Analyzing the OWASP Benchmark Project

Heinrich Rust, Norman Wenzel
RIGS IT GmbH

Last modified: February 28, 2020
Analyzing the OWASP Benchmark Project

Abstract: We will describe how the static application security test tool Xanitizer is set up for analyzing the OWASP benchmark project. This set-up is fairly easy, mostly the default settings can be used. The analysis results are quite good: all problematic test cases are detected, and the number of false alarms is also quite low, at least compared to the open source tools for which there are results on the benchmark's site. When interpreting the results, the differences between the OWASP benchmark project and typical real-world projects and their effects on the analysis results have to be kept in mind.

Table of Contents
1. Introduction .......................................................................................................................... 3
2. Analysis with Default Settings .......................................................................................... 5
3. Analysis with HTML Sanitizers for XSS ........................................................................... 7
4. Speeding Up the Analysis .................................................................................................. 10
   4.1. Reducing the Amount of Code to be Analyzed ....................................................... 10
   4.2. Removing Unneeded JAR Files ............................................................................. 11
5. Calling for Caution .............................................................................................................. 12
Analyzing the OWASP Benchmark Project

1. Introduction

Xanitizer ([www.xanitizer.com](http://www.xanitizer.com)) is a commercial SAST (static application security testing) tool for the analysis of Java Web applications, running on 64-bit variants of Windows, MacOS and Linux. Xanitizer is itself written in Java, and is primarily based on taint analysis, but it also allows to use some other techniques for detecting security flaws.

The OWASP benchmark project ([www.owasp.org/index.php/Benchmark](http://www.owasp.org/index.php/Benchmark)) is an open-source benchmark for Java Web applications. In version 1.2beta, it comprises about 2700 test case/weakness pairs, and for each of these pairs, it is specified if a security vulnerability for that weakness should be found in the test case or not. Thus, it can not only be used to measure what percentage of the problems are detected by an analysis (called “True Positive Rate” in the OWASP benchmark project), but also how many false alarms are generated, or more exactly, what percentage of the non-problems are signaled as problems (“False Positive Rate”).

The project comes with precomputed results for a number of commercial and non-commercial analysis tools. The accumulated scores for some tools are presented in Scorecard 1. Note that for the “True Positive Rate”, 100% is optimal, while for the “False Positive Rate”, 0% is optimal, and thus the overall optimal results are in the upper left corner. The numbers in the list on the right side of the chart are computed as the difference “True Positive Rate” - “False Positive Rate”. Values on the diagonal can be produced by simply guessing the detector result for each test case.

![OWASP Benchmark Results Comparison](https://rawgit.com/OWASP/Benchmark/master/scorecard/OWASP_Benchmark_Home.html)

*Scorecard 1: OWASP Benchmark Scorecard for different tools as supplied on https://rawgit.com/OWASP/Benchmark/master/scorecard/OWASP_Benchmark_Home.html (retrieved 2016-02-25)*
We first describe how Xanitizer is set up for analyzing the OWASP benchmark project, and describe the results with the default settings. Then we describe how to use project-specific knowledge in order to refine the set-up, and how this leads to better or quicker results. Finally, we discuss shortly how relevant our results are for real-world projects.

For this work, we used the OWASP benchmark project of version 1.2beta, and the Xanitizer version 2.1.0.
2. Analysis with Default Settings

We first describe the simplest set-up.

Let us assume that the OWASP benchmark project is located somewhere in the file system, and let us call the base directory `<OWASPDir>`, i.e., the directory containing the Maven configuration file “pom.xml”. Xanitizer needs byte code in order to analyze a project, which means that the project must be compiled. This is easily done using Maven, by invoking `mvn clean compile` in the `<OWASPDir>`.

Let us denote Xanitizer's installation directory with `<XanitizerDir>`.

We describe how Xanitizer is set up for the OWASP dependency project step-by-step:

1. Start Xanitizer's GUI, by invoking `<XanitizerDir>/Xanitizer`.

2. When Xanitizer is started for the first time, a dialog appears in which some settings can be specified, most importantly the main memory to be used for heap by the JVM. The default settings can be used for analyzing the OWASP benchmark project, assuming there is enough main memory available.

