THE LINUX FOUNDATION

» The Open Compliance Program

Dependency Checker Tool

Overview and Discussion

By Stew Benedict and Jeff Licquia, The Linux Foundation

A White Paper By The Linux Foundation
http://www.linuxfoundation.org

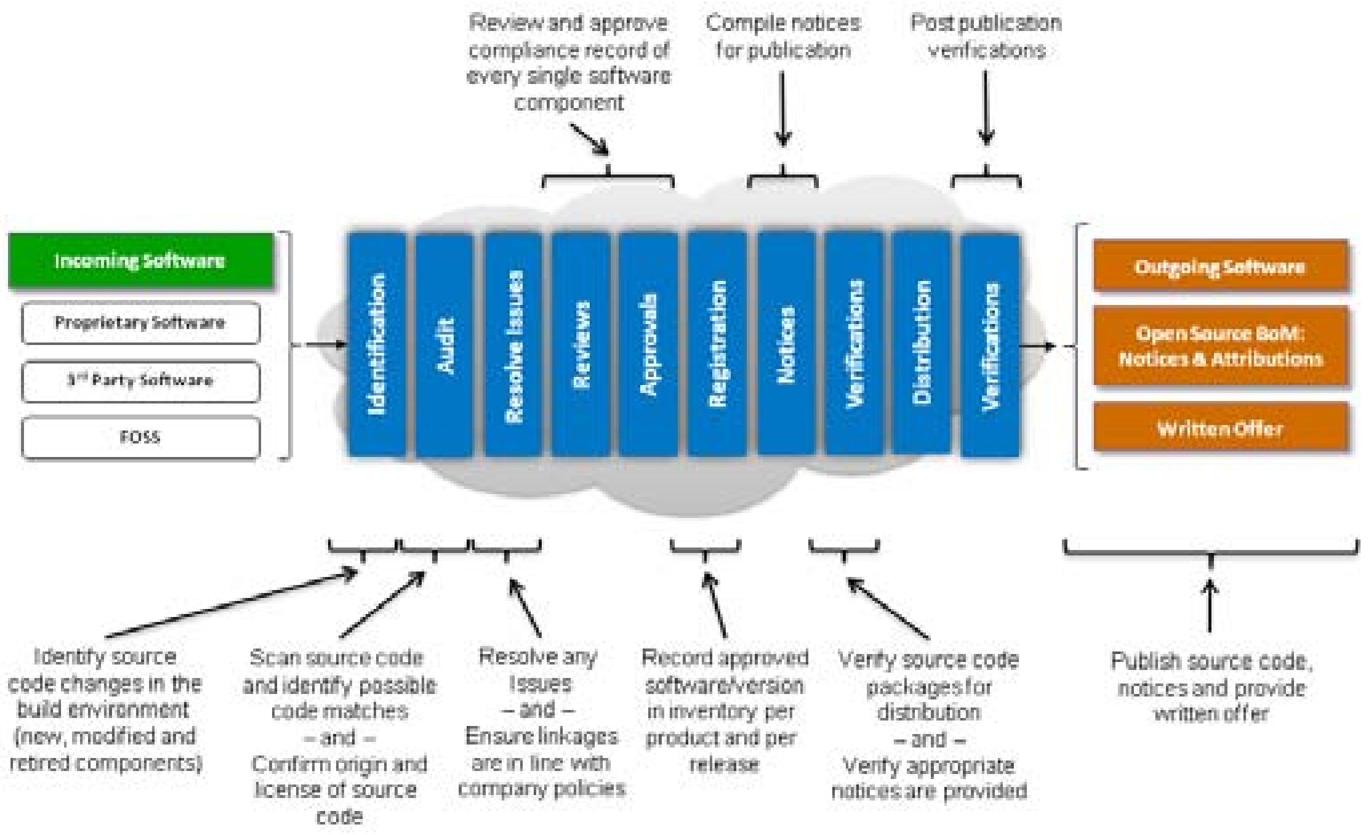
Introduction

In the past two months, the Linux Foundation has published several papers (available from http://www.linuxfoundation.org/publications) on the topic Free and Open Source Software Compliance in preparation of the launch of the Open Compliance Program. This paper is the fourth in the same series and provides an overview of the Dependency Checker Tool, an open source project initiated by the Linux Foundation as part of the Open Compliance Program, aimed at providing a tool to support FOSS compliance due diligence activities.

