"Better Untaught Than Ill-Taught"

An Overview of Larry Suto's white paper:

"Analyzing the Effectiveness and Coverage of Web Application Security Scanners"

Ory Segal, Security Products Architect,
Rational, Application Security (Watchfire)
IBM

A Clear Overview

In October 2007, Larry Suto published the whitepaper "Analyzing the Effectiveness and Coverage of Web Application Security Scanners". The paper, attempted to quantify the effectiveness of automated web application scanners, and in particular he compared between NTObjective's NTOSpider, HP (SPIDynamics) WebInspect, and IBM Rational AppScan.

Larry's comparative work attempted to analyze the effectiveness of the scanners in four areas:

1. Links crawled by the scanners' web crawler
2. Coverage of application code (tested using Fortify's Tracer tool)
3. Number of verified vulnerabilities that were found
4. Number of false positives

The paper's conclusion showed a large discrepancy in the number of vulnerabilities detected by WebInspect and AppScan, to those detected by NTOSpider, and in addition it seemed as if WebInspect and AppScan missed a large amount of vulnerabilities (i.e. False Negatives).

My overview, which is presented in this paper, will attempt to shed some light on the results presented in Suto's paper, as well as to rebut its accuracy.

This paper will show that there are fundamental flaws in the original report:

1. The findings are not appropriate for comparison purposes, as they compare numbers from products that are measuring vulnerabilities at different levels of granularity (i.e. Apples and Oranges).
2. The methodology used in the original whitepaper is questionable, because it wasn't fully explained.
3. After requesting information about the test environment and AppScan scan files from the author, our own experiments produced significantly different results from those published in the whitepaper.

Variations on Trust

It's no secret that I work for IBM/Watchfire however this paper will present the entire test process I have conducted in the most transparent form possible, without any unknowns, so that you can reproduce the experiment and see the results for yourself.

I don't expect you to trust me, and I challenge each reader to perform his/her own experiment, using the detailed methodology I will present here. This whitepaper will not present raw numbers that are left to interpretation. Everything that is presented below can be reproduced and analyzed.

Apples and Oranges

The first concern I'd like to raise, is the problematic nature of comparing raw results from different products, which don't necessarily report on issues in the same way. In order to clear things up, I'll first explain how AppScan reports security issues.

The highest level of an issue that is found will be reported by AppScan as an "Issue" (or Issue Type). An issue is a distinct type of vulnerability (e.g. Cross Site Scripting, SQL Injection, HTTP response splitting, etc.).

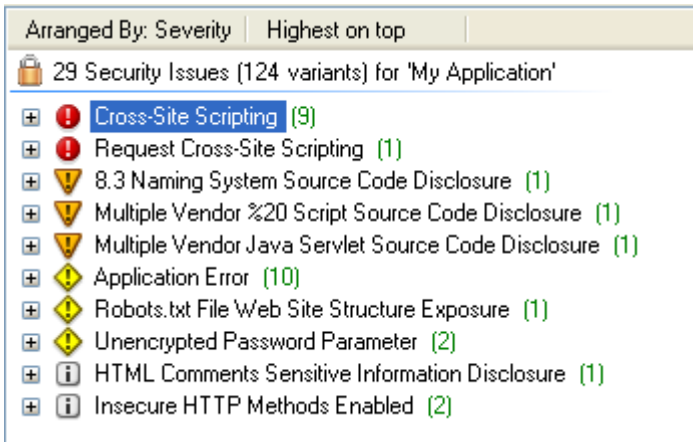

Figure 1: Highest level issue view (aggregated)

An issue can have many manifestations in different areas of the site, which brings us to the next level. When you drill down into an Issue that was reported, you are faced with the different URLs that are affected by that issue. For example:

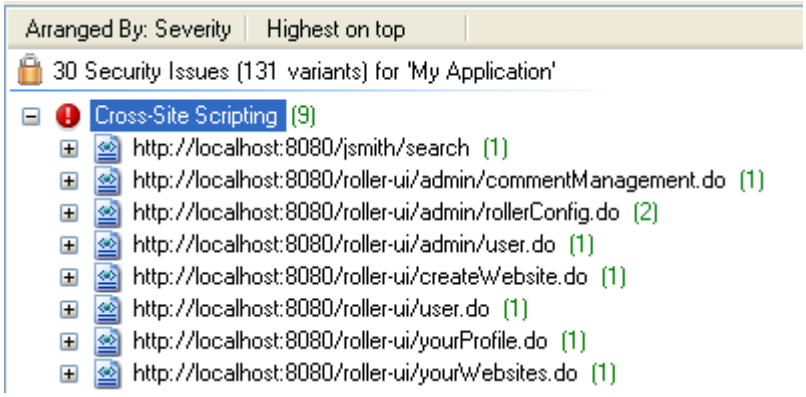

Figure 2: Expanded issue view (affected URLs)

