SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2401

Bad and good news about using software assurance tools

James A. Kupsch[1,2], Elisa Heymann[1,3,*,†], Barton Miller[1,2] and Vamshi Basupalli[1,2]

[1]*Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA*
[2]*Software Assurance Marketplace (SWAMP), 330 N. Orchard St., Madison, WI 53715, USA*
[3]*Computer Architecture and Operating Systems Department, Universitat Autonoma de Barcelona, Barcelona, Spain*

SUMMARY

Software assurance tools – tools that scan the source or binary code of a program to find weaknesses – are the first line of defense in assessing the security of a software project. Even though there are a plethora of such tools available, with multiple tools for almost every programming language, adoption of these tools is spotty at best. And even though different tools have distinct abilities to find different kinds of weaknesses, the use of multiple tools is even less common. And when the tools are used (or attempted to be used), they are often used in ways that reduce their effectiveness. We present a step-by-step discussion of how to use a software assurance tool, describing the challenges that can occur in this process. We also present quantitative evidence about the effects that can occur when assurance tools are applied in a simplistic or naive way. We base this presentation on our direct experiences with using a wide variety of assurance tools. We then present the US Department of Homeland Security funded Software Assurance Marketplace (SWAMP), an open facility where users can upload their software to have it automatically and continually assessed by a variety of tools. The goal of the SWAMP is to simplify the task of the programmer in using assurance tools, thereby removing many of the obstacles to their adoption. Copyright © 2016 The Authors. *Software: Practice and Experience* Published by John Wiley & Sons, Ltd.

Received 27 November 2015; Revised 19 February 2016; Accepted 15 March 2016

KEY WORDS: continuous assurance; static analysis; software security

1. INTRODUCTION

For any modern software project, with a face to the Internet, security assessment is an essential part of the software development life cycle. The most basic first step in assessment of the software is to run static analysis tools that scan the software's source or binary code to identify flaws or (more properly) *weaknesses* in the software. Such tools are widely available now, with examples including such commercial tools as coverity analysis [1][2] (C/C++/Java/C#), HP Fortify [3] (C/C++), Klockwork [4] (C/C++/Java/C#), Red Lizards Goanna [5] (C/C++), GrammaTech CodeSonar [6][7] (C/C++), IBM Security AppScan Source [8] (Java/Objective-C/JavaScript/HTML5/Cordova), and Parasoft C/C++test [9] (C/C++) and Jtest [10] (Java); and such open source tools as clang static analyzer [11] (C/C++/Objective-C), Cppcheck [12] (C/C++), OCLint [13] (C/C++/Objective-C), FindBugs [14] [15] with Find Security Bugs [16] (Java), PMD [17] (Java), Checkstyle [18] (Java), Error Prone [19] (Java), Pylint [20] (Python), Bandit [21] (Python), Flake8 [22] (Python),

*Correspondence to: Elisa Heymann, Computer Architecture and Operating Systems, Univesitat Autonoma de Barcelona, Barcelona, Spain.
†E-mail: elisa.heymann@uab.cat
The copyright line for this article was changed on 12 May 2016 after original online publication.

Dawn [23] (Ruby), Brakeman [24] (Ruby on Rails), PHP_CodeSniffer [25] (PHP), RIPS [26] (PHP), JSLint [27] (Javascript), Flow [28] (Javascript) and FxCop [29] (.NET CIL); and such tools as a service as Coverity Scan [30], Veracode [31], and Fortify on Demand [32].

Given the huge selection of such tools, their obvious value, and their easy availability, there should be little reason to not include such tools in a project's development cycle. However, such tools have surprisingly small market penetration. In recent studies by the SANS Institute [33] and BSIMM [34], surveying companies that already had software security programs in place, at least 25% of these companies, reported no use of software analysis tools. While awareness of the need for such tools and (the constantly surprising) resistance to fund security as a part of the software development process play a part in the slow adoption rate of these static analysis tools, there are other more fundamental problems.

First, different tools have different objectives; for example, one tool maybe be designed to generate results quickly at the expense of deeper semantic analysis of the code (with the obvious limitations in the quality of the results), while another may only identify a specific set of weaknesses in web applications, and a third may try to produce only sound results on a specific class of weaknesses such as memory buffer errors. Therefore, it is likely that more than a single tool will be needed to be used in large software projects.

Second, the effort to obtain and install these tools can be viewed as excessive by software developers or support teams. Programming groups have limited resources available for adding security to the development process and also have limited patience for selecting tools, obtaining these new tools installed, dealing with licensing issues, and getting the tools to work with the project's code base. Keeping up with new releases of the tools only exacerbates the problem.

Third, and related to the second issue, is configuring the tool so that it produces useful output. It is not unusual for a tool with default settings to produce so little or so much output that it provides no useful information to the developers. The tools can have a huge number of settings, many of which pertain to stylistic checks that may not apply to many organizations, and different tools have completely different selections of settings. The effort to tune this selection, to reduce the noise in the reports and still provide effective reports, can require in-depth knowledge of each tool and take significant time and experimentation. Programmers, support groups, and managers can easily conclude that the tools have no value before the tools have a chance to be deployed properly.

Fourth, applying the tool to complex programs requires identifying which source files should be assessed. Given the multi-level build structure of real-world applications, build-time generation of source files, and complex include structures, running a tool against the correct files can be an intricate process. Many of the widely distributed commercial tools have provided monitoring and automation to help the programmer with this step. Unfortunately, some of the commercial tools and most of the open source tools do not provide such automation support, so create an additional level of complexity in using what might otherwise be an effective tool.

As an attempt to simplify the process of adopting security analysis tools, and addressing the above four issues, the Software Assurance Marketplace (SWAMP) project [35] was established as an open facility to automate the application of analysis tools to software packages written in a variety of programming languages (currently C, C++, Java (including Android apps), and Python). The SWAMP has a wide variety of both open source and commercial tools installed, *and* set to run with an effective set of options. The SWAMP also provides the build monitoring and automation to allow even the simplest tool to run on programs that have complex build processes. Users can select multiple tools to run against their software package, obtaining unified merged reports from these tools.