The paper is divided into two main parts: The first part provides a discussion on the role of tools in ensuring FOSS compliance with a listing of tools used in the compliance end-to-end management process, and the second part is dedicated to the Dependency Checker Tool design and implementation details.

Figure 1 provides a sample end-to-end compliance management process that includes the following steps:

1. Identification: All incoming software is identified and entered into the compliance process
2. Audit: software is scanned and audited and an audit report is generated identifying origins and licenses of source code
3. Resolve Issues: Any pending issues from the audit are worked on and resolved
4. Reviews: compliance ticket is reviewed by the compliance team
5. Approvals: compliance ticket is approved by the compliance team
6. Registration: software component is added to the inventory of approved components for use in a specific product/version
7. Notices: all notices are compiled and sent to documentation team for publishing
8. Verifications: pre-distribution verification steps
9. Distributions: source code is published
10. Verifications: post-distribution verification steps



Review and approve compliance record of every single software component
Compile notices for publication
Post publication verifications
Incoming Software
Proprietary Software
3rd Party Software
FOSS
Identification
Audit
Resolve Issues
Reviews
Approvals
Registration
Notices
Verifications
Distribution
Verifications
Outgoing Software
Open Source BoM: Notices & Attributions
Written Offer
Identify source code changes in the build environment (new, modified and retired components)
Scan source code and identify possible code matches – and – Confirm origin and license of source code
Resolve any issues – and – Ensure linkages are in line with company policies
Record approved software/version in inventory per product and per release
Verify source code packages for distribution – and – Verify appropriate notices are provided
Publish source code, notices and provide written offer

Figure 1: Example of a compliance process

For the purpose of this article, we do not discuss in detail the compliance process. The goal is to illustrate a sample process and discuss which tools are being used in which steps in the process.

Tools and Automation

Tools are an essential building block in a compliance program that can assist companies with their compliance activities as efficiently and accurately as possible. There are a number of tools that can be very useful in a FOSS compliance program:

- Project management tool
- Source code scanning and license identification tool
- Linkage analysis tool
- Bill of material difference tool
- Source code review tool
- Linguistic review tool

Figure 2 illustrates a sample compliance end-to-end process and displays in which phases of the compliance process the tools are being used. In addition, the figures points out the three tools that the Linux Foundation is initiating as open source tools to help companies with their compliance efforts.

In the following sub-sections we describe briefly these tools and how they contribute to the compliance activities. It is worth mentioning that there exist in the market several commercial and open source tools that provide the various functionalities described below.