Drilling down another level exposes what element of the request in each URL was found to contain the security issue, for example, what parameters or cookies where found to contain the vulnerability:

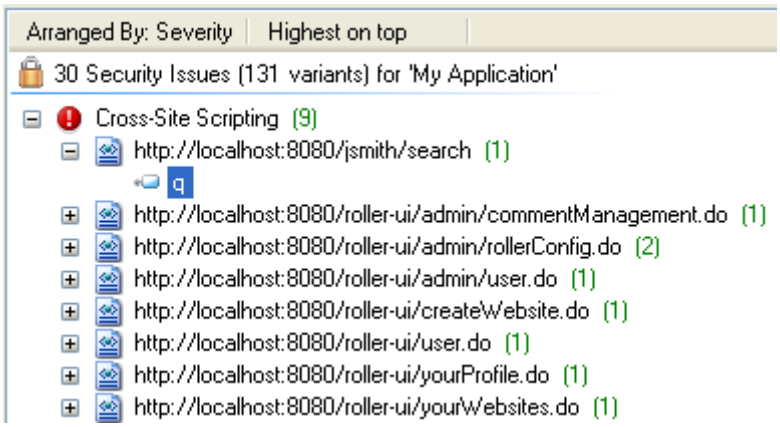

Figure 3: Vulnerable element issue view (affected parameter)

Since AppScan uses many different techniques (i.e. test variants) to perform certain application level attacks, the next level would be the different test variants that succeeded in defeating the application's security mechanisms. For example, for Cross-Site Scripting, the following variants might have succeeded:
<script>alert(..)</script>
<img src="javascript:alert(..);"></img>

Etc..