In this paper, we first describe the barriers to use analysis tools in more detail. Based on our experiences installing, configuring, and using a wide variety of software analysis tools, we provide concrete examples and quantified data (Section 2). We then describe the SWAMP facility and describe how it automates many of the difficult steps in using analysis tools (Section 3). The appendix shows an example of using the SWAMP.

2. BARRIERS TO TOOL USE

Using automated tools for vulnerability assessment during the development cycle of an application is crucial, as the cost of fixing a weakness at development time is significantly smaller than fixing it when the system is deployed and in production [36]. Moreover, if a vulnerability is found and exploited by attackers, then the costs increase even more. These costs include the strong possibility of different resources being compromised, such as exfiltrating personal information, modifying a database or installing malicious software.

Nevertheless, using automated assessment tools can often be complicated and frustrating. In this section, we describe a summary of our experiences using a wide collection of both open source and commercial automated assessment tools for C, C++, and Java code. We describe the issues that a user faces when using these tools, including obtaining and installing the tool, configuring it, learning how to use it, preparing the application program to be assessed by the tool, navigating through the results generated by the tool, and understanding those results.

To better understand the complexities of using assessment tools, we went through the complete process of assessing several application programs with several tools. We report on our experiences using six different automated assessment tools, three open source tools, and three commercial tools. The purpose of this section is to describe our experiences with tools in general and use these six tools as examples of the type of things we have found over a much broader collection of tools; the purpose is not to compare the effectiveness of the tools (we have performed that in a previous publication [37]). Note that our current license agreements for the commercial tools forbid both publishing results from these tools and comparing results with other tools. Therefore, we are not able to disclose the names of the commercial tools used. We will refer to the Java commercial tool as Comm-Java-Tool-1, and the two C/C++ tools as Comm-C-Tool-1 and Comm-C-Tool-2.

For the software to be assessed, we selected eight C and C++ programs and eleven Java programs. These programs are from the NIST/NSA Juliet test suite [38] and have labeled known vulnerabilities. We simplified them so that they had only the code that demonstrates the vulnerability and code that demonstrates how to avoid the vulnerability. We scanned each of these programs with the applicable assessment tools. The vulnerabilities that could be exploited in the eight C and C++ programs were as follows: path traversal manipulation, command injection, buffer overflow, integer overflow, sensitive information uncleared before release, hard coded passwords, race condition (TOCTOU), and sensitive information exposure. The vulnerabilities that could be exploited in the eleven Java programs were as follows: path traversal manipulation, OS command injection, cross-site scripting (XSS), improper neutralization of script in an error message web page, integer overflow, sensitive information uncleared before release, uncaught exception, hard coded passwords, information exposure through shell error messages, open redirect, and sensitive cookie in HTTPS session without the 'secure' attribute.

2.1. The steps to get a tool to work on your code

The process of installing and using a static assessment tool involves several steps, many of which have complicated choices. These steps need to be followed for each tool that is installed on each operating system platform. Later, we summarize these steps, discussing the challenges faced at each one.

1. *Find the latest version of the tool:* The selected tool needs to be appropriate for the architecture and operating system of the machine on which the tests will be run.
2. *Download the tool:* The tool needs to be downloaded, along with any security extensions, visualization tools and other needed components. The downloading step can be surprisingly time-consuming given the widely varying download sizes for different tools. Commercial tools are often quite large. For example, the download size for Comm-C-Tool-1 is 263 MB, Comm-C-Tool-2 is 3 GB, and Comm-Java-TOOL-1 is 770 MB. Open source packages (including tools) often have extensive dependencies on other packages. For example, Clang needs the LLVM compiler infrastructure, and without any tool extras has a download size of 476 MB. FindBugs weighs in at an economical 8.5 MB, and the compressed Cppcheck is 985 KB.

3. *Install the tool:* Typically, installing a tool is carried out on the machine where the assessments will be conducted (exceptions are where the tool is a service, and even in that case, there may need to be a client program installed). Many tools need a significant amount of disk space, usually larger than the downloaded file system. For example, Comm-C-Tool-2 once installed uses 4.8 GB of disk, and Comm-Java-Tool-1 uses 1.8 GB. As an example of the kind of problems that users may encounter, the space needed for all the tools together was greater than the amount we had available in a regular user account, so the tools were installed in an extra disk partition that was not automatically backed up.

   Some tools have simple installation processes, while others have additional steps such as the need to build the tool's executable from source code, or to download and build other components on which the tool is dependent. The need to build a tool (and its components) can be both time-consuming and error prone. For example, the build process for Clang and its dependent package, LLVM, ran overnight on our 64-bit Pentium platform. Open source tools are more likely to be built from the source code than commercial tools.
4. *Activating the tool:* If the tool is commercial, the user will have to obtain and install a license. The license may be associated with a particular host, user, or group. Typically, this requires the user to first run a command on the host where the license will be used. This command will generate a signature for that host. The signature is then sent to the vendor who returns a license keyed to the signature. The tool is then configured to use the license file; or in the case of a network license, a license server is installed, configured and operated with the license and the tool if configured to use the license server. The license must be periodically updated due to expiration, new versions of the tool, or limitations enforced by the license such as total lines of code allowed.
5. *Configure the user's environment to support the tool:* As with many software packages, it is common to have to modify environment variables, such as PATH in Linux, to include the directory that contains the tool's binaries. Most of the time on Windows, this step is often carried out automatically.
6. *Install additional software:* Some tools require users to install, configure, and start additional applications, such as a web server application or a tool specific server that stores results. These applications typically are also used to interact with results and to navigate the source code containing the weakness. The installation process for these extra components can be more error prone than the tool itself. Because those components are often servers, their installation is more difficult.
7. *Configuring the tool:* Configuring the tool can become one of the most onerous steps in using a tool. Some tools have little documentation, while others have hundreds of pages of documentation; both alternatives are painful. A user really wants a simple 'quick start' guide, such as found in Clang.

