

STeP-IN SUMMIT 2007

International Conference On
Software Testing

A New Approach to Application Security Testing Tool Design

By

Rajendra Gokhale and Madhura Halasgikar
Aztecsoft

Copyright: STeP-IN Forum and Quality Solutions for Information Technology Pvt. Ltd.

Published with permission for restricted use in STeP-IN SUMMIT 2007 in agreement with full copyrights from owner(s) / author(s) of material. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior consent of the owner(s) / author(s). This edition is manufactured in India and is authorized for distribution only during STeP-IN SUMMIT 2007 as per the applicable conditions.

Practices Experience Knowledge Automation

Produced By

STeP-IN
F o r u m

www.stepinforum.org

Hosted By



www.gsitglobal.com

Audience: Test Architects, Application Security Tool vendors

Key Objective/value to readers: This paper identifies limitations of current Application Security Testing tools and offers a number of suggestions that will help make them more effective.

Introduction

Testing web applications for security is fundamentally harder than testing them for functional correctness¹. Furthermore, since web application security testing is a recent concern, its automated tools are relatively immature. While it is true that automated testing is only one element of a multi-pronged approach towards ensuring application security (that also includes threat modeling, code scanning and code reviews) it is also true that the availability of effective automation tools can make a huge difference in the quality of security testing possible. It therefore makes sense to examine the current limitations of automated security testing tools and consider how they can be improved and made more relevant to the problem they aim to solve.

During the course of our work in web application security assessment we have had occasion to evaluate and use a number of these tools on real-life projects. This has given us an insight into their benefits as well as their limitations. Our experience has led us to conclude that, although these tools do help substantially in application security testing efforts, they are far less effective than they could be. We believe that these limitations are primarily due to tool vendors' adoption of a simplistic approach toward tool design. Perhaps guided more by marketing compulsions than by utilitarian considerations, commercial security testing tools attempt to implement a seductive vision of a wizard-based tool which when pointed at the application under test does a thorough check of the application and produces a beautiful report that will satisfy management. However we believe that effective security assessment of web applications cannot be carried out without the *active* involvement of skilled human test engineers. Our central thesis is that **to be truly effective, security testing tools should seek to *aid* rather than *eliminate* the human test engineer.**

This paper takes a detailed look at how these tools are currently designed, what they do well and what they do not. We explain why the limitations of these tools are serious in nature and how these could be overcome by redesigning these tools in tune with the methodological approach suggested above. In particular, we make a number of specific suggestions for mitigating these limitations.

¹ See [1] "*Challenges in Security Testing of Web Applications by Rajendra Gokhale and Susheel Kumar Sharma Step-In 2006* for a detailed discussion of this topic

How Application Security Tools Work

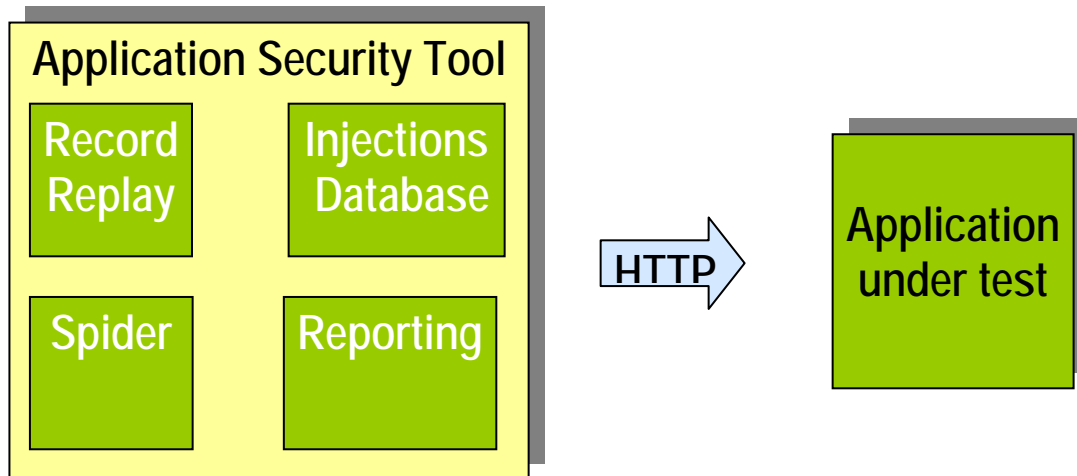


Figure 1 – Typical architecture of application security tools

Application Security Testing tools are often referred to by many names including “Penetration testing tools” and “Application Vulnerability scanners”. Whatever the name chosen, most of these tools follow the fairly standard architecture depicted in

Figure 1.