3. Create a Xanitizer project for the OWASP benchmark project:
   a) Invocation of the main menu item "Project > New..." starts the Xanitizer project creation wizard. On the first page of the wizard, we use “OWASPBenchmarkDefaultSettings” as "Project Name", and `<OWASPDir>` as "Project Root Directory". The last step automatically registers `<OWASPDir>` also as the "Configuration File Directory". This means that Xanitizer's configuration file will be stored here. We keep also this setting. We press “Next” in order to leave the first page of the wizard.
   
   b) Before the second wizard page is shown, Xanitizer collects, starting from the project's root directory, locations of JAR files, byte code files, source files etc. On the second page, the result of that search is shown and can be edited. We just keep the default settings and press “Next” again.
   
   c) From the source code files that have been found, Xanitizer now determines patterns for classes that the tool considers to belong to the project itself (and which are not just library code), and displays them on a third page. The default is ok in our case, so we press “Next” again.
   
   d) On the last page, the problem types that will be used during the analysis can be selected. In addition to the problem types natively supported by Xanitizer, also two integrated tools can be used for looking for security vulnerabilities, namely “OWASP Dependency Check” ([www.owasp.org/index.php/OWASP_Dependency_Check](http://www.owasp.org/index.php/OWASP_Dependency_Check)) and “Findbugs” ([findbugs.sourceforge.net](http://findbugs.sourceforge.net)). We switch them off, because we are only interested in the benchmark results of Xanitizer and the other tools would interfere.
   
   e) Pressing "Finish" leaves the wizard. Xanitizer now creates an internal representation of the project from the data that has been collected.
4. The analysis is started with menu item “**Analysis > Run Security Analysis**”. This can take quite some time; for example, on one of our test machines, it takes about 20 minutes. Xanitizer builds a call graph, looks for taint sources and taint sinks (for different problem types), and searches for paths between taint sources and taint sinks.

5. When the analysis has finished, we export the results into an XML file that can be processed by OWASP's benchmark project's scoring functionality. We invoke menu item “**Reporting > Generate Findings List Report...**”, and in the dialog that appears, specify “XML” as format, specify a location for the output file, and press “Generate Report”.

A parser for Xanitizer's output file was added to the OWASP Benchmark project in April 2016. The scorecard for Xanitizer can be created by invoking `mvn validate -Pbenchmarkscore -Dexec.args="expectedresults-1.2beta.csv <OutputDir>"` in the `<OWASPDir>`. `<OutputDir>` is the directory that contains the generated XML report file.

When the resulting XML file is processed by the scoring functionality of the OWASP benchmark project, the diagram shown as “Scorecard 2” is produced which displays the overall results for all the test categories. The “True Positive Rate” is 100% for all categories. This means that all problems in the code have been identified. The “False Positive Rate” lies between 0% (the optimum) and 37% (100% - 63%, for Cross Site Scripting).

Note that this result has been achieved just with default settings. It is not difficult to reduce the number of false alarms for Cross Site Scripting, as will be shown in the next section.

![Benchmark v1.2beta Scorecard for Xanitizer](image)

**Scorecard 2: OWASP Benchmark Score Card for Xanitizer with Default Settings**
3. Analysis with HTML Sanitizers for XSS

Xanitizer comes with a large number of predefined detectors for different problem types. For XSS (cross site scripting) and some other problem types, Xanitizer determines where tainted data can enter the application code (i.e., “taint sources”), and where tainted data can cause harm (i.e., “taint sinks”), and it then checks if there are data flow paths from taint sources to taint sinks. If there is such a path, a vulnerability is signaled.

Sometimes, the software developer uses a “taint sanitizer” on the data flow path from a taint source to a taint sink. This is a code fragment that changes the flowing data in a way so that it cannot cause harm at the taint sink. For XSS, if the taint sink is an output of text in an HTML context, the escaping of special HTML-characters can be such a taint sanitizer.

