

Web Application Vulnerability Scanners - a Benchmark

Andreas Wiegenstein, Frederik Weidemann, Dr. Markus Schumacher, Sebastian Schinzel
Version 1.0 - 2006-10-04

Overview

Watching the history of security defects in applications for the last decades, it seems that all software has hidden and unexpected security defects – a really critical issue, especially for Web applications

One possible way to deal with such nasty defects is to use so called *Web application vulnerability scanners*.

The idea behind these scanners is to conduct security checks automatically and to produce a report describing the bugs in a application. Many companies rely on this approach. This whitepaper focuses on *black box* vulnerability scanners for Web applications and their capability to find application security defects.

Out of scope are scanners that analyze the underlying OS, Web servers or databases for specific, *known* vulnerabilities in order to determine if they have been patched correctly, as well as code analysis tools.

We wanted to see how efficient a scanner is in finding typical types of vulnerabilities in applications, using their detection algorithm instead of a database with known vulnerabilities of specific products.

If you ever asked yourself: "How secure is my application after I used a *black box* scanner and fixed all the bugs that have been reported?", this is the article of choice for you.

Target audience

Everybody using or planning to use *black box* application scanners, in particular:

- Security Testers
- CERT Teams
- IT Management

General Note

This whitepaper does not yet cover all scanners on the market. Therefore we may update it in the future.

If you have any comments on this whitepaper or wish to be notified about new versions, please contact us via wavs-whitepaper@virtualforge.de



Contents

Introduction	3
Chapter I: Preparation	4
Objectivity.....	4
Background	4
Scope	5
Mechanics of a scanner	5
Step 1: Spidering	5
Step 2: Initial analysis.....	6
Step 3: Input fuzzing.....	6
Chapter II: Setup	7
Spidering and form submission.....	7
Technical test cases	7
Business logic test cases	9
What was impossible from the start.....	10
Quest for Scanners	10
Chapter III: Benchmark.....	11
Spidering and form completion	11
Technical test cases	12
Business logic test cases	12
Overall ratings	13
Reporting.....	13
Chapter IV: Conclusion.....	14
Scanner efficiency.....	14
Where to use black box scanners.....	15
When to consult human security experts.....	15
Some TCO considerations.....	16
...and what about white box scanners?.....	16
Final word.....	16
References / Further Information.....	17

Introduction

In today's common security practice, companies use vulnerability scanners to assess the security of their applications. Some companies outsource this process and hire "security testers" that conduct the scanner tests for them.

Mostly, companies do this for one reason: scanners can actually find various security vulnerabilities in a short time frame with limited resources. This is very convenient, since tangible results can be presented to management, seemingly justifying the investment.

However, the important question to ask here is not "What *did* the scanner find?", but "How many bugs are still there that the scanner *did not* detect?".

There is no easy answer, though. Even if you knew all the remaining bugs in one application, this would still be insufficient, since the efficiency of a scanner is dependent on the programming language, type of application and web framework under analysis. A scanner may be effective for one technology or framework, but completely ineffective for another.

It's a big difference to test e.g. CGI apps written in C and JSPs written in Java. It's an even bigger difference to test a small app for newsletter subscriptions based on Apache or a CRM business process based on SAP®'s Web Dynpro concept.

Another important factor (but not discussed in detail in this whitepaper) are questions regarding the total cost of ownership (TCO) for using a scanner such as the cost for the tool itself, cost for personnel handling the tool, cost for training, cost for fine-tuning, maintenance and support, cost for testing of *false positives*, etc.

In this whitepaper we focus on the technical capabilities of scanners regarding the testing of vulnerabilities in (custom) web applications.

The following chapters describe the steps involved in designing, building and operating the benchmark system that is required to achieve a meaningful scanner evaluation.

The final chapter (*Conclusion*) contains a high-level summary of the benchmark results.

Chapter I: Preparation

In this chapter we describe our motivation to perform a scanner benchmark as well as general considerations regarding requirements for such a benchmark.

Objectivity

All authors of this whitepaper work for Virtual Forge, a security testing company. This immediately triggers the neutrality question: "A company that offers application security tests talks about the pros and cons of scanners (read: *their competitors*)? I can hear you."

Fortunately for us, we are both, security testers as well as security scanner developers, which makes us kind of neutral. And no, our scanner is not included in this test, because it works on a semi-automatic level only and was designed to assist testers rather than to replace them.