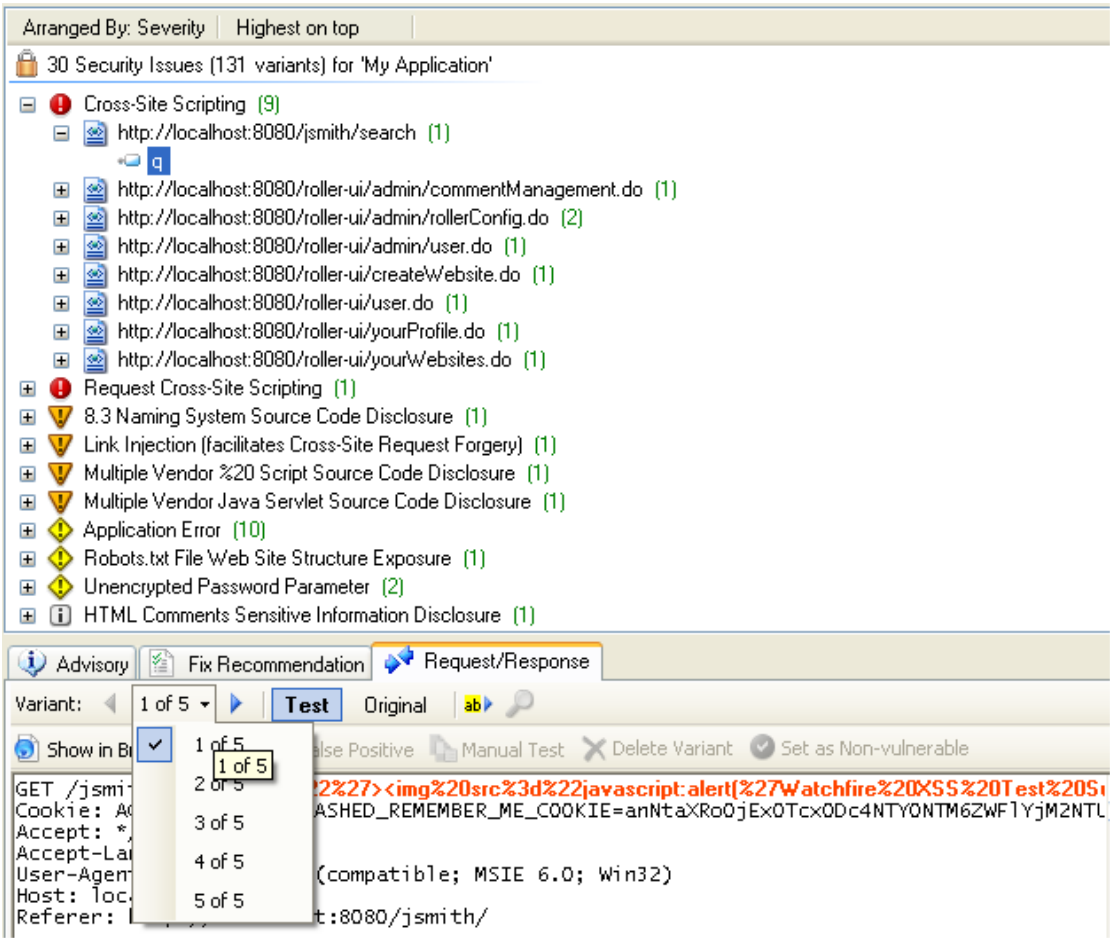

Figure 4: Successul test variants

As can be seen, AppScan's results view contains many different levels of information, each more comprehensive than its predecessor. This kind of view was created in order to simplify the tedious work of analyzing results, by aggregating and hiding unnecessary information from the user. In case the user is interested in more information, he/she can always drill down another level, until the required information is found.

In the example above, you can see that AppScan reported 9 Cross-Site Scripting vulnerabilities. This means that **9** different URLs are vulnerable to Cross-Site Scripting. But if you drill down to the vulnerable element level, and you summarize the amount of Cross-Site Scripting test variants that succeeded, you end up with **25** different successful test variants, i.e. the application contains 9 Cross-Site Scripting vulnerabilities that can be exploited in 25 different ways. But, since different test variants are still the same vulnerability, AppScan aggregates this information for the user, and reports only on 9 vulnerabilities (notice the difference in terminology – Issues vs. Test Variants).

So how is this related to the original experiment you ask? That's simple – WebInspect, NTOSpider and AppScan, each report their results differently. The term "vulnerability" or "issue" has different meanings in different products.

The table below shows the original OpenCMS experiment results:

OpenCMS - Open Source Customer Management application

Scanner	Links Crawled	Database API	Web API	Total API	Vuln Findings	False Positives
NTOSpider	3,380	47	158	205	225	0
AppScan	742	36	132	168	27	0
WebInspect	1,687	45	140	185	11	3

Figure 5: OpenCMS scan results – taken from the original whitepaper

Figure 5 shows that NTOSpider found 225 vulnerabilities, while AppScan found only 27, however, this table reports on Issues that AppScan found, and not on specific Variants.

After looking at how NTOSpider reports vulnerabilities, I have validated that NTOSpider lists and counts each "Test Variant" as a separate vulnerability. This means that the number of findings found by AppScan is significantly under-represented compared to NTOSpider due to how vulnerability counts are aggregated/grouped in AppScan but not in NTOSpider.

At our request, Suto was willing to provide me with the XML exported scan files from his review. The observant reader will notice that the following numbers are not exactly the same as the ones published in the original whitepaper. We assume that these results are from a different scan, but nonetheless we can still use them to clarify our point:

Visited Links: 129
Distinct Issue Types Detected: 19 different issues
Number of Issues: 56 (these are the vulnerable URLs)
Number of Variants: 377 successful test variants

When drilling down to the Cross-Site Scripting vulnerabilities, I found the following numbers:

Number of URLs affected by XSS: 4 different URLs
Number of successful XSS test variants: 27 different tests

You can clearly see how 4 issues could easily be counted as 27 different vulnerabilities, but in reality, there are only 4 issues here.

A Word about Efficiency and Coverage

It appears that one important conclusion is missing from the original whitepaper, and that is the efficiency of the crawlers when compared to one another.

Totals

Scanner	Links Crawled	Database API	Web API	Total API	Vuln Findings	False Positives
NTOSpider	4,207	69	314	383	227	0
AppScan	984	54	254	308	27	5
WebInspect	2,441	63	231	294	12	13

Figure 6: Original experiment results

A closer look at these numbers, reveals that AppScan crawled a total of 984 links, and managed to execute 308 APIs (31%), while NTOSpider crawled a total of 4207 links, and executed 383 APIs (9%).
This seems to conclude that AppScan is able to access more APIs (i.e. better coverage) with far fewer links, probably by skipping redundant URLs and saving precious scan time.