All these tools follow a common approach towards security testing:

- **Recording** – In this phase the tool tries to learn about the structure of the application in one of two ways:
 - In the **Spidering** mode the tool is given an initial starting URL for the application to be tested along with some information necessary to traverse through the application (e.g. credentials required for accessing the application). The tool then attempts to spider through the application links, keeping track of the addresses and structure of the pages it traverses.
 - In the **manual recording** mode, the human tester browses through the application just as a typical user of the application would in the course of his normal interaction with the application. While this is going on, the tool captures and records information about pages visited as well as data submitted.

-
- **Attack simulation** - During this phase the tool modifies the originally recorded requests and resubmits them with a view to executing various attacks from its dirty-tricks repertoire that is the injections database. Examples of these injections include
 - SQL fragments designed to detect SQL injection vulnerabilities
 - Script code designed to uncover Cross Site scripting vulnerabilities
 - Injection of long text in form fields with a view to induce buffer overflows and/or application exceptions
 - **Analysis** - After the tool has gone through the Attack simulation phase, it analyzes the responses received, collates its findings and creates a report listing out the vulnerabilities observed in the application, along with a canned set of recommendations.

One important characteristic of the approach followed by these tools is that there is apparently no need for an engineer with special knowledge of security testing issues. The human tester is able to control the testing process only to the extent that he can choose what parts of the application to test and which subset of the attacks database to fire².

Limitations of current approach to security tool design

This approach is very effective in testing for a number of vulnerabilities, especially ones that do not require any understanding of the application logic. Examples of such vulnerabilities include:

- Reflected cross-site scripting wherein the site echoes user input that is embedded in the request into the response, in the process leaving an innocent user vulnerable to a number of attacks, including identity theft.
- Buffer overflow vulnerabilities
- Testing the web server configuration
- Testing for HTML comments in response

Appealing as this simple approach may be, it has numerous problems:

- Although these tools can detect *reflected* cross-site scripting vulnerabilities very well, they are not capable of detecting the more subtle *stored* XSS vulnerabilities that require some understanding of how data flows through the application
- A securely designed application may respond to an attempted injection with a session logout. Since the tool has no way of detecting that it has been logged out, it merrily continues to carry out its injections, blissfully unaware that the application is refusing to take cognizance of these requests, coming as they are from an expired session. As a result, the application is never really subjected to a

² The tool “Hailstorm” from Cenzic is an honorable exception insofar as it allows users to add attack “policies” written in JavaScript. While this flexibility is a definite plus, we would prefer to see simpler mechanisms of the type described later to aid the testing process.

number of attacks that the tool claims to have executed³. It takes a lot of diligent efforts from the testing team to uncover the true state of affairs – in the best case a lot of time and effort gets wasted, while in the worst case a security breach occurs when the application is brought online.

- Since tools are incapable of understanding application logic, they end up carrying out numerous attacks that are meaningless in the context of the application. For example, consider a request that involves deleting an item from a shopping cart. If this request involved submitting a form with “N” parameters, one of which represents the item to be removed from the shopping cart, the tool would attempt to manipulate each of these “N” parameters many times and resubmit the same form with minor variations. Unfortunately since the item has already been deleted from the shopping cart after the first request, the application may ignore all subsequent requests and any vulnerability in the application’s handling of the other form parameters on this page could get masked during the test effort.
- They are incapable of detecting *logical* vulnerabilities. For example, these tools have no means of detecting if a bank’s “fund transfer” page allows a user to transfer a negative amount to another user (in other words, it allows a user to transfer a *positive* amount *into his own account*) nor do they have any mechanism to *help* the human testers to suspect this state of affairs
- They cannot detect vulnerabilities such as the “*Insecure Id vulnerability*” wherein a user can pretend to be another user by modifying an “id” field in a form submission or cookie.
- They fail to work well with applications such as Amazon.com that use dynamically generated URLs using techniques such as URL rewriting in which a session-specific string is embedded into all URLs that a user accesses. Since the URLs recorded during the recording phase contain an old timed-out value of this session-specific string, all requests made by the tool are dishonored by the application so that the application never really gets tested meaningfully!

It is evident that these are very serious limitations and in our own experience we have found that most applications have vulnerabilities that would go unnoticed if tested using automated security tools alone.

In search of an effective approach

An analysis of the limitations listed above reveals one common thread – ***in all these cases, the tool would have benefited from enlisting the help of the human being driving it!*** To be able to do this, the tools needs to be designed in such a way that they can be meaningfully controlled and guided by the human tester. For vulnerabilities such as the “insecure id” vulnerability that the tool is itself incapable of detecting, it should do

³ Hailstorm has an (imperfect) mechanism to detect this condition – if it notices that it has received the same response to a specified number of consecutive requests it assumes that the session has been logged out and attempts to re-login.

everything in its power to *help* the human tester. In other words, the right approach to building tools for this space would be to marry the computer's ability to carry out large numbers of repetitive tasks with the intelligence, creative genius and domain knowledge of a security tester.

