, and the frames therein sometimes make it hard to find the information you seek. Once you are familiar with Hailstorm however, it becomes a very powerful tool to do deep pen tests of web applications.

Hailstorm clearly is quite an all-round tool, i.e. it also detects a lot of other vulnerabilities. However, CodeSecure is not a bad tool at all either. As shown in CodeSecure's general scan results (figure 5 on page 19), it also detects a lot more than just these two vulnerabilities. The question here is if all those items are not for the most part false positives, but it looks like that is not the case. Only about half of the reports seem to be false positives. CodeSecure really only has some trouble with guessing the resulting output after entering values into form fields, but that was to be expected. Sometimes it guesses correctly, while at other times the guess is wrong.

www.manaraa.com

As I already mentioned in section 6.3 on page 32, Hailstorm does not report any vulnerabilities of this particular type. In this case CodeSecure does a better job, but this is mainly because CodeSecure can only see what *would* happen if the input was 'infected' as they call it. Hailstorm can verify this directly, because it need not guess the output. However this does not mean Hailstorm is faulty in not finding any of these vulnerabilities. In this case, I prefer to think of not being able to find it as it not being exploitable (yet). This may change in the future, when an update to Simple Invoicing is released changing the current SQL query behavior.

It seems like CodeSecure is somewhat more prone to false positives, due to the fact that it has to guess output instead of observing it. Hailstorm reports significantly less false positives, but also manages to miss the SQL Injections, although that is not entirely due to Hailstorm but more due to the injections not being exploitable (yet).

Regarding the difficulty of fixing detected flaws, in my opinion CodeSecure provides a user with much more feedback than Hailstorm does, at least when reviewing Cross-Site Scripting vulnerabilities. CodeSecure gives you a detailed trace of the vulnerability. Unfortunately I have not been able to get the IDE plugin for CodeSecure up and working, because somehow the connection kept getting refused. It would have been interesting to see to what extent it were possible to use the IDE to perform a scan, but more importantly, to get the results directly back into the IDE.

In contrast to CodeSecure's reporting capabilities, Hailstorm merely provides you with its location and it will also give you some steps towards remediation. However these remediation steps are constrained to giving a few basic tips for each programming language.

Things I like about both tools is the fact they give you a breakdown of the flaws they encounter in the scanned applications. According to my employer these stats are primarily interesting for the management of the company hosting the web application, whereas the security staff generally want more information like traces of the problem. I myself find these stats handy for a quick look at the overall vulnerability of a given web application. Both tools also feature extensive report generating capabilities.

One feature I would have liked to make use of is the importing of CodeSecure's scanning results into Hailstorm. I have sent an email to CodeSecure's support regarding the topic of the blank XML files, but I have not received a reply on this yet. It sure could come in handy for scanning with Hail-

www.manaraa.com

storm, as CodeSecure has already reported places to look. Hailstorm would then just have to verify the possible vulnerability instead of looking for them. I hope, when or if it actually works, that Hailstorm then also keeps performing scans and does not solely rely on CodeSecure's reports to find vulnerabilities.

Currently the general software development life cycle for building web applications goes as follows:

1. Build the web application
2. Perform an automated penetration test using any automated penetration tester.
3. Review the results to make sure no severe vulnerabilities are found anymore.
4. Repeat the above steps.

This looks like an inefficient cycle. In my opinion, like I previously stated in chapter 7 on page 39, the best way to integrate security review into the software development life cycle is:

1. Start by programming a small function of the web application.
2. Upload this new function to CodeSecure by means of the plugin added to your IDE.
3. Review the results from CodeSecure to make sure no severe vulnerabilities are found in the function you have written.
4. Repeat the above until a large enough portion of the web application is done to do a pen test.
5. Perform an automated penetration test using Hailstorm.
6. Review the results from Hailstorm to make sure no severe vulnerabilities are found in the finished portion of the web application.
7. Repeat the above until your web application is finished and no severe vulnerabilities are found anymore.

Unfortunately I haven't been able to automate this process due to the API not being publicly accessible, as previously discussed in chapter 7 on page 39.

www.manaraa.com

References

[1] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *Security & Privacy, IEEE*, 3(1):84–87, 2005.

[2] Dejan Baca. *Automated static code analysis [Elektronisk resurs] : A tool for early vulnerability detection*. Blekinge Institute of Technology School of Engineering - Dept. of Systems and Software Engineering., 2009.

[3] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

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

[5] J. Conallen. Modeling Web application architectures with UML. *Communications of the ACM*, 42(10):70, 1999.

[6] N.L. de Poel, F.B. Brokken, and G.R.R. de Lavalette. Automated Security Review of PHP Web Applications with Static Code Analysis. Master's thesis, State University of Groningen, 2010.

[7] J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 365–372. IEEE, 2008.

[8] M. Howard, D. LeBlanc, and J. Viega. *19 deadly sins of software security*. McGraw-Hill/Osborne, 2005.

[9] Y.W. Huang, C.H. Tsai, T.P. Lin, S.K. Huang, DT Lee, and S.Y. Kuo. A testing framework for Web application security assessment. *Computer Networks*, 48(5):739–761, 2005.

[10] S.C. Johnson and inc Bell Telephone Laboratories. *Lint, a C program checker*. Citeseer, 1977.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–263. IEEE, 2006.

www.manaraa.com

[12] Security master students of the Radboud University Nijmegen. https://lab.cs.ru.nl/algemeen/Software_Security/_CodeSecure.

[13] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17. Citeseer, 2000.

[14] J. Williams and D. Wichers. OWASP Top 10 - 2010. http://www.owasp.org/index.php/Main_Page, 2010.

www.manaraa.com