In addition, as will be seen below, when I have attempted to recreate the exact scans, I received a higher numbers of links for both applications.

Beauty is in the Eye of the Beholder

With the exception of one in-house developed web application on which we have no information about, the original experiment used two publicly available web applications – Roller, an open source Java-based blogging platform, and OpenCMS an open source Java-based content management system.
While the experiment document contained the above information, it never mentioned the content that was added to these applications. If we take Roller as an example – the original paper did not mention how many articles were posted in the blog, or how many comments were added to the existing articles. This information is critical, since every additional post or comment, can affect the number of URLs crawled by the scanner substantially, numbers that were a significant part of the experiment results and conclusions.

When I recreated the experiment, I have created a single user to the Roller application, with a single blog containing one sample post. According to information provided to me by Larry Suto, which exposed that his user had Administrator rights on the blog, I have provided my own user with the same credentials.

Details of the Experiment

In order to conduct an accurate experiment, which will be as close as possible to the one portrayed in the original whitepaper, I have contacted the author, asking for more information regarding the results and the environments in which the experiment was done. The author provided me with the following information:

1. Two files containing an XML Export of AppScan's runs on Roller and OpenCMS
2. Web application versions – Roller v3.1 with no CAPTCHA installed, and OpenCMS v7.0.1

After opening AppScan's XML Export files, I noticed the following things:

- AppScan v7.6.832 was used during the original experiment
- OpenCMS was installed on Windows XP Professional

I then proceeded with installing the exact web applications, which were mentioned in the original paper, in the following manner:

1. All web applications were installed on Windows XP Professional Service Pack 2 with the most recent Windows Update (as of November 25, 2007)
2. Installed Apache Tomcat 6.0.14 Server (for Windows)
3. Installed Java Platform Standard Edition 6, JRE 1.6.0_03
4. Installed MySQL Database Version 5.0.45
5. Installed OpenCMS v7.0.1 (details below)
6. Installed Roller v3.1 (details below)

The Roller installation was performed word by word according to the Roller Installation Guide (http://www.apache.org/dist/roller/roller-3/v3.1.0/docs/roller-install-guide.pdf), with the following five deviations:

1. I had to install Roller in a separate directory outside of the Apache Tomcat /webapps directory, since having it inside caused issues with Tomcat's autodeploy feature, which kept overriding my context.xml file that was created and placed in the %CATALINE_HOME%/conf/Catalina/localhost directory.
2. I did not enable email features in the Tomcat context file.
3. I created a blog administrator user called "jsmith", with password "demo1234", created a blog for that user with the URL http://localhost:8080/jsmith , added a single sample blog post, with a single comment.
4. I mapped the default site URL to the blog http://localhost:8080/jsmith (this was also the starting point for my scan)
5. In order to remove the CAPTCHA (math question comment authenticator), I performed the following changes to the roller.properties file:

   Commented out the following line (line wrapped):

   ```
   #comment.authenticator.classname=org.apache.roller.ui.ren
   dering.util.MathCommentAuthenticator
   ```

   Added the following line instead (line wrapped):

   ```
   comment.authenticator.classname=org.apache.roller.ui.rend
   ering.util.DefaultCommentAuthenticator
   ```

The OpenCMS installation was performed exactly according to the OpenCMS 7.0.1 installation guide. There were no deviations from the guide. I have also created a new user to the application called "jsmith" with password "demo1234". This user got administrator privileges over the application.

One additional crucial question that was never answered in the original document, is whether the applications and their databases were reset between each product scans. Assuming all products were given such credentials that allow site modification (e.g. posting a blog article, modifying site configuration settings, etc.), then with each scan, more and more links will exist in the application (both due to the automated form fillers doing their work, as well as the numerous tests that might create new content), and in some scenarios exception errors and other anomalies that will be covered later on in this document, might occur.

In order to narrow down the type of noise created by running multiple scans on the same application, I have decided to reset the application to its original state, and recreate the backend database before each scan.

On both applications, AppScan ran using out-of-the-box configuration, and the only two detail items I provided it with were the starting point URL and username/password for each application.

