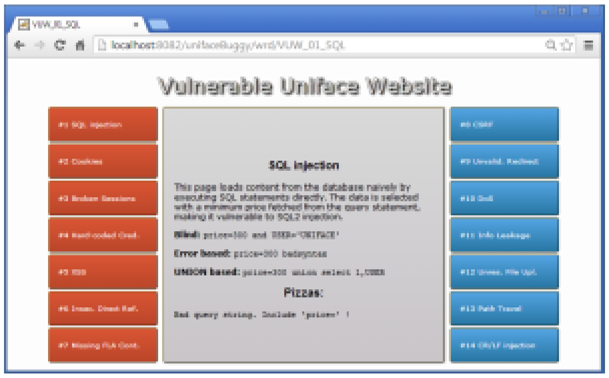
Automated Security Analysis for Uniface Web Applications

(Original creator: chillebrink)

Guest contributor, Job Jonkergouw, Uniface Intern Last February I started my internship at the Uniface. In need of a research project for my Master's in Software Engineering, I tried my luck at the Uniface headquarters in Amsterdam which offered a subject that was both challenging and socially relevant: security of web applications. Security is a hot issue in today's IT landscape as news of stolen user databases and hacked websites regularly hit the headlines. Traditionally, developers react by implementing counter measures such as firewalls and SSL but according to experts this is not enough: "secure features do not equal security features" (see Howard & LeBlanc, *Writing Secure Code*). Software has to be written with security in the mind and hearts of the developers. In an attempt at ensuring code security, models like Microsoft's MS SDL and OWASP's SAM recommend various steps in development. These include security requirement specification, architecture review, threat modeling and other practices. Another important guideline is security code review. However, done by humans this can be tedious and requires a high level of expertise, which is why many developers opt for something quicker. Automated code review will be familiar to anyone who has used Word's spell checker or a sophisticated IDE such as Eclipse.

For the purpose of security analysis, automated tools can check each line of code for dangerous function calls, iffy comments or unchecked in and output. This is commonly known as *static security analysis,* contrasting with a technique called *dynamic security analysis:* emulating actual attacks on the web application. Also known as *pen testing,* it is commonly executed by sending HTTP requests containing dangerous payloads such as SQL injection or cross-site scripting. The objective of my research project was to gauge the difference between using dynamic and static security analysis for Uniface web applications. To test this empirically, I designed an experimental website that contained several exploitable vulnerabilities. Several tools — both dynamic and static — were then tested by their ability to find each of these exploits.

The first objective was to identify the security analysis tools that were to be used. Some of the popular brands such as IBM's *AppScan* and HP's *WebInspect* require thousands of dollars of licensing fees, making them impractical for my studies, while others don't support the technologies used by the Uniface framework. Another issue concerned how more and more commercial products are being offered as a *Software-as-a-Service* (SaaS) on the cloud. While this makes it easier for the vendor to manage their licenses, it can be detrimental for developers who would not like to upload their source code to a third party or to have a testable web application deployed live on the web. Although the previously mentioned scrapped many of the popular solutions from my list, there were still enough tools left to experiment with, most of the open source. Making the final cut were five static analysis tools – FindBugs, LAPSE+, VCG, Veracode and Yasca -– and five dynamic analysis tools – IronWASP, N-Stalker, Wapiti, w3af and ZAP. The test environment was developed quickly using the Uniface Development Framework. During this step, I injected several vulnerabilities by removing a few important lines of proc code and twisting the properties of some of the widgets. These included accessing other user pages by modifying the user ID in the URL and unrestricted file uploading. As these were mainly behavioral issues, these types of exploits were only detectable with dynamic analysis as no static tools can read proc code. Other modifications I made at the Java source code level on the web server. These included important sanitization checks that normally prevent dangerous attacks such SQL injection and cross-site scripting. Notably different is that Java code is well understood by many static analysis tools.



SQL Injection

This page loads content from the database naively by executing SQL statements directly. The data is selected with a minimum price fetched from the query statement, making it vulnerable to SQL2 injection.

Blind: `price=300 and USER='UNIFACE'`

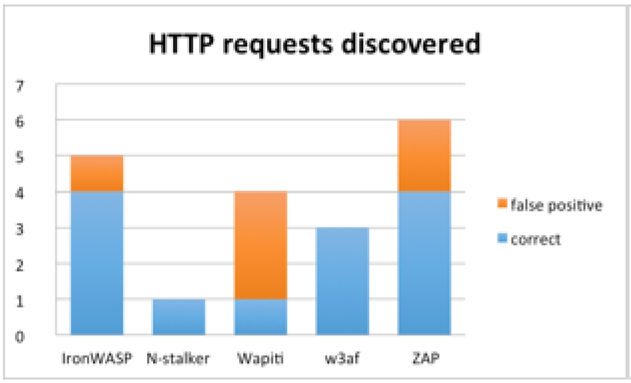
Error based: `price=300 badsyntax`

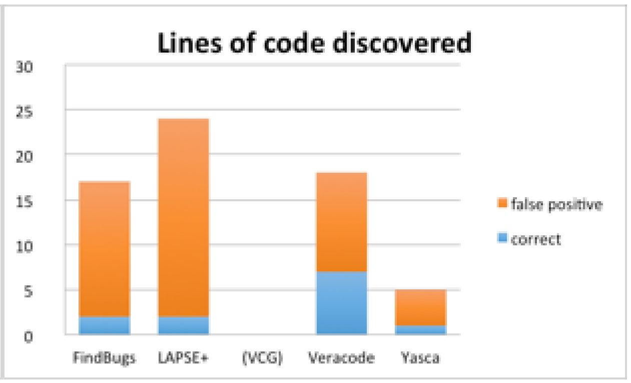
UNION based: `price=300 union select 1,USER`

Pizzas:

```
Bad query string. Include 'price=' !
```

The resulting website containing the vulnerabilities is shown above. Each tool was tested on its rate of discovery and the number of false positives. This latter number was much higher for most static tools, but was expected due to prior research and for theoretical reasons. The number of vulnerabilities tool found what varied widely as can be seen in the graphs below. Some vulnerabilities itself were hard to found altogether (such as path traversal requiring guessing of the right file name). But this was perhaps due to the nature of Uniface of being hard to scan, which makes it harder for actual attackers. A more detailed discussion on the results can be found in my final thesis: [link].





Despite the results containing few surprises, the internship offered me a great time at the Uniface development department, which proved to be both helpful and educational. In just a few months' time I was able to learn a new development language, build an application and carry out the work for my thesis thanks to the working environment and colleagues that helped me overcome any big hurdle. For this, my gratitude.