In Xanitizer, there is a predefined definition of some commonly used HTML sanitizers. But by default, they are not assigned to the XSS problem types (Xanitizer has several XSS problem types; for example, distinguishes between “Reflected XSS” where the taint source gets its data from the browser, and “Stored XSS” where the tainted data comes from a database).

The reason for not activating this sanitizer by default is that in some scenarios, the output of a web application does not have to be escaped for an HTML context, but, e.g., for a JavaScript, a Cascading Style Sheet or a JSON context. Since Xanitizer does not detect the context of an output, developer knowledge is needed to decide if the use of an HTML sanitizer is appropriate or not.

By looking at the code in the OWASP benchmark project, we see that the HTML sanitizers are only used in an HTML output context. Thus, whenever an HTML sanitizer is touched during a flow of tainted data from a taint source to a taint sink, this path should be suppressed.

We now describe step-by-step how Xanitizer's configuration can be changed to reflect this knowledge:

1. We first save the existing configuration under a new name that reflects the new scenario. This is not strictly necessary, but it helps in order to distinguish different versions of the analysis results. We invoke the menu item “Project > Save Project Configuration As…”, in the appearing dialog click “Browse”, and replace the configuration file name by “OWASPBenchmarkHTMLCleanerForXSS.XanitizerConfig” in the same directory. The new name reflects that we intend to register an HTML cleaner (i.e., a sanitizer) with the XSS problem types. Clicking “OK” twice leads us back to the main window.

2. We edit the configuration via main menu item “Project > Edit Project Configuration…”. This opens a tab titled “Configuration for ‘OWASPBenchmarkHTMLCleanerForXSS’” on the right-hand side of the GUI. This tab contains a number of sub-tabs which can be selected at the bottom of the page. We select the tab “Taint Flow Problem Types”. Here, the taint flow problem types can be configured. This page is split in halves:

   (a) On the left-hand side, the taint flow problem types are listed in a tree. Beneath each node, the taint sources, taint sinks and taint sanitizers to be used for this problem type are given. The problem types include “XSS Stored” and “XSS Reflected”, two XSS variants Xanitizer supports.

   (b) On the right-hand side, the currently defined taint sources, taint sinks and taint sanitizers are listed. The sanitizer kinds include “HTML Cleaner”.

Page 7/13
When we look into “XSS Stored” and “XSS Reflected”, we see that “HTML Cleaner” is not listed as a sanitizer of any of these two problem types. By drag-and-drop, we change this: We first drag “HTML Cleaner” and drop it on “XSS Stored”, and then we drag “HTML Cleaner” once more and drop it on “XSS Reflected”. The two XSS nodes' color now has turned from black to blue (this signals that there are some user-changes), and when we expand them, we see that “HTML Cleaner” (in blue again) is now listed underneath these two problem types.

A warning note is in order here: What we just did means that always when a taint flow for XSS touches some method listed under “HTML Cleaner”, this taint flow will end, and the taint sink will not be reached. If the taint sink can lead to an output in non-HTML context, this HTML cleaning might be just insufficient to avoid cross site scripting. Thus, this assignment of the sanitizer “HTML Cleaner” to the problem types “XSS Stored” and “XSS Reflected” means that we trust the developers that they use the HTML sanitizer methods only in the proper contexts.

3. Menu item “Project > Save Project Configuration” saves the changed version of the configuration file to disk, menu item “Analysis > Run Security Analysis” runs the analysis with the changed configuration, and after the analysis has ended, “Reporting > Generate Findings List Report...” is used to generate an XML report, as before.

The scorecard resulting from the change is given in Scorecard 3. The false alarms for cross site scripting have been reduced to 10%, other values did not change.

![Benchmark v1.2beta Scorecard for Xanitizer](image)

Scorecard 3: OWASP Benchmark Score Card for Xanitizer with HTML Cleaner Registered for XSS Problem Types
In Scorecard 4 you can see the benchmark score of the Xanitizer analysis compared to other free tools.

Scorecard 4: OWASP Benchmark Scorecard for different tools compared to Xanitizer result
4. Speeding Up the Analysis

For many projects, Xanitizer requires quite some time and memory. Sometimes, it is not possible to analyze a project as a whole (typically because of memory constraints), and it must be cut into pieces – in which case taint flows between different pieces would not be found.