   Some tools support plug-ins that enhance the tool with additional functionality. The user must discover, acquire, install, and configure the plug-ins. Sometimes the plug-ins greatly add to the capabilities that a user may be interested in. For example, when the Find Security Bugs plug-in is used with FindBugs, it increased the number of security checks by five times, and Comm-C-Tool-1 also has a proprietary security plug-in.

   A key issue in configuring a tool is choosing the build configuration (this determines which files the tool will scan), and tool options (this determines which weaknesses the tool will report), so that the user receives a reasonable selection of results. With some tools, it is easy to carelessly set tool options that report too little or overwhelmingly large amounts of output. If the tool is configured to produce too little output (either too few types of checks or depth of analysis), the reported results are likely to be incomplete and perhaps useless. In that case, the user will have a false illusion of security once the few weaknesses that were reported are fixed. Likewise, selecting too aggressive of reporting will bury the user in so much output that the results can be useless, even for tools that try to prioritize the severity of their results. Inexperienced programmers are strongly tempted to run tools in a naive way, with every possible tool option set, and will therefore obtain excessive results. This configuration problem is exacerbated when a tool is applied for the first time to a legacy code base. Our experiences with legacy code is that it is easy, with almost any tool, to set options that will provide a weakness count comparable with the lines of code of the application itself.

   It is worth noting that several tools, such as Cppcheck, PMD, and Comm-Java-Tool-1, do not have default settings for performing a scan, therefore the user is forced to determine and specify build configuration and tool options to be able to run even an initial assessment. For example, FindBugs need to know the CLASSPATH to be able to include libraries when analyzing. If those libraries are not analyzed, the output provided by the tool will be incomplete, imprecise, or most likely both.

8. *Prepare the code to be analyzed:* For a tool to be able to perform an analysis, it needs the application's code to be prepared in a specific way, such as converting it to a tool-specific intermediate form, as is the case for both COMM-C-TOOL-1 and COMM-C-TOOL-2. Some tools use a huge amount of disk space for this intermediate form; for example, COMM-C-TOOL-2 uses 2.5 MB of disk, for a 50 line C++ program.

9. *Perform the analysis:* The tool needs to be told which files to scan and about which compiler options to be aware. If a tool ignores compile options such as include file directories and conditional compilation directives such as `ifdef` directives, then the results can be inaccurate and incomplete.

We classify tools in three categories depending on how they are used:

1. **Tools unaware of the build process:** These tools scan code of an individual file or directory and do not know anything about the process for building the program. They are unaware of compilation options, directories for included files, or the existence of dynamically generated files (such as for RPC stubs or parser generators). These tools have inherent deficiencies given they work based on the code they find; therefore, their results are inherently incomplete. For example, FindBugs with the find security bugs plugin will not have information about the libraries used during the build when the user does not specify the CLASSPATH. For languages that are interpreted such as scripts, this may be effective.
2. **Tools that model the compilation command but do not monitor the build process:** These tools are invoked with the full set of compilation options for building the program. The user is required to know the build structure so as to properly invoke the tools. Given the myriad of multi-level complex build methods, figuring out how to modify the build files to make such a change varies from quick and easy to ridiculously complicated. An example of this kind of tool is Clang.
3. **Tools that monitor the build process:** Sophisticated commercial tools monitor the build process and allow the tool to be applied with the right build configuration settings without user intervention. Example of this kind of tools are Coverity, Goanna, and CodeSonar.

An additional complication is that some tools like Cppcheck, FindBugs, and PMD fall in both Categories 1 and 2, depending on if the user supplies build configuration options when they are invoked. Such tools can fool a user into thinking that they obtain reliable results.

10. *View the results:* At this point, the user can navigate through the results reported by the tool. It is especially useful for a user to relate the weaknesses reported by the tool with their occurence in the source code.

Different tools report results in different formats and forms. Some tools report the weaknesses in text mode, and some others offer a graphic interface, while some other interacts with an external component (such as a web server or a tool supplied product) to navigate through the reported results.

For the same application program, different tools often report different weaknesses depending on the focus and capabilities of the tool. So, it is common that software teams will need to use more than one tool on their software to get a complete view of weaknesses that can be found. Dealing with multiple result files, and tool specific result viewers, is difficult to manage and use. Ideally, users should be able to view the merged union of the results from all tools, view the weaknesses in context of the source code, and see other vulnerabilities nearby.

11. *Update the tool:* When a new version of the tool appears, the aforementioned process has to be repeated, although the user's familiarity with the tool should mean that some of the steps will take less time. The key issue is that it is up to the user to periodically check if there is a new version of a tool they are using and act accordingly.

2.2. *Quantitative results on configuring tools*

To understand the effect of using a tool with naive configuration or option settings, we compared the results given by three tools, Cppcheck, Comm-Java-Tool-1, and PMD, when run with the naive settings versus settings selected by experienced analysts. In the case of this study, the settings that we used for the experienced analyst come from those used in the SWAMP (Section 3).

For the first example, we compared running the Cppcheck tool naively with no options against running it using the same pre-processor options used when compiling each source file during a build. Without the proper settings, Cppcheck does not understand what code in the source files to excluded or included in the compilation process. Table I shows the number of weaknesses resulting when scanning several common software packages, lighttpd-1.4.33, lua-5.2.2, nagios-4.0.0, and openssl-1.0.1 with Cppcheck. The first row shows the number of weaknesses reported by the tool without supplying any options (naive), while the second row contains the number of weaknesses when running Cppcheck in the SWAMP, with the right build configuration options. The result of running Cppcheck in the naive way is that it missed significant weaknesses in the application code.

The second example is similar to the first; we ran the Comm-Java-Tool-1 on 38 different Java software packages, including Hadoop-1.1.2, Twitter4J-3.0.3, and Yazd-1.0. In the naive mode (without any build configuration information), the tool reported between 0% and 100% the number of weaknesses reported when the tool ran in the SWAMP mode, which had the right build configuration. The distribution of those percentages is surprisingly very uniform, and the average percentage of weaknesses reported by the naive configuration was 34% the number of vulnerabilities reported by the SWAMP configuration. Again, we see that the tool run in the naive configuration reports an almost unpredictable number of the interesting weaknesses in the software packages.