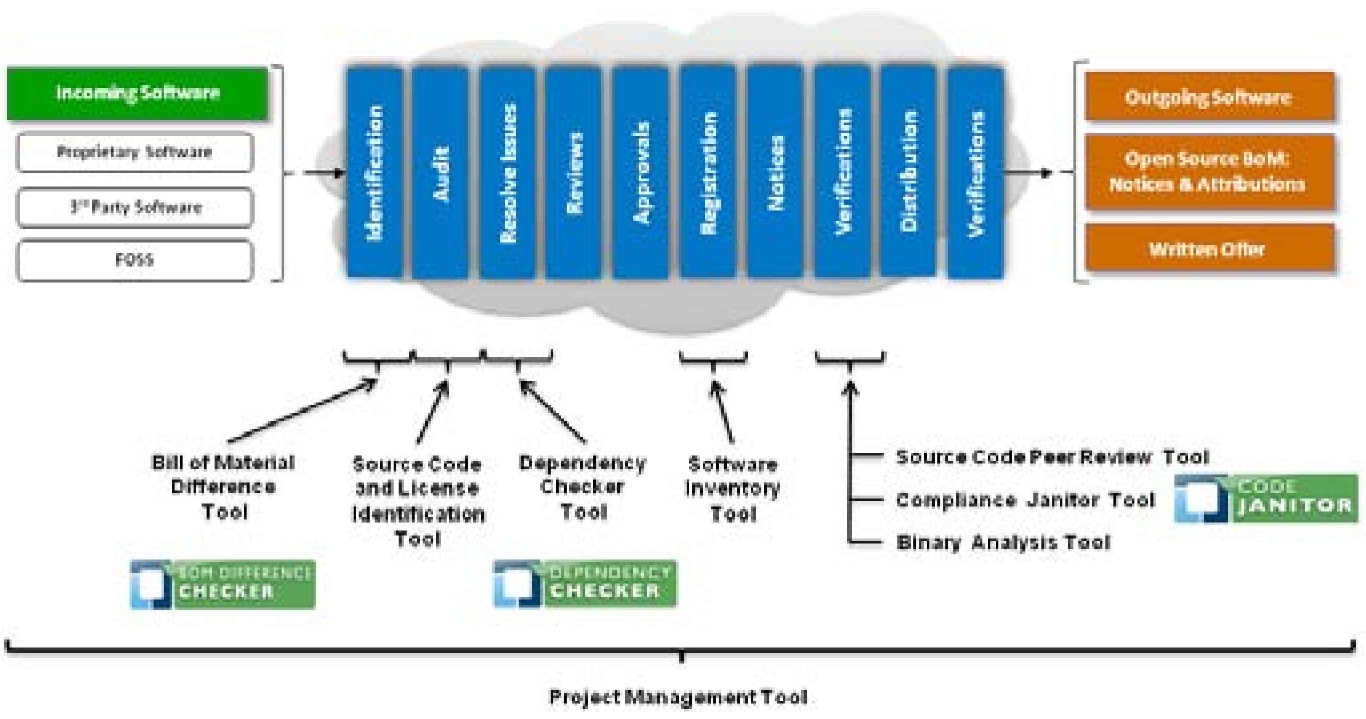


Figure 2: Example of tools used in compliance management

Project Management Tool

A project management tool is essential to manage and track the compliance activities. Some companies use their bug tracking tools that are already in place with a customized compliance workflow; other companies rely on specific project management tools or even in-house solutions. Whatever your preference is, you would want the tool to reflect the workflow of your compliance process, allowing you to move compliance tickets from one phase of the process to another, providing you with task and resources management, time tracking, email notifications, project statistic and reporting capabilities.

Bill of Material (BoM) Difference Checker

The goal of the BoM difference checker is to compute the difference between two BoM and produce a list of changes. This will enable efficient incremental compliance on the BoM delta on two versions of the same product.

The inputs to the BoM difference checker are two BoM files that represent the list of components available on two different version of the product. For instance BoM_v1.0 represents version 1.0 and BoM_v1.1 represents version 1.1. The output of the BoM difference checker would be the list of new components, retired components, and modified components.

Source Code and License Identification Tool

The source code and license identification tool provides recognition and analysis capabilities for both source code and binaries and assists the user of the tool in identifying the origin of the source code and the associated licenses.

Dependency Checker Tool

The goal of the dependency checker is to flag problematic code combinations at the dynamic and static link level. The tool identifies a linkage conflict between binaries and libraries based on predefined license policies that the user of the tool has already defined. The Dependency Checker Tool is discussed in full in the second half of the paper.

Software Inventory Tool

It is critical to keep track of approved software components, the products in which they are deployed, and specific information such as version numbers, licensing and compliance information. Such data is usually stored in a software inventory tool that maintains information about all software components, not just FOSS assets. The software inventory tool provides significant advantages in cases such as compliance audits or corporate mergers and acquisitions.