For the OWASP benchmark project, we can do several things in order to speed it up. Here, we will discuss the following:

- Reducing the amount of code to be analyzed
- Removing some JAR files from the configuration file

4.1. Reducing the Amount of Code to be Analyzed

Many Web applications are built using frameworks. Frameworks often use reflection and other mechanisms that are not easy to deal with during a static code analysis. Because of this, Xanitizer does not attempt to analyze the framework code itself, but it generates Java code that simulates the activity of the framework for the project at hand, and in a form that can be analyzed by Xanitizer.

Sometimes, this generated code can be very voluminous, leading to resource problems during the analysis. For the OWASP benchmark project, Xanitizer generates a large number of methods that invoke the `doPost(...)` methods of the test cases. While this does not simulate a proper framework, it does simulate the behavior of the standard Web servlet mechanism implemented by most Java-based Web servers.

But here, this simulation is not necessary. For the OWASP benchmark project, all the interesting things happen in the `doPost(...)` methods of the test case classes. Using this knowledge, we can switch off all the framework simulations, and specify directly to Xanitizer where the control flow enters the application code from outside.

We describe this again step by step:

1. As before, we first save the existing configuration under a name that reflects the new scenario: Invoke the menu item “Project > Save Project Configuration As...”, click “Browse” in the appearing dialog, and replace the configuration file name by “OWASPBenchmarkPerformanceTuning.XanitizerConfig”. As before, clicking “OK” twice leads us back to the main window.

2. If the Configuration view is not already open, open it via main menu item “Project > Edit Project Configuration...”.

3. In the Configuration view we open the tab “Frameworks”. We switch off all the framework simulations, also the one for “Enable Basic Generic Framework Support” in the lower half.
4. Registering `doPost(...)` as start methods: Now we choose the tab "Control Flow Start Methods". On the blue node "My Start Methods", we invoke the context menu entry "**Add Start Method**". A wizard is shown that allows to select a start method. We type "doPost" in the "Method Name" field. All methods starting with "doPost" are then listed. We double-click the first one. This switches to a new page on which the selected method is described. In the "Declaring Class", the fully qualified name of the class ending in "BenchmarkTest00001" is specified. Since we do not want to register the `doPost(...)` method for each benchmark test class, we turn that entry into a pattern, by replacing "00001" by a single star, so that it looks like this:

"Next" leads to the last page. We can use the default settings here, and press "Finish".

### 4.2. Removing Unneeded JAR Files

Xanitizer's configuration for the OWASP benchmark project can be further optimized. For example, most of the many JAR files that Xanitizer has picked up during the initial project set-up are not in fact needed for the analysis, and removing them from the configuration can again speed up the analysis. Xanitizer contains some functionality which helps to identify possibly extraneous JAR files.

On the left-hand side of the GUI, there is a tab "Issues & Information". There, under the node "Workspace Issues", we find a node “Library Not Needed”. Underneath this node, libraries are listed for which Xanitizer did not find references in the workspace code. We will remove these libraries from Xanitizer's configuration:

1. If the Configuration view is not already open, open it via main menu item “**Project > Edit Project Configuration...**”. In the Configuration view we open the tab “Search Paths”.
2. First, we remove the search path `target/benchmark.war`. This is because all the JARs contained in this WAR file occur as copies directly in the “target” directory, and thus, the WAR file need not be considered.
3. Then, all JAR files underneath `target/benchmark/WEB-INF/lib` that occur in the issue category “Library Not Needed” are removed. After this, about 30 JAR files will have been kept back.

We save the configuration and run the analysis. The running time is about half as long as that of the analysis before. We save the analysis results in an XML file again, and feed the result into the scoring functionality of the OWASP benchmark project. The results are equal to the one before the optimizations.
5. Calling for Caution

Xanitizer works fairly well on the OWASP benchmark project, but some cautious remarks are in order: One should not expect similar vulnerability detection quality in typical real-world projects, especially not out-of-the-box.