My View of the Results

This section will concentrate on analyzing the scan results. In order to be thorough, I will analyze results from 3 different sources –

1. Tables presented in the original whitepaper
2. Results extracted and analyzed from AppScan scan files, as received from the author
3. My own scan results, performed with the same version of AppScan, on my own environment, which should provide me with similar results to those found in the original whitepaper, thus allowing me to analyze vulnerabilities and False Positives.

Analysis of the Roller Application

Figure 7 below, shows the original whitepaper's Roller results:

Roller - Open Source Blogging platform

Scanner	Links Crawled	Database API	Web API	Total API	Vuln Findings	False Positives
NTOSpider	736	2	121	123	2	0
AppScan	129	1	98	99	0	5
WebInspect	663	1	68	69	1	10

Figure 7: Original experiment Roller results

In table 1 below you can view the results, as extracted from the scan performed in the original experiment that was sent to me (as an XML Export AppScan file)

	Results
Links	129
No. of Distinct Issues	19
No. of Vulnerable 'entities' (URLs/Parameters)	56
Test Variants	377
False Positives (measured by distinct Issue Types)	7 (36%)
False Positives (measured by Test Variants)	62 (16.4%)

Table 1: Roller results – taken from original experiment's XML output

Since the number of False Positives looked pretty high, I decided to look into them and analyze the reasons for their existence. However, the file I got from the author was an XML export file, which does not allow me to fully understand what went wrong during the scan, so I had to recreate the scan in my environment, and correlate the results. Using this technique, I managed to locate and recreate all of the False Positives from the original report:

The reported False Positives are:

- Blind SQL Injection
- Cross-Site Scripting
- Alternate Version of File Detected
- Application Test Script Detected
- Direct Access to Administration Pages
- Oracle Log File Information Disclosure

- Potential Order Information Found
- Potential Registration Information Found

Blind SQL Injection (4 test variants) - since AppScan was pointed both at the blog site, as well as the administration site, it seems that during the explore or test phases, AppScan's automatic form filler managed to mess around with some of the site's configuration settings. The mess in the configuration settings caused requests to various parts of the site to return a Java exception page. Since the Blind SQL Injection is automatically validated by comparing responses of several requests to each other, and to the original request/response, this caused the False Positive. The simple way to re-validate this issue and get rid of the False Positive is to revert the application configuration and database to their original/default state, and click "Re-Test" in AppScan. If all goes well, this vulnerability should turn into a "Not Vulnerable".

"Alternate Version of File Detected", "Application Test Script Detected", "Direct Access to Administration Pages", "Oracle Log File Information Disclosure", "Potential Order Information Found", "Potential Registration Information Found" (6 Issues - 56 test variants) seem to exist due to two different problems.

The first problem was that according to the HTTP traffic, AppScan was not logged in to the application at the time of the test. You can see this in the following HTTP traffic:

```
HTTP/1.1 302 Moved Temporarily
Content-Length: 0
Server: Apache-Coyote/1.1
Cookie: JSESSIONID=6FAE97FECF7F1A84AB2BF241A755FA0A
Location: http://xxxx.xxxx.xxxx.xxxx/roller/roller-ui/login.do
Date: Fri, 31 Aug 2007 01:47:28 GMT
Connection: close
```

Naturally, this is a redirect to the Login page of the application. At this point, AppScan requested the Login page, which results with the following response:

```
HTTP/1.1 200 OK
Content-Length: 6109
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=UTF-8
Content-Language: en-US
Date: Fri, 31 Aug 2007 01:47:28 GMT
Connection: close
```

…

The second problem has to do with the Customized Error Page Recognition and the way these tests are validated. As shown in the traffic above, when AppScan followed the 302 redirection, it eventually received a 200 OK response, leading it to believe that the vulnerable page exists.

As a side note, it seems that in my own experiment, AppScan managed to cope with the session management, and all of these requests got an actual HTTP 404, thus these vulnerabilities did not exist in my own scan.

Cross-Site Scripting: the reported vulnerable URL is /roller/[blog-name]/entry/[entry-name], and the reported vulnerable parameters are "name" and "email". The nature of this False Positive lies in the way AppScan v7.6 sent its XSS payloads – all containing the same string alert('Watchfire XSS Test Successful').

It appears that one of the earlier Cross-Site Scripting attacks was successful as well as persistent, causing these 2 variants to match as well (AppScan located the JavaScript in the

response, but the JavaScript payload belonged to an earlier persistent attack). This was fixed in AppScan v7.7, which from now on, submits arbitrary test strings, and correlates the matched string with each test variant.