Source Code Peer Review Tool

As part of the distribution process, some companies elect to review the source code changes they applied to existing FOSS components before making the components available for download from their web site. A source code review tool is very useful in this context to allow developers to review the changes made to the original source code.

Code Janitor Tool

Before releasing source code to the public, companies often perform a linguistic review to make sure that developers did not leave any comments about future products, product code names, mention of competitors, or any inappropriate comments. The Code Janitor tool was initiated by the Linux Foundation as an Open Source project and it allows its users to create a database of inappropriate words (or keywords) the tool will use in a brute force scan of the code. The result is a list of files that contains the offending "keywords." The Code Janitor tool is available from http://git.linuxfoundation.org/?p=janitor.git.

Binary Analysis Tool

The Binary Analysis Tool is a modular framework that assists with auditing the contents of compiled software. It allows the discovery of software components that were used to create compiled code. The tool supports the automated extraction of the version and configuration of BusyBox, extraction of file system information, automated checking for the Linux kernel , and a capability for brute force scanning of firmware. The binary analysis tool is available from http://www.binaryanalysis.org/.

Note

In some cases, a single tool can fulfill more than one of the roles described above.

Dependency Checker Tool

The Dependency Checker Tool is the first tool released by the Linux Foundation (under the MIT license) as a part of the Open Compliance Program to help companies and individuals with their compliance due diligence. Like any other open source project, the project is open (open mailing list, open git repository, open bugzilla) and governed by open source development processes. The tool in its initial release (discussed in a later section) provides a number of features and capabilities that will grow with time. The tool is to be used as a platform to innovate and enable easy compliance. Such tools provide no product differentiation for companies and our hope is that companies, individuals, and open source developers will join the effort and contribute in building solid tools that help everyone using open source software.

Problem Description

There are various legal opinions on how linkage methods (static versus dynamic) may or may not affect your compliance with FOSS licenses. Companies often define policies that govern what is acceptable from a linkage/licensing perspective. For instance, sample policies may look as follows:

Components/binaries licensed under license A can link statically or dynamically to libraries licensed under license B.

Components/binaries licensed under license C can link dynamically only to libraries licensed under license D and E.

Components/binaries licensed under license F can not link statically or dynamically to libraries licensed under license H.

Therefore, there is a need for a tool that offers capabilities to:

1. Discover linkages occurring between a given binary and any given library
2. Discover the type of linkage (static or dynamic)Linkage analysis tool
3. Offer a license framework to allow companies to define their policies
4. Alert users anytime there is a linkage/license conflict discovered by the tool

Goal

The goal of the tool is to flag problematic code combinations at the dynamic and static link level based on predefined linkage/license policies.

Inputs and Outputs

There are 2 major inputs to the tool:

1. Location of the root file system that you want to scan. The tool then crawls the root file system looking for binaries and works to identify to which libraries they link and how.
2. Linkages/license policies that are defined in the tool and correspond to any specific company policies (examples were provided earlier).

For any given binary, the output is the list of libraries it links to, linkage method and a flag that illustrates if the linkage/license combination is:

- (Red flag) Not allowed by the defined policies
- (Orange flag) a new combination of linkage method and license that it not available in the defined policies

Discovering Dynamic Linkages

The software takes either a path, or a path and a filename, as input. The path may be to a single file or to a directory. If the path is a directory, the directory is searched recursively, either for any ELF files or for the additional filename given. We only consider ELF files for analysis.

Once an ELF file is discovered, we find the dynamically linked libraries using an equivalent of the command:

```
readelf -a  file  | grep NEEDED | awk '{print $5}'
```

This gives us a list of the SONAMES of the libraries this binary depends on. If recursion is requested, we run ldd on the original file to get the library paths, and resolve any symbolic links to the actual file. We then repeat the sequence on each library to find its dependencies, and continue to repeat up to the recursion level or until no more recursion is possible.