The following aspects of the OWASP benchmark project differs from real-world problems:

1. **Path Length:** Xanitizer performs a data flow analysis, following tainted data from taint sources to taint sinks. In the OWASP benchmark projects, these paths are not very long.

   In a real-world project, such paths can be much longer, can span many method calls, and can involve tainted data that is stored on the heap rather than in local variables. All these differences make an analysis more difficult (often, unfortunately, far more difficult): The analysis needs more time and memory, and the number of false alarms will rise because of simplifying and generalizing assumptions that Xanitizer has to use in order to keep needed memory and runtime in check.

   Xanitizer's memory requirements and running time do not, unfortunately, simply grow proportionally with the number of classes or byte code instruction of the project. Other properties of the project, like data dependencies, and the number of methods that depend on one another in a single control flow, are very important for resource needs, and are not easy to estimate.

2. **Frameworks:** The OWASP benchmark project does not use Web application frameworks.

   As already mentioned, Xanitizer generates Java source code for the simulation of the control- and data flow in Web application frameworks, because the framework code itself typically is based essentially on reflection that is quite difficult to analyze. For large systems, this source code can become voluminous, leading to long running times and large memory requirements.

   Additionally, some Web application frameworks or their components have a long release history, like, e.g., Spring. This means that there might be a large number of ways to configure a Web application, as with XML files, by letting the application code implement specific interfaces, or with annotations. Sometimes, several of these methods are used in the same project. Xanitizer tries to support most common cases, but there are typically some configuration technologies and combinations which are not (yet) completely supported.

3. ** Templating Engines:** The OWASP benchmark project does not use templating engines.

   XSS vulnerabilities depend essentially on properties of the templating engine in use. Xanitizer attempts to analyze, in addition to the project's byte code, also the templates that it finds. But there is such a vast number of different template mechanisms, and some of them are themselves so feature-rich, that this template analysis is (a) often not complete or not yet supported, and (b) for template mechanisms that are supported, it is more error-prone than byte-code analysis, and (c) when a system contains many templates, it can become quite resource-intensive.

4. **Project Specific Taint Sources, Taint Sinks and Taint Sanitizers:** The OWASP benchmark uses standard Java libraries or common libraries for accessing possibly tainted data and writing it to possibly taint sinks.
Taint sources, taint sinks and taint sanitizers that are provided in standard Java libraries are mostly detected out-of-the-box by Xanitizer. For a real-world project that uses in-house libraries, Xanitizer has to be adapted: Definitions for taint sources, taint sinks and taint sanitizers would have to be devised for the libraries. With Xanitizer's GUI, the user can do this her- or himself; this adaptation need not be done in Xanitizer's source code itself.

5. **Callgraph Size:** The OWASP benchmark project contains several thousands of project classes, but they do not depend much on each other.

   In real-world projects of the size of the OWASP benchmark project, the interdependence of the classes is typically much higher, which means that Xanitizer needs more memory and time for representing and analyzing the system. This means that in a real-world project, the user might need to cut the whole system into parts that can be analyzed independently from each other. In order to do this, one would have to know the system under analysis, so that one does not cut the system into parts that might be connected by a common taint path.

   Xanitizer provides a large number of tuning options that can help to turn an unmanageable or just barely manageable analysis into a quick one, but they typically require that one is well versed (a) in the technology of taint analysis in general, and (b) in Xanitizer's behavior in particular.

6. **Constant Conditions:** Xanitizer evaluates some constant conditions in the code in order to avoid some false alarms. Only constant conditions based on boolean values, numbers and some string functions are evaluated.

   In the OWASP benchmark project, this evaluation of constant conditions allows to avoid quite a number of false alarms. In real-world projects, the evaluation of constant conditions is not expected to be as effective for avoiding false alarms: Often, the structure of collections on the heap would have to be analyzed in detail, which Xanitizer does not do.

In short, the OWASP benchmark project is, for Xanitizer, a project that is fairly easy to analyze, but for other projects, typically more work has to be done so that Xanitizer can deliver results of high quality in acceptable time.

But Xanitizer's architecture is versatile enough to allow extensions and adaptations to project specific requirements and many of these adaptations can be performed by the end user.