The third example tests the opposite situation where a tool has no default settings and requires the user to make *some* choices about which options to set. Commonly, in this case, a user will turn on the maximum information setting because they have little concept about which options might not be useful. Of course, the result is often an excessive amount of output from the tool, causing users to have a difficult time knowing which weakness reports to evaluate and which to ignore. In the test, we ran PMD with all options set (the *naive* settings) on 40 different Java software components and compared that to running in the SWAMP, where the options were set by the staff analysts. Table II reports the results for three of the software packages. For the 40 packages, the naive mode reported between two and seven times the number of weaknesses reported when the tool ran in the SWAMP mode. Nevertheless, for 90% of the software packages the number of weaknesses reported by the naive mode of execution was between two and three times the number of vulnerabilities reported by the SWAMP mode. On average, the naive configuration reported three times the number of weaknesses reported by the SWAMP configuration. Even with the SWAMP settings, the number of weaknesses is large; however, the naive setting of options clearly adds significant noise to the results. The SWAMP runs have eliminated the reporting of the more stylistic complaints about the codes, such as missing comments, size of the comments, variable names being too short or too long, naming conventions for variables, methods and classes; and local variables and method arguments that can be declared as read-only for optimization purposes.

Finding the right build configuration and tool options requires an experienced analyst to follow a fine (and slow) iterative tuning process for every tool.

Table I. Number of weaknesses found by Cppcheck in both the naive and SWAMP build configurations.

	lighttdp	lua	nagios	openssl
Default configuration	2	0	23	11
SWAMP configuration	101	16	633	817

The default tool configuration sets no options; the SWAMP automatically sets the tool options to match those given to the compiler during the build process.

Table II. Number of weaknesses for PMD in both the naive (with all options set) and SWAMP configuration options (with only selected options set).

	hadoop	twitter4j	yazd
Naive configuration	116,947	20,499	10,336
SWAMP configuration	43,848	6024	4223

3. THE SOFTWARE ASSURANCE MARKETPLACE

The SWAMP is a web-based facility that automates assessing software with software assurance tools. This section describes the SWAMP, how it eliminates most of the barriers to tool use described in Section 2, and how it provides crucial economies of scale.

Users of the SWAMP upload their software to the SWAMP, and the SWAMP applies the software analysis tools without further interaction. The SWAMP currently supports static analysis tools that analyze C, C++, Java, Python, and Ruby running on Linux platforms. When analysis completes, users can review the merged results of multiple static analysis tools together in the result viewer.

Figure 1 shows the workflow of the SWAMP, including how the assessment is performed in VMs and how the tool results can be viewed in unified form. The details of this figure are described in the rest of this section.

3.1. Managing the tools

The analysis tools are managed by the SWAMP staff reducing barriers to tool use described in items 1–4, 6, and 11 from Section 2. The SWAMP staff identifies the tools, downloads them, installs them on each platform, manages the licenses, and periodically updates the tools as new versions become available.

An important aspect of the SWAMP is that software packages are independent of analysis tools, so the user does not have to modify their packages to work in the SWAMP. Similarly, tools are independent of the software packages, the tools are not modified by the SWAMP staff to be used with a particular software package. This has the implication that once a software package is entered into the SWAMP, it can be assessed by any of the applicable tools that are available; tools added

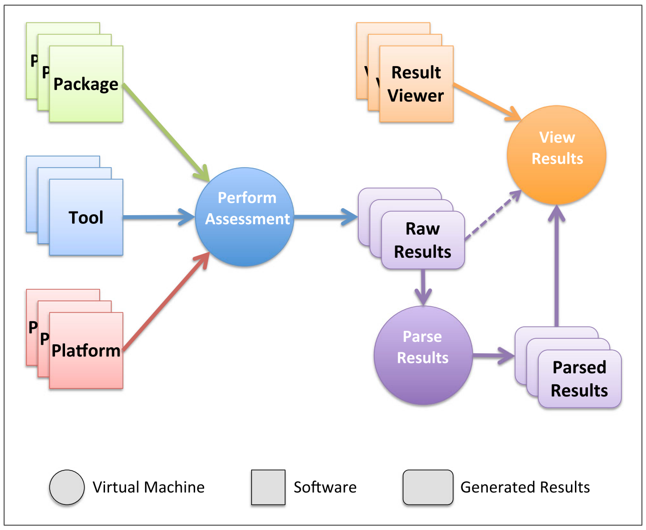


Figure 1. The Software Assurance Marketplace architecture. Shows workflow to assess a software package with a tool, producing raw results that are converted to parsed results, and finally accessible to the user from a web-based viewer. Work is carried out within virtual machines to provide a secure, isolated environment.

in the future will also be able to be applied to software packages in the same way. Similarly, new software packages can be added without changes to the existing tools.

As a result, a user who wishes to assess M software packages with N static analysis tools avoids $M \times N$ effort to configure each of the M packages with N tools. In the SWAMP, the user does a small amount of work for each of the M software packages. As shown later, the information required for the software package configuration for use in the SWAMP is generally simpler than setting up each software package to be assessed with a static analysis tool, reducing the burden even further.

3.2. Structure of the Software Assurance Marketplace

The major components of the SWAMP are a web-based front-end that allows a user to interact with the SWAMP facility, a data store, and a back-end that manages the assessments requested by the user.

The SWAMP front-end consists of web servers that interact with the user's web client to perform tasks within the SWAMP. The data from these interactions and data from running back-end tasks are persistently stored in a file system and an SQL database.

The SWAMP back-end consists of a large number of compute nodes and the software to manage the workload on the compute nodes generated by the front end. The management includes selection of a compute node with available capacity, starting an assessment, monitoring for completion or errors, and storing the output generated in the data store.