Broadly speaking, we have identified a number of categories of improvements that will help these tools to achieve these objectives:

- Mechanisms that allow testers to guide the tool
- Mechanisms that allow testers to control the tool during test execution
- Mechanisms to ease testers' tasks during recording/scripting
- Mechanisms that give testers a better way to analyze test results

Here is how a tool incorporating these improvements would address each of the limitations we listed previously:

- **Testing for “Stored XSS”:** The tool should allow a user to model data flow through the system. For example, if the tester could tell the tool that an article submitted by a bulletin board user on the “Submit Article” page gets displayed to another user reading articles on the “Read Article” page, it would be possible for the tool to represent this fact internally in the form of a directed graph. The tool could then use this graph to infer a correlation between the “Submit Article” and the “Read Article” pages and design its cross-site scripting attacks accordingly.
- **Detecting session logout:** The tool should allow itself to be trained to detect session logout. This could be achieved through the simple mechanism of allowing the user to specify that a redirect to a specified URL (e.g. a redirect to the login page) indicates that the application has logged out the session and that the tool needs to do a re-login into the application before continuing.
- **Avoiding multiple deletes of same item from shopping cart:** This can be avoided in two ways:
 - The tool could allow the user to specify a URL to be submitted after every injection on a given page. This URL could roll back the effects of the last request – in the present case, it could be used to restore the item back to the shopping cart
 - The tool could allow the user to embed code that is to be executed after every injection. This code could be used to undo the effects of the last request
- **Preventing transfer of a negative amount:** Tools could be enhanced so that they can report client side validations to the tester. For example, the tool may have a feature whereby it fills a form with strange values and attempts to submit it using a full-fledged browser implementation. It should check to see whether the browser actually submitted the request. Detecting that the browser has not submitted the request would signal to the tool that submitted values are not being accepted due to client-side validation. This would be reported to the tester, who would then understand the need for designing appropriate logical tests that bypass

client-side validation. In our “negative amount transfer” example, the tester would learn that the transfer of negative amounts may be an interesting test. He could then design a test wherein the tool bypasses the client-side validation code and checks to see if the resulting response contains the string “funds transfer successful”. The point we are making here is not that the tool will be able to detect this or other vulnerabilities of this nature – rather we are saying that with some thought, it would be possible to add many such features that will enlist the tools in guiding and aiding the human testers to home in on such logical vulnerabilities.

- **Detecting “Insecure Id” vulnerability:** The tool should analyze all forms and cookies submitted during the recording phase. It should apply heuristics to the names and values of each of these parameters and present the human tester with a list of parameters that are potentially “id” parameters. Examples of such heuristics include:
 - Parameters with names including strings such as “id”, “num”, etc. are likely to be “id” parameters
 - Parameters with integer or alphanumeric values
 - Parameters having constant values
- **Handling URL rewriting:** The tool should allow the tester to specify what part of a URL is session-specific and also allow him to spell out how this session-specific portion of a URL should be inferred during test execution. Load testing tools already have this functionality so it is neither hard to design nor to implement this functionality.

Conclusion

We have attempted to demonstrate that redesigning application security tools with the goal of empowering the human security tester will make them far more powerful and relevant to the needs of web application security testing. While we do not claim to have all the prescriptions for making this happen, we are confident that tool designers will be able to come up with many effective solutions if they only approach the problem of tool design with this different mindset. We are optimistic that the world will then be a somewhat safer place.

References

1. *Challenges in Security Testing of Web Applications* by Rajendra Gokhale and Susheel Kumar Sharma Step-In 2006
2. <http://www.iese.fraunhofer.de/download/Security-Checker-Tools-for-Web-Applications.pdf> has a good discussion on the strengths and weaknesses of security tools and anticipates some of the suggestions we make in this paper.
3. <http://seclists.org/webappsec/2006/q2/0308.html> contains a rebuttal by Ory Segal of Watchfire, a tool vendor of some points made in the previous reference.

-
4. http://www.whitehatsec.com/presentations/challenges_of_scanning.pdf by Jeremiah Grossman highlights the inability of automated tools to test for logical flaws.
 5. http://searchappsecurity.techtarget.com/tip/0,289483,sid92_gci1189767,00.html?bucket=ETA&topic=302537 “Technology alone cannot defeat Web application attacks: Understanding technical vs. logical vulnerabilities” by Grossman