Background

From our early days as security testers we believed that it should be possible to build a tool that finds almost all bugs in a Web application automatically in order to make our day to day work more comfortable. We spent a lot of effort in writing such a tool (a *black box* scanner) and finally, after about three years developing and using this scanner in parallel to our manual testing, we came to the conclusion that black box scanners are only a rather small brick in the wall of application security audits. Don't get us wrong, the scanner works fine and is provably able to find certain bugs quite efficiently. However, we just know exactly where its limitations are and therefore use it for specific tasks only. It takes far more than a tool to find all the bugs in an application. In several cases we did a code review in parallel to a scanner sweep and we always ended up with far more findings in the manual review. In a way we knew for quite a while that scanners are not the complete answer to make an application secure. Actually this conclusion was more a gut feeling that came up when the pile of the *How-is-any-program-ever-able-to-find-a-bug-like-that?* vulnerabilities we found kept growing and growing over the years.

By the end of 2005 our customers began asking questions like "What is more efficient - a security tester or a scanner?" and "How secure are we when we use a scanner?" more often.

And we didn't want to give an answer based on our gut feeling.

We felt the need for a systematic approach, a straightforward analysis, something everyone could verify. Fortunately, we could rely on our long-year manual testing expertise. We have identified and analyzed several thousand security defects in business applications. Besides, our efforts to write our own scanner gave us a very good starting point for understanding what kind of test framework would be needed for this study.

Finally, in January 2006 we decided to conduct a benchmark test. It took about half a year to design and build the required benchmarking system, but finally we could start our tests in August and are happy to present the results now.

Scope

The candidates for our benchmark are *black box* scanners that analyze applications by sending requests and analyzing the responses. They have no understanding of the internal (business) logic of an application, its source code or how it connects to backend systems.

When we are talking about the efficiency of *black box* scanners, we refer to their capability to find as many vulnerabilities in (custom) software as possible. Basically, that means finding a certain type of problem (e.g. Cross Site Scripting) in an application with analytical methods. At the same time we expect to see only a minimum number of *false positives* in the report. Since every *false positive* will cost some time to detect that the reported vulnerability was actually not an issue, it can be a severe waste of time if there are too many false positives. Of course, this increases the operative costs of using a scanner (TCO).

Mechanics of a scanner

In order to build a benchmark system that yields meaningful results, we have to understand how scanners work. How does a scanner analyze an application without any internal information? Actually this is a three-step process:

1. The scanner processes the URL of a starting page for the Web application and tries to find all pages that are part of that application. This process is called spidering.
2. The completed spidering process leads to a list of pages that are going to be analyzed. The scanner tries to identify the input vectors of the pages such as forms, request parameters and cookies and searches for various suspicious patterns in the page content that are stored in its database.
3. Finally, every input vector of every page is “bombarded” with a variety of attack patterns - often referred to as *input fuzzing* - and the resulting pages are scanned for indications of a vulnerability. In other words: scanners send all the attack patterns in their database against every input parameter in every identified page and analyze the response from the server.

Let's take a closer look:

Step 1: Spidering

Finding all pages of a Web application is more difficult than usually expected. When accessing a given page, the scanner has to identify all links to subsequent pages. The simple task is to extract the (static) hyperlinks in a recursive process until no more new links are identified. What if the workflow of the application is determined dynamically depending on specific user actions that are evaluated by scripts on the client-side during runtime? The scanner would have to simulate all possible script executions in the page and compute all resulting links. Consider that those links could be inconspicuous strings concatenated at runtime by a scripting language on the client.

Another complex problem is form input. Every business application consists of dozens (if not hundreds) of forms that (when filled in correctly) lead to other pages and even more forms. When filled in incorrectly, however, the form will end up in an error condition rather than the intended next page. Unfortunately scanners don't understand the meaning of the data that has to be filled in, which makes this part of the spidering difficult without human guidance.

However, any human interaction increases the testing time and the operative costs. For complex business applications this means that a considerable amount of human interaction would be necessary, if at all possible. Remember: you have to fill in all pages for all possible work flow routes which means you can end up in the same form(s) many times, potentially resulting in thousands of forms to complete.