Discovering Static Linkages

First, for each ELF file found, we gather a list of all interfaces linked into it from any source, using the equivalent of this shell command:

```
readelf -s file  | grep FUNC  | awk '{ print $8 }'
```

Second, we read a list of all interfaces that have debugging information available. This is done using readelf -wi and parsing the output, looking for all entries of type DW_TAG_subprogram and storing the DW_AT_name attribute.

Once we have these two lists, we look for symbol names that appear in the first list but not in the second list. These are functions embedded in the executable, but without debugging information. The linking of static libraries is the most common cause for the presence of symbols like this. To determine the actual library involved, we consult a table that contains all of the symbols in a number of libraries.

Rather than ship an embedded (and possibly stale) table of symbols, we generate the table by reading the symbol tables from the dynamic libraries provided on the system. Initially, the table is empty; we provide both a command-line tool and a button in the user interface (under the "Settings" tab) for analyzing the system and showing fresh data. We show warnings if the table is empty when a scan is done.

This process is not as exact as the process for finding dynamic libraries. See the "Limitations" section below for a discussion of the constraints and reliability of this method.

Using the Command Line Interface (CLI)

The command line program is called readelf.py, and it resides in /opt/ linuxfoundation/bin (or in dep-checker/bin for a git checkout):

```
~/dep-checker/bin$ ./readelf.py --help
Usage: readelf.py [options] <file/dir tree to examine> [recursion depth]
Options:
-c                          output in csv format
-d                          write the output into the results database
-s DIR                      directory tree to search
--comments=COMMENTS         test comments (when writing to database)
--project=PROJECT           project name (when writing to database)
--no-static                 don't look for static dependencies
--version                   show program's version number and exit
-h, --help                  show this help message and exit
```

-c Option

The -c option is primarily used to pass data to the GUI. The format without this argument is more human-readable if you are using the command line directly.

-d Option

The -d option forces -c style output and writes the data to the same database used by the GUI. You can use –project and –comments to fill these fields in the test record. The username is acquired from the environment. Using -d allows unattended test runs via cron or other means that can be later reviewed in the GUI.

-s DIR Option

The -s option expects a directory as an argument. If you specify this option, the program will attempt to drill down through the directory mentioned to find only files with the name specified by the next argument to analyze:

```
/opt/linuxfoundation/bin/readelf.py -s /foo bar
```

The program will search everything under /foo, for ELF files named bar
Specifying only a directory will search and report on every ELF file in that directory tree:

```
/opt/linuxfoundation/bin/readelf.py /foo
```

Specifying only a file will attempt to test only the specified file:

```
/opt/linuxfoundation/bin/readelf.py /foo/bar/baz
```

The recursion level is an optional argument, that will attempt to not only report the direct dependencies, but also report the dependencies of each library used by the target file:

```
/opt/linuxfoundation/bin/readelf.py /foo/bar/baz 4
```

This would attempt to recurse down 4 levels from the target file, giving output something like this:

```
[1]/foo/bar/baz:
 libtermcap.so.2
  [2]/lib/libtermcap.so.2.0.8:
   libc.so.6
    [3]/lib/i686/libc-2.10.1.so:
     ld-linux.so.2
[1]/foo/bar/baz:
 libdl.so.2
  [2]/lib/libdl-2.10.1.so:
  libc.so.6
  [3]/lib/i686/libc-2.10.1.so:
   ld-linux.so.2
[2]/lib/libdl-2.10.1.so:
  ld-linux.so.2
[1]/foo/bar/baz:
 libc.so.6
  [2]/lib/i686/libc-2.10.1.so:
   ld-linux.so.2
```

You will note that even though we asked for a recursion level of 4, the test stopped at level 3, as the program detects when no further recursion is possible.
Static library dependencies will appear with (static) appended to the SONAME:

```
libncurses.so.5 (static)
```

--comments=COMMENTS

The –comments option is used to populate the comments field in the test record when using -d. If the comment is multiple words, it should be quoted:

```
--comments='this is a comment'
```

--project=PROJECT

The –project option is used to populate the project field in the test record when using -d. If the comment is multiple words, it should be quoted:

```
--project='my FOSS project'
```

--no-static Option

The --no-static option suppresses trying to resolve statically linked dependencies.