Each assessment is performed within a single-use virtual machine (VM). The VM is configured with an input and output disk image. The input disk image is provisioned with the user's selected package and static analysis tool, and the SWAMP tool application framework (the software that drives the assessment). When the tool completes, the assessment results are retrieved from the VM and placed in the output disk image. Each VM is used once and then discarded, and the network routes for each VM prevent it from accessing the SWAMP infrastructure or other VMs. This structure provides for strong isolation, so performing the assessment can not affect future or concurrent assessments, nor can the assessment attack the SWAMP infrastructure. As a result, the SWAMP can safely run untrusted code provided by users or untrusted static analysis tools. The VMs are run on a large collection of compute nodes that are managed by the HTCondor high-throughput scheduling system [39] to reliably execute each assessment.

3.3. Interacting with the Software Assurance Marketplace

1. *Add a software package:* To add a software package to the SWAMP, the user needs to provide the SWAMP with a package archive (the files that make up the software package) and the package configuration (the information necessary for the tool application frameworks to operate, and the meta information to name and describe the software packages such as the name and version). If the user is familiar with building the software package, the package configuration information required should be easily provided as it is essentially the information required to build the software package.

   The package archive can be provided to the SWAMP using two methods: upload an archive or provide a URL for a publicly accessible Git repository. The SWAMP supports most common archive formats, including zip and tar, and common compression formats. For most tools available in the SWAMP, the archive must contain the source code for the software package or, for some Java tools, the bytecode in addition or instead of the source code.

   After the files of the software package are loaded into the SWAMP, the SWAMP inspects them and uses heuristics to select the likely values for the package configuration. The user is allowed to override these values as needed, but for packages that follow common conventions, the package options are typically correct and should work without modification.

   The package configuration describes the two major steps that mimic how software is built: the optional configuration of the software, and the required building of the software.

   The reason for this separation is that the configuration step usually sets up a software package to be built for a specific host environment. The set-up is carried out by creating files and executables that are only used to probe the system for features and are not used during the actual building of the software. Separating the configuration step from the build eliminates assessing auxiliary code that is not directly part of the software package, thereby eliminating results that are not of interest to the user.

For C/C++ packages, the build command can be an arbitrary shell command; for Java, the build uses one of the common build systems such as Ant, Maven, or Gradle; for Python, there is either no build (for simple interpreted programs), or the Python Distutils standard for building and installing Python applications is used.

Below is the complete list of fields of the package configuration with a short description. Other than *build-sys* and usually *package-dir*, the remaining fields are optional and can be heuristically determined by the SWAMP:

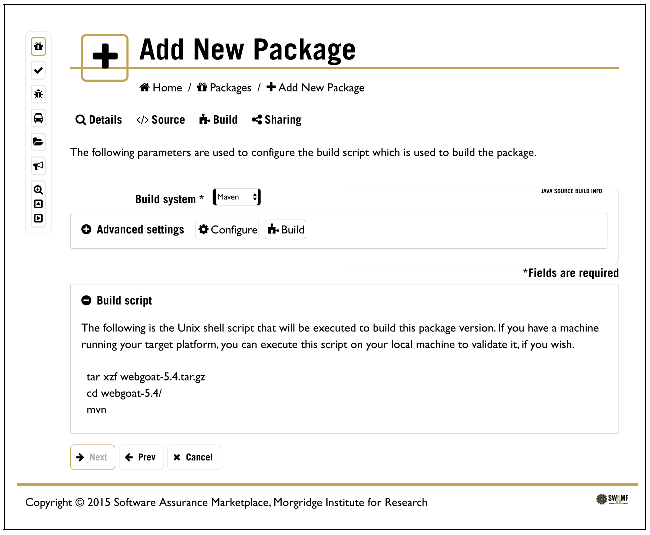
- *build-sys* – the type of the build system;
- *package-dir* – the initial directory in which to begin the assessment;
- *package-language* – for Python assessments, the language dialect (Python 2.x or 3.x);
- *config-cmd* – the configuration command;
- *config-dir* – the directory in which to perform the configuration step;
- *config-opt* – options to pass to the configuration command;
- *build-dir* – the directory in which to perform the build step;
- *build-file* – the build system's configuration file such as 'Makefile' or 'build.xml';
- *build-opt* – options to pass to the configuration command;
- *build-target* – the target used by the build system, such as 'all' or 'release';
- *android-redo-build* – remove the 'build.xml' to create a clean instance;
- *android-sdk-target* – the Android SDK version with which to configure the software package.

No other information is required for a software package to be assessed in the SWAMP.

Based on the value of the fields, the SWAMP displays the commands that correspond to the actions that will be taken in the SWAMP.

The *build-sys* field describes the build system that the software package uses, such as make, ant, maven, or python-distutils. It can also describe common build idioms for C/C++-based software packages, such as running `./configure` or '`cmake .`' followed by `make`. In these cases, it also sets defaults for the *config-cmd*.

Figure 2 shows an example of adding a software package using the SWAMP web interface.



The following parameters are used to configure the build script which is used to build the package.

Build system * Maven

*Fields are required

Build script

The following is the Unix shell script that will be executed to build this package version. If you have a machine running your target platform, you can execute this script on your local machine to validate it, if you wish.

```
tar xzf webgoat-5.4.tar.gz
cd webgoat-5.4/
mvn
```

Copyright © 2015 Software Assurance Marketplace, Morgridge Institute for Research

Figure 2. Software Assurance Marketplace web interface to add a software package. Shows adding the webgoat-5.4 package that uses the Maven build system. The *Build script* section shows the commands used to build the software.

2. *Perform an assessment:* Performing an assessment in the SWAMP simply requires a user to select a software package and an appropriate analysis tool. When assessing a C/C++ software packages, the user is additionally allowed to select a platform (a specific version of the operating system) on which to perform the assessment. Java and Python assessments have fixed platforms as their interpreted environments vary little based on the operating system.

   The user can configure an assessment of a package with one or more of the analysis tools in the SWAMP and, for C/C++, on one or more of the platforms available.

   The assessment can be performed immediately or scheduled to run at a future time.
3. *Manage assessments:* Once the assessment is scheduled, the progress can be monitored on the results page that displays all the user's assessments and their current status. From the dashboard-like display, the user can determine if the assessment has started, is in-progress, has failed, or has completed. From this display, the user can also cancel and delete assessments. For completed results, the user can request to view the results (Section 3.5).
4. *Control access:* Access to information in the SWAMP, including the software packages and results from accessing them, is under the control of the user. The default behavior does not allow access to other users.