The spidering phase is the most important one, since it is the starting point for all subsequent security tests. Obviously a scanner can't find a vulnerability in a page if it can't find the vulnerable page before. It is therefore imperative to separate spidering tests from vulnerability tests in order to achieve a meaningful result.

Other vulnerability test systems (e.g. WebGoat) were not designed for scanner benchmarking, which is why they don't (have to) make this distinction.

Step 2: Initial analysis

The "spidered" pages are now analyzed for obvious problems such as *information disclosure* in comments or password fields that don't mask their input. Also (more importantly) all input vectors of every given page are enumerated. Input vectors in this context mean obvious input such as fields in a form or parameters in a URL, but also less visible input like information stored in cookies or other parts of HTTP headers like the *user agent* or the referring page. With this list of input vectors, the scanner starts the testing phase.

Step 3: Input fuzzing

The scanner sends potential attack patterns from its database to any input vector that has been identified in the previous step. Then it analyzes the response from the server for suspicious patterns that indicate a vulnerability. And this is the second important quality of a *black box* scanner: defining the appropriate patterns to find the vulnerability. First, there must be patterns for the different types and variations of vulnerabilities. Second, vendors have to get the balance right between patterns that are too specific and patterns that are too generic. While the former result in *false negatives* (vulnerabilities that stay below the radar) the latter will result in *false positives* (vulnerabilities that are actually no problem at all).

From these three steps, there are two key aspects to keep in mind for the design and implementation of a benchmark system:

1. Spidering is the most important quality of any scanner.
2. Failure to find a given vulnerability can originate from bad spidering, missing test patterns or insufficient analysis of server responses. All three aspects must be considered in the benchmark.

Chapter II: Setup

This section describes the various test cases that we have built for our benchmark platform as well as general problem areas that are hard (or impossible) to detect by a scanner. Furthermore we provide some information how many and what kind of scanners have been included in the benchmark.

Spidering and form submission

As described in the section *Mechanics of a scanner* in the previous chapter, it is vital for a benchmark to analyze how efficient a scanner is in identifying links and filling in forms.

Thus we have built more than 30 test cases analyzing this efficiency aspect. All of those test cases monitor access to the prepared pages and are able to log successful attempts, i.e. whether follow-up pages are reached or not. That way we could see in the logs of the benchmark platform which of our test cases were only reached and which were actually resolved by a scanner candidate.

Following are some examples of the test cases:

- Does a scanner understand redirects via the HTTP 'meta refresh' tag?
- Can a scanner interpret script code (from an external location) that computes the follow-up page dynamically?
- What about links or page names that are commented out but still exist?
- Are all required form fields filled in?
- Does a scanner understand that it has to fill in e.g. an e-mail address into a certain form field?
- Are length restrictions of input fields taken into account?

Please note that for all of the following test cases we used only the simplest methods of navigational access: static hyperlinks and simple form POST without any input validation. This was to make sure that the scanners reached all parts of those test cases. Thus, if any test case could not be resolved, we could conclude that the scanner's patterns or analysis methods are insufficient, not its spidering engine. This is an important difference for the benchmark.

Technical test cases

Technical test cases comprise vulnerabilities that have a technical root cause, in contrast to (business) logical problems or architectural misconceptions.

Typical problems in this area are:

Buffer overflow

Vulnerabilities related to the usage of insecure functions in connection with insufficiently allocated memory in unmanaged programming language environments.

Code injection

Remote execution of code an attacker manages to embed in input passed to the application.

Cross Site Scripting (XSS)

Through XSS, an attacker can manipulate Web pages other users will render in their browser and this way attack them.

Directory traversal

Input used as part of file paths allows attackers to access arbitrary files (in other directories) on the server.

Error Handling

Error conditions caused by malformed input reveal internal information about the server or its current state to users.

File upload

Attackers might try to upload large or malicious files to a system.

In-Band Signaling

Commands in the data channel are accidentally executed by an application, rather than treated as data. This is the generic problem behind e.g. Cross Site Scripting and SQL injection.

Information Disclosure

Important or confidential information might be accidentally revealed to users.

Phishing

Attackers might trick an application into including 3rd party content that appears to be part of the attacked application, possibly misleading users that hold the external content for authentic.

SQL injection

An attacker may alter database queries by sending malicious input to a web application.

For all those areas we created in total 85 *technically distinct* variations for the first version of our test system. Many more test cases are currently under development. There will be extensions to existing test categories as well as new test areas not yet covered such as *HTTP Response Splitting*.