--version

The –version option reports the version of the command line program.

-h, --help

The -h, --help option returns the usage/options as listed earlier.

Overview of the Graphical User Interface

The GUI interface is simple by design with tabs to access various aspects of program:

• Check Dependencies	Test entry, initiate form
• Review Results	Tabular list of test results
• Licenses	License name/version/alias entry tab
• License Bindings	Define license bindings for targets (test files) and libraries
• License Policies	Define linkage license policies to be flagged during testing
• Settings	View and change settings, load/reload the static database
• Documentation	User documentation

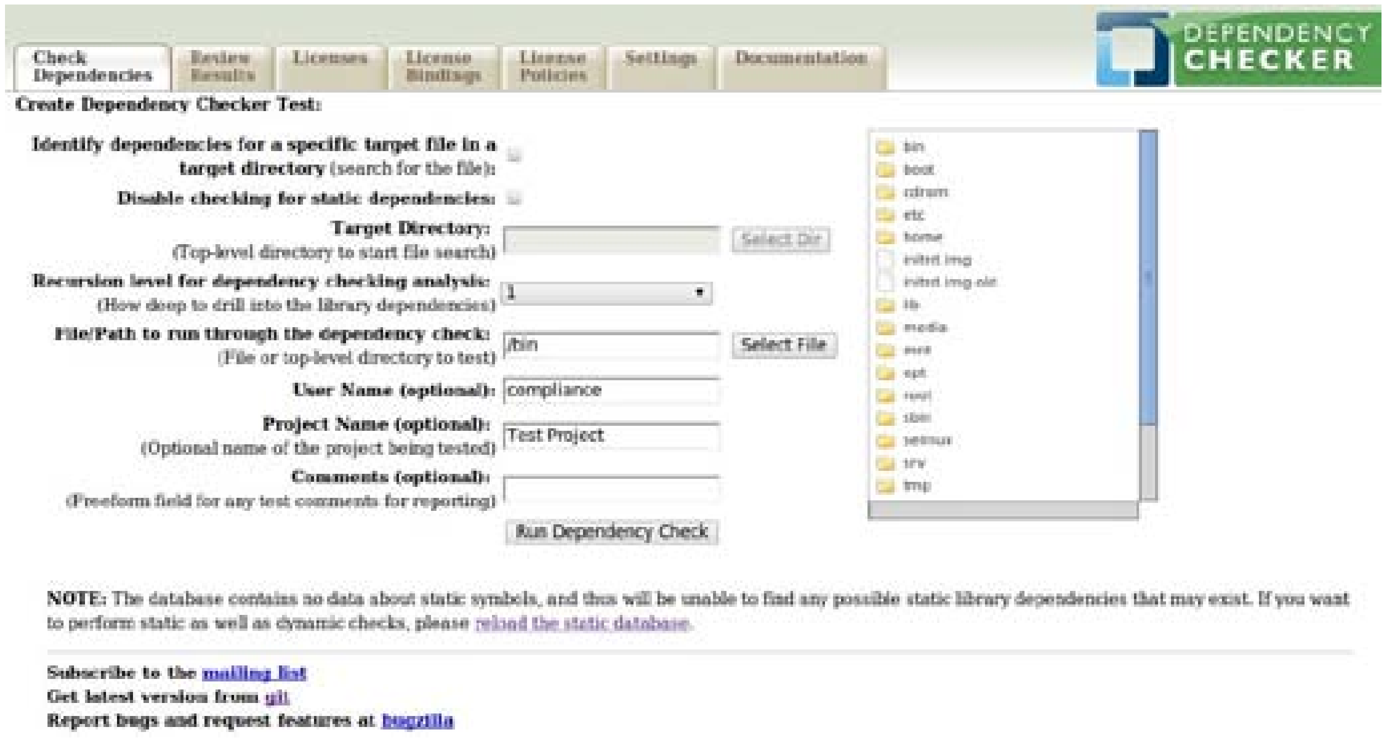
The final page, which isn't visible in the tabs, is the test results detail page, brought up by either running a test, or clicking on the link in the results page.