To sum things up, most of the aforementioned False Positives are handled gracefully in newer versions of AppScan, with the exception of the Blind SQL Injection, which was caused by the automatic form filler running on the administration part of the blog application and messing things up. This issue could easily be resolved by limiting the automatic crawler from accessing sensitive blog configuration pages.

In order to complete my analysis, I will present the results and numbers that I got when running AppScan v7.6.883 SP1, on the same version of Roller (3.1).

The main difference between the scans is the amount of crawled links, which in my scan was 283 (visited links), that's 154 additional links.

In addition, after scrubbing the several SQL Injection False Positives that were caused from the same reasons as mentioned above, I concluded with the following numbers:

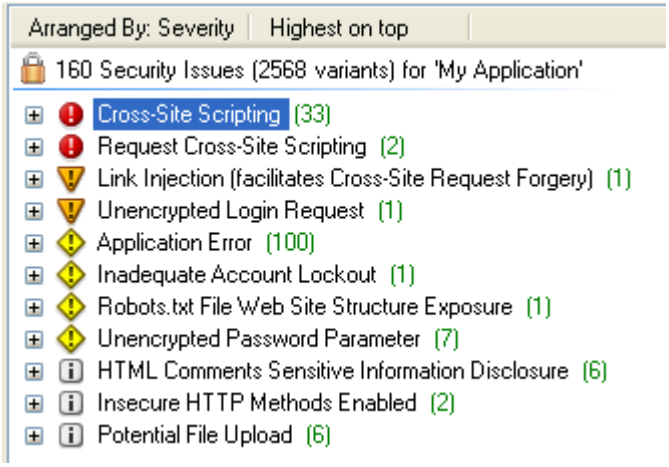

Figure 8: Roller results – as performed in the lab

That's 11 distinct Issues, spanning across 160 URLs, with 2568 successful test variants.

Here are the statistics, before/after scrubbing the SQL Injection False Positives:

	AppScan (w/ SQLi FP)	AppScan (w/o SQLi FP)
Links	283	283
No. of Distinct Issues	14	11
No. of Vulnerable 'entities' (URLs/Parameters)	189	160
Test Variants	2603	2568
False Positives (measured by distinct Issue Types)	3 (21%)	0 (0%)[1]
False Positives (measured by Variants)	**35 (1%)**	**2 (0.07%)**

Table 2: Roller results – as performed in lab, with and without SQLi FP

[1] The only False Positives remaining in the scan are the 2 "persistent" Cross-Site Scripting variants, but since XSS is an actual issue, there are no False Positive issues, but rather only 2 XSS False Positive variants

- The SQL Injection False Positive scrubbing was done by resetting the application and its database back to the original state and simply clicking "Re-Test" on the SQL Injection vulnerabilities.

Analysis of the OpenCMS Application

Figure 9 presents the results from the original whitepaper:

OpenCMS - Open Source Customer Management application

Scanner	Links Crawled	Database API	Web API	Total API	Vuln Findings	False Positives
NTOSpider	3,380	47	158	205	225	0
AppScan	742	36	132	168	27	0
WebInspect	1,687	45	140	185	11	3

Figure 9: OpenCMS results – taken from the original experiment

Table 3 below presents the results, as extracted from the scan performed in the original experiment that was sent to me (as an XML Export AppScan file)

	AppScan
Links	716
No. of Distinct Issues	17
No. of Vulnerable 'entities' (URLs/Parameters)	338
Test Variants	1071
False Positives (measured by Issues)	N/A
False Positives (measured by Variants)	N/A

Table 3: OpenCMS results - extracted from original experiment's XML export file

Again, it seems that the XML Export I got from the author is not from the exact same scan that was used to generate the whitepaper, and the discrepancy in numbers is quite large. This forces me to use my own scan results and validate each and every one of them.