Note that there are several different ways to attack each type of vulnerability. It is therefore important to analyze if scanners are able to identify possible attack variations in order to bypass defensive filters implemented by an application.

Let's explain this in the context of Cross Site Scripting (XSS):

Simply put, the problem of XSS is that user input (from a persistent storage location) is written back to an HTML page without any encoding of potential tags or commands. A typical test if a certain input field is vulnerable would be to enter a string such as `"><script>alert ("XSS Bug!") </script>` as its value. If you submit this data and the following page is vulnerable to XSS and displays your input, then a little popup with the text "XSS Bug!" is displayed. Of course the popup is not exactly an attack, but it proves that you can execute additional code in a page *someone else* opens in a browser.

Feeding this string to input vectors and checking if it will appear unchanged in the response is a typical scanner test case for a Cross Site Scripting vulnerability. Now let's assume a page encodes all < and > characters to counter XSS attacks. Then the scanner would observe that its input was changed, hence it would indicate that there is no vulnerability.

Unfortunately there are dozens of ways to do bad things in an HTML page.

Sometimes a pattern like " `onmouseover=alert()` is all it takes to open a popup and the previously described encoding will not work against this input. It all depends on where exactly the user input is written to and to what extent the given page encodes input.

If you like to explore XSS in more detail, please read reference [10] in the *References / Further Information* section.

To cover the most typical variants of XSS attacks, we built 31 test cases.

Likewise, all other test case areas also cover various methods of attack or show vulnerabilities that are exploitable under specific circumstances. For example, consider a buffer overflow vulnerability that occurs only if the input for field `name` is at least 4096 bytes in length. If the scanner uses a 2048 bytes pattern to check this, the vulnerability remains unnoticed. What if the buffer overflow in field `name` occurs only if field `country` has the value `DE` at the same time? Of course this is almost impossible to detect by a scanner. But that's the whole point of the benchmark: finding out what scanners can't find. Naturally we built one trivial test case for all areas to see if scanners at least try to find every given kind of vulnerability.

Business logic test cases

Business Logic test cases are far more difficult to detect by brainless entities (such as scanners) than the technical test cases. You have to understand a given business process in order to determine if there is a problem or not.

For example let's think of a page that displays personal details of one of the employees in your department: name, birthday, social security number, and salary. You may be their manager and thus permitted to see this.

The corresponding page would be invoked like this: <http://intranet/empdetails.jsp?id=364534>.

But what if you change the value of `id` to another number and hit refresh in your browser? The page might now display another user, possibly from another department where you should have no access. Do you have any idea how a scanner could detect this kind of problem *generically*? We don't.

Obviously, our expectation here was that *black box* scanners are not able to find this kind of problem in an application. We still built several test cases, just to be thorough.

What was impossible from the start

From the last section we learned that there are some bugs that are hard, if not impossible, to detect for scanners. In this section we want to point out that there are also (lot's of) bugs that can't be detected at all from the outside view in a *black box* context.

Let's consider the following scenario: customers logon to their online banking application. They can view their current balance, transfer money etc. But what if the application fails to log the transfers? Or what if transfers are executed through a call to the bank's backend system with a technical user? In both cases the transfer can't be related to the user who initiated it. This is a security as well as a compliance problem. But since it happens "under the hood" of the application, this behavior can't be observed from a user's perspective while executing a transaction. There is no way any scanner (or tester) could detect this from the outside.

If you start thinking about architectural problems like this you will end up with a long list of impossibilities. Clearly this is not part of the benchmark, but still an important thing to keep in mind when talking about the capabilities of *black box* scanners.

Quest for Scanners

After setting up our benchmark platform, the only thing left has been to get our hands on as many scanners as possible. We have googled the web for open and closed source solutions and contacted vendors of commercial scanners. Many of them gave us test licenses after a short talk about what we were up to and were very cooperative. We like to take this opportunity to say "Thank you" once more. Or only promise to all of them was to not reveal any names. We don't want to promote or blame any specific product for its scanning qualities. We just want to see how those tools work in general.