   Assessments are carried out in the context of a *project*, and a user can invite other SWAMP users into a project. Software packages and results can be associated with multiple projects, and all member of a project are allowed to use and view the software package or results.

   A user can revoke access to software packages and results to projects at any time. Also, users can remove themselves from a project, and a project owner can remove a user from their project.

3.4. *Applying tools*

A critical part of the SWAMP infrastructure is the software that automates the application of analysis tools to software packages. This automation reduces the barriers to tool use described in Steps 5–9 in Section 2. The SWAMP configures the environment to support the tool, installs any additional software required to use the tool, configures the tool, prepares the code to be analyzed, and finally performs the analysis.

The SWAMP provides infrastructure to monitor the build process as part of the tool application frameworks and uses the results of the monitoring to provide all tools with the information needed to accurately invoke a tool on a specific piece of software. With this infrastructure, the SWAMP can apply any static analysis tool with the most complete and accurate build information, making it possible to have the tool interpret the code in the same way as the compilers that build the software. This infrastructures ensures that assessment results are as closely based on the software that is built as possible. This SWAMP build monitoring software is available as open source to the tool development community.

The SWAMP builds an application under supervision of the monitoring infrastructure, extracting information such as commands that were executed during the build, their environment and current directory. The commands that were executed are further analyzed to determine the files created by the build along with parameters from compilation, library creation, and linking commands. From this information, the SWAMP produces the directives to describe how to apply the analysis tools to the software package.

A brief description of the build monitoring is as follows:

1. *C/C++: monitoring system calls:* For software that has a compilation step other than Java, the kernel system call interface is monitored to determine the programs that are started, their arguments, their environment, and the current working directory.
2. *Java: monitoring the build systems:* For Java software, the common Java build software, Ant, Maven, and Gradle are instrumented. The same information is collected but more efficiently determined than monitoring system calls.
3. *Scripting languages: inspecting the file system:* Scripting languages, being interpreted, do not have a build step but may have a configuration step that creates files that the software needs to run.

3.5. Handling results

The SWAMP provides result viewers that can merge the results from multiple tools into a unified set of results and viewed using a common display. This reduces the barrier to tool use described in Step 10 in Section 2.

Running a tool produces raw tool result files. The SWAMP has developed a result parser for each of the tools supported in the SWAMP, which parses the results and then creates a single result file in a common format called SWAMP common assessment result format (SCARF). Fields with common meanings between tool's raw results, such as type of weakness, or the weakness location (file and line), are mapped to the same field in SCARF. The result parser does not interpret every tool specific field but includes them so further normalization tools are easier to write as they only have to interpret the SCARF XML format instead of each tool's raw output.

The SWAMP currently has two mechanisms to view results: the Native viewer and secure decisions' code Dx™ [40].

The Native viewer can display the results from an assessment of a single software package with a single analysis tool on a single platform. It shows the location of the weakness in the source code, the type of the weakness, its severity, and a description of the weakness.

Secure Decisions' Code Dx has more functionality. Code Dx allows the results from multiple tools applied to the same software package to be displayed concurrently in the same display. The user can filter, sort, and visualize the results based on criteria such as the tool that reported the problem, the location in the source code, or the type of the weakness. These features allow users to focus on the weaknesses of most interest and eliminate those of least. The user can view the source code where the problem is reported with code Dx and triage the problems.

3.6. Future functionality

In the future, the SWAMP will expand across several dimensions: languages, platforms, and other types of analyses. New languages that are planned to be supported include PHP, JavaScript, HTML, CSS, and XML. New platforms planned to be support include MacOS, iOS, and Windows. Analyses with dynamic assessment tools will also be possible with more input from users.

4. CONCLUSION

The challenges in using software assurance tools are many, and the use of these tools is critical as a first line of defense in assessing the security of software. Even in its early years, the SWAMP is able to automate and simplify many of the key tasks that are required to apply software assurance tools to software packages. The SWAMP's high degree of automation, strong security, ability to apply multiple tools easily, and ability to view the results in a unified way are critical to reducing the barriers to tool adoption, and many of the SWAMP features that enable automation also simplify the task of the tool writer, allowing more tools to be effectively applied to more software.

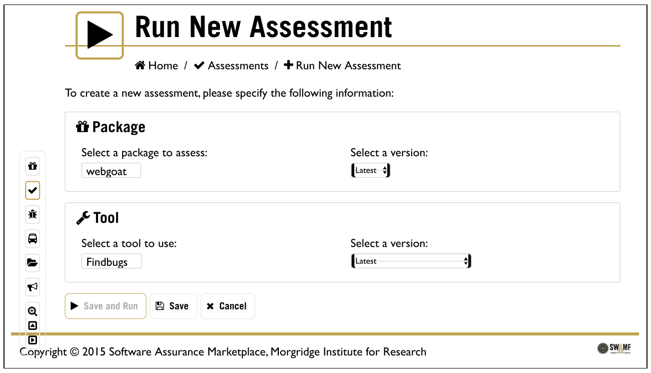
APPENDIX A: USING THE SOFTWARE ASSURANCE MARKETPLACE

The Software Assurance Marketplace (SWAMP) is an open and operational service that allows users to assess their software with a wide variety of assurance tools. Users can freely sign up for an account and assess their software by visiting the SWAMP web site:

https://continuousassurance.org/

This appendix broadly covers the following features of the SWAMP:

- Adding a software package to the SWAMP.
- Running an *assessment* with multiple software assurance (SwA) tools on one of more OS platforms.
- Viewing *assessment results* in the Code Dx viewer.