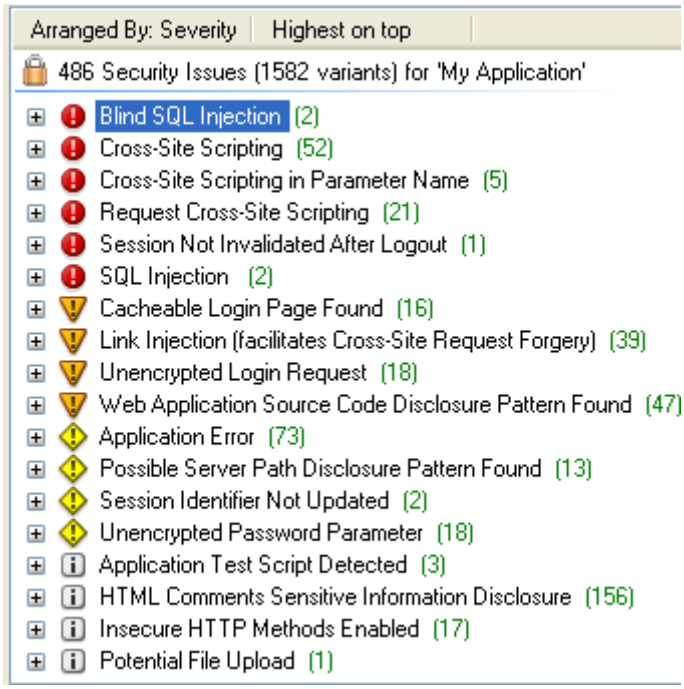

Figure 10: OpenCMS results - as performed in the lab

	Results
Links	2787
No. of Distinct Issues	18
No. of Vulnerable 'entities' (URLs/Parameters)	486
Test Variants	1582
False Positives (measured by Issue Types)	4 issue types (22%)
False Positives (measured by Test Variants)	65 (4%)

Table 4: OpenCMS lab results analysis

The Reported False Positives are:

- Blind SQL Injection False Positive occurred in the /opencms/opencms/alkacon-documentation/examples_search/result.jsp (Search Page Results), due to some shortcoming in the Blind SQL Injection validation algorithm, which attempts to detect similarities in the response pages. Since the HTTP response for the Blind SQL Injection page was quite small (229 Bytes long), minor changes in the pages, created large variance in similarity, which triggered the algorithm to believe that the pages are different, while in reality, they were quite similar.

- 8 Cross-Site Scripting variants were triggered due to a persistent Cross-Site Scripting vulnerability that triggered these tests to believe that they were successful. As mentioned in the "Roller" analysis, these issues were fixed in AppScan v7.7, which submits arbitrary Cross-Site Scripting payload, and correlates each request with the payload in the response page.

- 8 Path Disclosure False Positives were caused by detecting absolute paths inside documentation pages

- 47 Source Code Disclosure False Positives were caused by detecting what looked like application source code, which eventually turned to be a part of the documentation.

In addition, to the False Positives mentioned above, there were a few redundant test variants. All redundancies were caused by the fact that the application's login page also exists in 16 different release notes and documentation pages. This triggered issues such as "Cacheable Login Page", "Unencrypted Login Request", "Link Injection" and "Unencrypted Password Parameter", all of which are vulnerable, but each of them should be counted as a single vulnerability.

Criticism, Words of Admonition and a Cliff Hanger

Naturally, since I work for one of the product vendors that got criticized, it could be expected that I will complain about the unfavorable results presented in the original paper, however this is not why I chose to write this paper. Like my colleague, Jeff Forristal from HP/SPIDynamics (http://portal.spidynamics.com/blogs/spilabs/attachment/71302.ashx), I have decided to take action, and prove that Suto's experiment was questionable and misleading. My protest has little to do with the experiment's outcome, but rather with the questionable test results that contain multiple discrepancies.

In addition, I am concerned by the web application security industry - an industry filled with gifted security experts and practitioners, who embraced Suto's whitepaper warmly, without questioning its results or the methodology by which it was conducted for a single moment.

Suto, having good intentions published what he thought was in the best interest of the industry, and my biggest complaint to him was that his experiment methodology was never fully disclosed to the public, therefore could never be confirmed nor rebutted.

On the other hand, one would expect security experts to use a little more judgment when reading technical whitepapers, and be skeptical of results from experiments that are not well documented. Putting numbers into a table doesn't make them meaningful.

The bottom line of this is that any meaningful and productive discussion about the differences between web application scanners has to begin with a clearly defined test scenario that can be easily reproduced by others as well as a unified "results terminology", which will allow the tester to compare results from products, which report vulnerabilities in different ways. Without these prerequisites, the entire discussion is futile.

If you would like to receive more information or to send me feedback about this whitepaper, send an email to segalory [at] il.ibm.com