When running the scans we used the tool's set of test cases *as is*, that is we didn't add or modify any test logic. We simply pointed them towards our server and let them do the job. While you could argue whether this is the fairest approach, we decided that it makes no sense to add specific test patterns for the test cases we built into our benchmark platform. Additionally, no matter how good your custom test patterns are, remember that they won't help you if the spidering algorithm of your scanner is insufficient. Besides, out-of-the-box usage of a scanner tool is a common use case for non-experts – they have no clue how to tinker the tool appropriately.

To sum up this chapter, there are several kinds of tests to consider. Within each test area there are (many) different types of potential problems, each of them with a lot of variations.

Also, scanners can only detect a bug if they can deduce its presence from the server's response. This makes technical problems more likely to find because of the error conditions they produce. Business logic issues or architectural problems, however, are extremely difficult to find, if at all.

Chapter III: Benchmark

In this chapter we present the results of the actual benchmark of seven scanners that we have analyzed. Besides the described criteria, we will also talk about the quality of the report the tools produced.

Note that we had seven scanners in the test lab, but we can only provide results for five of them. One of the scanners failed to find our test case list, because the list is located in a subfolder of our test system and that particular scanner could only start its scans in the root (!) folder of a domain. The second scanner simply crashed during its analysis. After three attempts / crashes we took that one off the list. Due to this misfortune, all of the remaining scanners in the race have been commercial products.

In the tables below, we named the scanners *A* to *E* without any relation to vendor or product names. Each table lists the total number of test cases (*#TC*) in a test area, number of vulnerabilities detected per scanner as well as the average (*Avg*) number of vulnerabilities found.

Note that static hyperlinks were not included in the test cases, because resolving them is the minimum requirement for any scanner.

Spidering and form completion

Area	# TC	Scanner under test					Avg
		A	B	C	D	E	
Spidering	19	8	1	0	7	9	5,0
Form Completion	12	8	0	7	9	7	6,2

Some key observations:

- *Spidering* capabilities are at a rather questionable point with two completely unacceptable results.
- *Form completion* is at an acceptable level (with one exception), although not sufficient for applications consisting mainly of forms.
- Scanners B and C have highly insufficient spidering algorithms.
- Scanner B can't fill in forms at all (by design). You have to do this manually for the scanner for *all* forms it encounters, which can give you a severe headache as mentioned in the section *Mechanics of a scanner*.
- *None* of the scanners is able to find all pages in a more complex business application.

Technical test cases

Area	# TC	Scanner under test					Avg
		A	B	C	D	E	
<i>Buffer Overflow</i>	4	0	0	2	0	0	0,4
<i>Code Injection</i>	4	1	0	0	0	1	0,4
Cross Site Scripting	31	1	4	4	3	9	4,2
Directory Traversal	11	0	0	0	0	0	0,0
<i>Error Handling</i>	2	1	0	0	0	0	0,2
File upload	4	0	0	0	0	0	0,0
In-Band Signaling	6	0	0	0	0	0	0,0
Information Disclosure	4	0	1	0	1	1	0,6
<i>Phishing</i>	4	1	0	0	1	0	0,4
SQL injection	7	4	0	4	0	2	2,0

Some key observations:

- The overall result is disappointing.
- There are *three* test areas where the scanners found none of the vulnerabilities (marked as bold), not even the trivial ones. And another *four* (marked as italic) where they found almost none.
- No suite of test cases has been resolved to an acceptable level.
- No scanner was able to spot at least one bug in every area; in fact each scanner failed to detect vulnerabilities in at least five test areas.

The one thing that really puzzled us here was that there were no findings for *Directory Traversal*. We went through the HTTP server logs to find out why: The method used to check if a directory traversal attack is possible was to access specific well know files, one for Linux and one for Microsoft® Windows®. If the files couldn't be retrieved, the scanners following this approach concluded that there was no vulnerability, instead of concluding that the files were not present or *accessible*, as was the case for our test system.

Business logic test cases

Area	# TC	Scanner under test					Avg
		A	B	C	D	E	
Access Control	3	0	0	0	0	0	0,0
Business Logic	2	0	0	0	0	0	0,0
Forceful Browsing	3	0	0	0	0	0	0,0

These test cases include problems like weak authentication, client-side validation, manipulation of state data stored in hidden fields as well as bypassing disabled UI controls.

As we expected, none of the test cases have been resolved. Still, we made the tests to be thorough.