Run New Assessment

Home / Assessments / Run New Assessment

To create a new assessment, please specify the following information:

Package

Select a package to assess: webgoat

Select a version: Latest

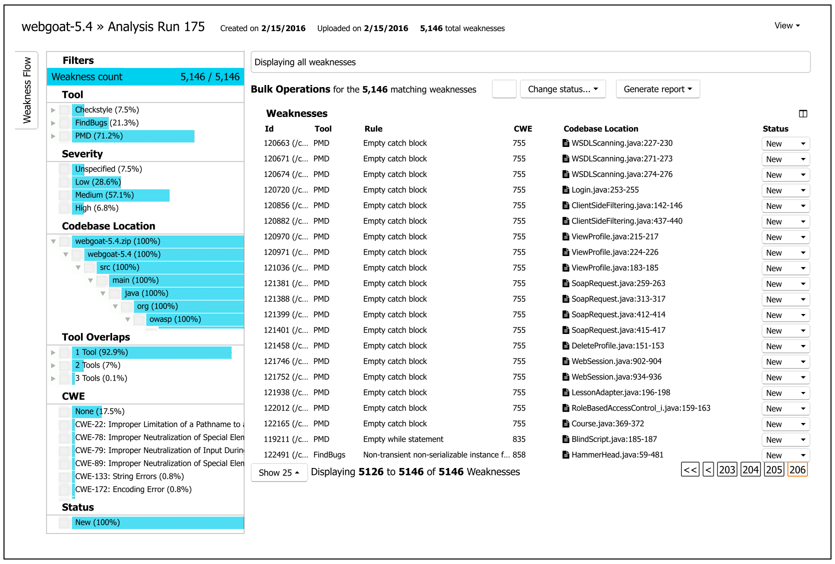
Tool

Select a tool to use: Findbugs

Select a version: Latest

Copyright © 2015 Software Assurance Marketplace, Morgridge Institute for Research

Figure A.1. Running an assessment of the webgoat package with Findbugs in the Software Assurance Marketplace.



webgoat-5.4 » Analysis Run 175 Created on **2/15/2016** Uploaded on **2/15/2016** **5,146** total weaknesses

Filters

Weakness count 5,146 / 5,146

Tool
- Checkstyle (7.5%)
- FindBugs (21.3%)
- PMD (71.2%)

Severity
- Unspecified (7.5%)
- Low (28.6%)
- Medium (57.1%)
- High (6.8%)

Codebase Location
- webgoat-5.4.zip (100%)
  - webgoat-5.4 (100%)
    - src (100%)
      - main (100%)
        - java (100%)
          - org (100%)
            - owasp (100%)

Tool Overlaps
- 1 Tool (92.9%)
- 2 Tools (7%)
- 3 Tools (0.1%)

CWE
- None (17.5%)
- CWE-22: Improper Limitation of a Pathname to
- CWE-78: Improper Neutralization of Special Elen
- CWE-79: Improper Neutralization of Input Durin
- CWE-89: Improper Neutralization of Special Elen
- CWE-133: String Errors (0.8%)
- CWE-172: Encoding Error (0.8%)

Status
- New (100%)

Displaying all weaknesses

Bulk Operations for the **5,146** matching weaknesses

Weaknesses

Id	Tool	Rule	CWE	Codebase Location	Status
120663 (/c...	PMD	Empty catch block	755	WSDLScanning.java:227-230	New
120671 (/c...	PMD	Empty catch block	755	WSDLScanning.java:271-273	New
120674 (/c...	PMD	Empty catch block	755	WSDLScanning.java:274-276	New
120720 (/c...	PMD	Empty catch block	755	Login.java:253-255	New
120856 (/c...	PMD	Empty catch block	755	ClientSideFiltering.java:142-146	New
120882 (/c...	PMD	Empty catch block	755	ClientSideFiltering.java:437-440	New
120970 (/c...	PMD	Empty catch block	755	ViewProfile.java:215-217	New
120971 (/c...	PMD	Empty catch block	755	ViewProfile.java:224-226	New
121036 (/c...	PMD	Empty catch block	755	ViewProfile.java:183-185	New
121381 (/c...	PMD	Empty catch block	755	SoapRequest.java:259-263	New
121388 (/c...	PMD	Empty catch block	755	SoapRequest.java:313-317	New
121399 (/c...	PMD	Empty catch block	755	SoapRequest.java:412-414	New
121401 (/c...	PMD	Empty catch block	755	SoapRequest.java:415-417	New
121458 (/c...	PMD	Empty catch block	755	DeleteProfile.java:151-153	New
121746 (/c...	PMD	Empty catch block	755	WebSession.java:902-904	New
121752 (/c...	PMD	Empty catch block	755	WebSession.java:934-936	New
121938 (/c...	PMD	Empty catch block	755	LessonAdapter.java:196-198	New
122012 (/c...	PMD	Empty catch block	755	RoleBasedAccessControl_i.java:159-163	New
122165 (/c...	PMD	Empty catch block	755	Course.java:369-372	New
119211 (/c...	PMD	Empty while statement	835	BlindScript.java:185-187	New
122491 (/c...	FindBugs	Non-transient non-serializable instance f...	858	HammerHead.java:59-481	New

Show 25 Displaying **5126** to **5146** of **5146** Weaknesses

Figure A.2. Viewing assessment results of webgoat with multiple tools in Code Dx.

A.1 Adding a package to the Software Assurance Marketplace

Adding a package to the SWAMP involves the user specifying a public *GitHub* repository, or uploading an archive such as *zip* or a *tar* file. The SWAMP inspects the software package, and using heuristics determines the primary language, and how to build the software. The user is given the opportunity to modify these selections. Figure 2 is the screen shot of *Add New Package* page.

A.2 Running an assessment

Running an assessment is the process of running a SwA tool on a software package. To run an assessment, a user just selects a package and an SwA tool they wish to use (and additionally the operating system platform for C and C++ packages). Figure A.1 is the screen shot of *Run New Assessment* page.

A.3 Viewing assessment results

To view the results of an assessment, the user selects one or more of their assessments that has successfully completed and the viewer to use: the rudimentary *native* viewer, or *Code Dx*. Code Dx is more sophisticated and allows simultaneous display of the results from multiple assessments, filtering, sorting, visualization, and triaging of results. Figure A.2 is a screen shot of the Code Dx viewer.