Overall ratings

	#TC	Scanner under test					Avg
		A	B	C	D	E	
Total Issues Found	85	9	4	10	5	13	8,2
False Positives	?	3	5	29	16	1	10,8
Xtras	?	1	1	1	0	1	0,8
Issues missed	85	9	4	5	27	10	11,0

Some key observations:

- The best scanner (E) found 13 out of 85 vulnerabilities, the worst (B) only 4.
- The best scanner (E) reported only one *false positive*, the worst (C) as much as 29.
- Four scanners (and we don't want to hide that from you) found vulnerabilities that were not supposed to be there (*Xtras*), like an information disclosure in one of our code injection examples.
- The best scanner (B) missed only 4 issues, the worst (D) as much as 27. A missed issue (false negative) is a vulnerability where the scanner send a correct pattern to detect a problem but in its subsequent response analysis failed to do so.

Reporting

The final step of the benchmark was to go through the reports and analyze their quality.

In general we have the followings expectations towards a good report:

- o It should list every problem only once. *Most scanners did.*
- o It should not include *false positives*. *We already saw that all scanners failed here.*
- o It should offer recommendations how to solve the problems. *All did to some degree.*
- o It should rate the issues, so the bug fixing process can be prioritized. *Most did.*
- o It should contain an executive summary with the highlights. *Most did.*
- o It should describe the testing scope. *Most scanners allow specifying the test scope.*
- o It should describe the testing methodology. *None did.*

Some key observations:

- The longest reports were 946 and 742 pages in size. Remember that the best scanner spotted only 13 vulnerabilities...
- One scanner reported 7735(!) senseless instances of IP disclosure (instead of aggregating them) even *twice*.
- Most bugs reported by one of the scanners were not reproducible the way they were documented. This is very problematic because developers tend to ignore the problem in such cases.
- Two scanners flagged one test case mistakenly with the same exotic *false positive* as "*Extropia WebBanner Remote Command Execution*". From this we conclude that different scanners share (at least some) common test cases.

Chapter IV: Conclusion

This chapter analyzes the findings of the benchmark test and discusses some consequences.

Scanner efficiency

The test clearly showed several things:

a) Spidering deficits

Result from the benchmark:

Scanners can't resolve all possible types of links; hence they will not find all pages in a web application, *if that application (or underlying framework) uses some of the more complex linking methods.*

Furthermore, scanners can't automatically fill in all forms. This means they either need human support or will not find all pages in a given application, since the forms will run into error conditions. Please also refer to the TCO section later in this chapter.

Consequence:

Scanners can only try to detect vulnerabilities in a page they know about. The more pages a scanner misses during spidering, the higher the risk that some of those missed pages contain a hidden vulnerability.

Workaround:

In order to guarantee coverage, you'd have to compile a list of all web pages in your application. You would then have to compare this list to the list of pages in the scanner report or in your web server log.

Restrictions:

This workaround only works if the web pages don't display content based on input parameters, like e.g. some content management systems and business frameworks do.

Recommendations:

Don't rely on scanners to test complex business applications on frameworks like SAP[®], IBM[®] or Oracle[®].

b) Vulnerability detection

Result from the benchmark:

There are generally three types of vulnerabilities:

1) Technical vulnerabilities, which can be detected (in principle) from the outside through interaction with the software and *technical analysis* of the response (like error codes).

The best scanner found 13 out of 85 technical vulnerabilities (15.3%).

2) Business logic vulnerabilities, which can be detected from the outside only by *understanding the data* in the server response for a given request.

No scanner resolved any of the 8 test cases in this area (which was expected).

3) Architectural deficits, which in most instances can't be detected at all from the outside.

We didn't test for this because we know scanners can't find such a bug even if it would be visible from the outside.

Consequence:

Even if a scanner can find a page, it will most likely only spot the simplest forms of attack for any given type of vulnerability. Many scanner test cases we analyzed worked insufficiently and many types of problems can't be detected at all by a *black box* scanner.

Workaround:

To increase the efficiency of your scanner's vulnerability detection capabilities *for technical test cases* you'd have to tune / extend its pattern recognition database. To validate the efficiency of your modifications, you'll need a benchmark system to test the scanner.

Restrictions:

As already stated, there are some types of vulnerabilities no *black box* scanner can ever find.

Recommendations:

Don't rely on scanner-based tests alone.

Where to use black box scanners

Judging from the previous section, it appears doubtful if *black box* scanners are of any real help at all.

Still, we recommend using them for several reasons:

Scanners are fast and can cover a lot of ground in only a fraction of the time a human needs. If they find bugs they are very useful for (cheap) regression tests against fixes for those bugs.

And although scanners will probably only find a small fraction of all the bugs in an application, they still eliminate a lot of *low hanging fruit*.

Removing the most obvious bugs has two advantages for your company:

- 1) It will keep *Script Kiddies* away (Hackers with little skill but the intend to do harm)
- 2) Security consultants don't waste their skills with trivial bugs and can focus on the more difficult problems.

But please be advised, that *relying **only** on a scanner is definitely the wrong approach*.

If you are serious about the security of your business applications you need human experts with a *completely different* testing methodology.

When to consult human security experts

If you are designing business applications, the first information you need is a risk analysis that helps you understand all potential threats your assets are exposed to. And you need this information *before* you design or even write an application. Addressing the most critical areas from the beginning will save your company not only a lot of money but will also make risks transparent for your management. Whereas scanners can only be brought in at the last stage of application development, consultants can already help you in the security requirements phase.

Of course this won't help you very much if you have applications up and running that need to be tested now. In this case we recommend hiring security consultants to first identify all existing threats for your running applications and then to do specific (code level) testing for the most critical areas of your most critical applications. This approach will make sure you invest your budget wisely, since human testers are more costly than scanners and it's not efficient to manually check every line of code.

When initially screening external security consultants, it is important to make sure they fully understand the technologies you are using. It doesn't make much sense to hire PHP geeks to screen an ESS scenario based on SAP® NetWeaver™.

Unfortunately you can't couple scanner test results with human testers very good. The scanner does not describe its methodology in a way a human tester could continue where the scanner stopped. Also, scanners will probably miss bugs even if their test pattern was sufficient, because they most likely fail in finding all pages of an application. The human tester has no alternative as to start from scratch. The only good thing is that they won't have to deal with many trivial bugs, since the scanner hopefully found most of them. That is, if your company fixed them before hiring the tester...

Some TCO considerations

Besides the licensing cost for the scanner, there are several other aspects to consider:

- 1) Someone in your company will have to operate the scanner. This person needs application security know-how (security training) and must understand how the scanner works (product training).
- 2) The scanner patterns will have to be enhanced, in order to achieve acceptable results (research or expert know-how required).
- 3) Your developers will most likely waste some time dealing with *false positives* or insufficient bug descriptions.
- 4) Scanners can only operate on the *running* application. This is the most expensive point in the development lifecycle to fix bugs.
- 5) For some scanners, your testers will have to assist the tool (remember that filling in forms is not trivial for a tool), thus reducing their availability for other projects / tasks.
- 6) You still need additional budget to hire security experts that weed out the remaining bugs from the system.

...and what about white box scanners?

We talked a lot about *black box* scanners in this paper, which is good, since that was our topic here. For all of you that are interested in *white box* scanners aka *static code analysis* tools, stay tuned, since that area is currently under research in our test lab...

Final thought

If you are running scanners to protect your business applications from attacks, there will most likely be several vulnerabilities left in your software.

Someone will find them.

You can decide who: a hacker or a security expert.

References / Further Information

- [1] Mark Curphey, Rudolph Araujo, Web Application Security Assessment Tools, IEEE Security & Privacy (Vol. 4, No. 4), July / August 2006
- [2] Arian Evans, Software Security Quality: Testing Taxonomy and Testing Tool Classification, 2nd Annual OWASP Conference, Washington DC, Oct. 2005
- [3] Gary McGraw, Software Security: Building Security In, Addison-Wesley Professional, 2006
- [4] Michael Howard, David LeBlanc, and John Viega, 19 Deadly Sins of Software Security – Programming Flaws and How to Fix Them, Osborne McGraw-Hill, 2005
- [7] H.Q. Nguyen, B. Johnson, M. Hackett, Testing Applications on the Web, Wiley Publishing, 2003
- [8] Holger Peine, Stefan Mandel: Sicherheitsprüfwerkzeuge für Web-Anwendungen, Technical Report, FHG IESE, 2006
- [9] Jeffrey Rubin, Review: Web Vulnerability Scanners, Secure Enterprise Magazine, Sept. 2006
- [10] Andreas Wiegenstein: A short story about Cross Site Scripting
<https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/2422>