ACKNOWLEDGEMENTS

This work was supported in part by the US National Science Foundation Cyber Security Center of Excellence grant ACI-1547272, US Department of Homeland Security under AFRL Contract FA8750-12-2-0289, and MINECO Spain contract number TIN2014-53172-P.

REFERENCES

1. Bessey A, Block K, Chelf B, Chou A, Fulton B, Hallem S, Henri-Gros C, Kamsky A, McPeak S, Engler D. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM* February 2010; **53**(2):66–75. DOI:10.1145/1646353.1646374.
2. Coverity. *SAVE Coverity*. Available from: http://www.coverity.com/products/coverity-save/ [last accessed November 2015].
3. Packard H. *HP Fortify Static Code Analyzer*. Available from: http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/ [last accessed November 2015].
4. Roque Wave. *Klockwork*. Available from: http://www.klocwork.com/products-services/klocwork [last accessed November 2015].
5. Red Lizards. *Red Lizards Goanna*. Available from: http://redlizards.com/ [last accessed November 2015].
6. Grammatech. *GRAMMATECH CodeSonar*. Available from: http://www.grammatech.com/codesonar [last accessed November 2015].
7. *Embedded Software Design: Best Practices for Static Analysis Tools*. Available from: http://www.grammatech.com/images/pdf/embedded-software-design-whitepaper.pdf [last accessed November 2015].
8. IBM *IBM® Security AppScan® Source*. Available from: http://www-03.ibm.com/software/products/en/appscan-source [last accessed November 2015].
9. Parasoft *Parasoft C/C++test*. Available from: http://www.parasoft.com/product/cpptest/ [last accessed November 2015].
10. Parasoft *Parasoft Jtest*. Available from: http://www.parasoft.com/product/jtest/ [last accessed November 2015].
11. *Clang Static Analyzer*. Available from: http://clang-analyzer.llvm.org/ [last accessed November 2015].
12. *Cppcheck*. Available from: http://cppcheck.sourceforge.net/ [last accessed November 2015].
13. *OCLint*. Available from: http://oclint.org/ [last accessed November 2015].
14. Hovemeyer D, Pugh W. Finding Bugs is Easy. *SIGPLAN Notices* December 2004; **39**(12):92–106. DOI: 10.1145/1052883.1052895.
15. *FindBugs*. Available from: http://findbugs.sourceforge.net/ [last accessed November 2015].
16. *Find Security Bugs*. Available from: http://h3xstream.github.io/find-sec-bugs/ [last accessed November 2015].
17. *PMD*. Available from: http://pmd.sourceforge.net/ [last accessed November 2015].
18. *checkstyle*. Available from: http://checkstyle.sourceforge.net/ [last accessed November 2015].
19. Google Inc. *Error Prone*. Available from: http://errorprone.info/ [last accessed November 2015].
20. *Pylint*. Available from: http://checkstyle.sourceforge.net/ [last accessed November 2015].
21. *Bandit*. Available from: https://wiki.openstack.org/wiki/Security/Projects/Bandit [last accessed November 2015].
22. *Flake8*. Available from: https://pypi.python.org/pypi/flake8 [last accessed November 2015].
23. *Dawn*. Available from: https://github.com/thesp0nge/dawnscanner [last accessed November 2015].
24. *Brakeman – Rails Security Scanner*. Available from: http://brakemanscanner.org/ [last accessed November 2015].
25. *PHP_CodeSniffer*: Squiz Labs. Available from: http://www.squizlabs.com/php-codesniffer [last accessed November 2015].
26. *RIPS*. Available from: http://php-security.org/downloads/rips.pdf [last accessed November 2015].
27. *JSLint*. Available from: http://www.jslint.com/ [last accessed November 2015].
28. *Flow, a New Static Type Checker for JavaScript*: Facebook Inc. Available from: https://code.facebook.com/posts/1505962329687926/flow-a-new-static-type-checker-for-javascript/ [last accessed November 2015].
29. *FxCop*: Mircosoft. Available from:https://msdn.microsoft.com/en-us/library/bb429476%28v=vs.80%29.aspx [last accessed November 2015].
30. *Coverity Scan*: Coverity. Available from: https://scan.coverity.com/ [last accessed November 2015].
31. *VERACODE*: VERACODE. Available from: https://www.veracode.com/sites/default/files/Resources/Datasheets/binary-static-analysis-datasheet.pdf [last accessed November 2015].
32. Hewlett Packard. *Fortify On Demand*. Available from: http://www8.hp.com/us/en/software-solutions/application-security-testing/ [last accessed November 2015].
33. Bird J, Kim F. *San Survey on Application Security Programs and Practices*: SANS Institute, 2012. Available from: https://www.sans.org/reading-room/whitepapers/analyst/survey-application-security-programs-practices-35150 [last accessed November 2015].
34. McGraw G, Migues S, West J. *Building Security in Maturity Model (BSIMM)*: Cigital, 2013. Available from: https://www.bsimm.com/ [last accessed November 2015].

35. *Software Assurance Marketplace (SWAMP) web site*: Software Assurance Marketplace. Available from: https://www.continuousassurance.org/ [last accessed November 2015].
36. Shull F, Basili V, Boehm B, Brown AW, Costa P, Lindvall M, Port D, Rus I, Tesoriero R, Zelkowitz M. What We Have Learned About Fighting Defects. *8th IEEE International Symposium on Software Metrics*, Ottawa, Canada, 2002; 249–258.
37. Kupsch JA, Miller BP. Manual vs. Automated Vulnerability Assessment: A Case Study. *The first International Workshop on Managing Insider Security Threats*, Lafayette, IN, 2009; 83–97.
38. Boland T, Black PE. The Juliet 1.1 C/C++ and Java Test Suite. *IEEE Computer* 2012 October; **45**(10):88–90. DOI:10.1109/MC.2012.345.
39. Thain D, Tannenbaum T, Livny M. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience* 2005; **17**:2–4.
40. Secure Decisions. *Code Dx*. Available from: http://codedx.com/ [last accessed November 2015].