Check Dependencies

Figure 3 provides an illustration of the home page of the Dependency Checker Tool once launched in a web browser. This is also known as the Check Dependencies tab.
When running the Dependency Checker for the very first time, you should start by loading the static symbol tables. If this has not been done, the page will include a reminder to do this. See below, under the Settings tab, for instructions on how to do this.

Once this has been done, a test sequence would typically start at the Check Dependencies page, where you enter the test criteria. This setup parallels the operation of the command line program, where you select whether to search for a file under a directory, test a whole directory, or just a single file. There is also a drop-down to select the recursion level. You can disable checking for static dependencies via a checkbox.

The user field is pre-populated with the compliance user, but can be overridden. The project and comments fields are optional for your use in tracking tests.
Once you enter the test criteria, click on the Run Dependency Check button. After the test runs you will be presented with the detailed test results in tabular form. Depending on the number of files to be tested and the recursion level, the test can take a few minutes, so be patient. A status box will replace the form on the screen to show the progression of the test.



Create Dependency Checker Test:

Identify dependencies for a specific target file in a target directory (search for the file): ☐

Disable checking for static dependencies: ☐

Target Directory: (Top-level directory to start file search)

Recursion level for dependency checking analysis: (How deep to drill into the library dependencies) 1

File/Path to run through the dependency check: (File or top-level directory to test) /bin

User Name (optional): compliance

Project Name (optional): (Optional name of the project being tested) Test Project

Comments (optional): (Freeform field for any test comments for reporting)

NOTE: The database contains no data about static symbols, and thus will be unable to find any possible static library dependencies that may exist. If you want to perform static as well as dynamic checks, please reload the static database.

Subscribe to the mailing list
Get latest version from git
Report bugs and request features at bugzilla

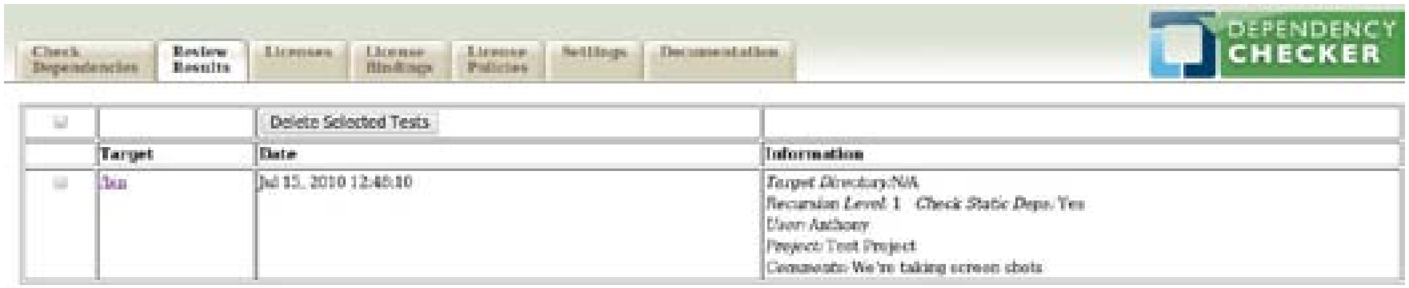
Figure 3: Main screen of Dependency Checker

Until there are licenses and bindings defined, the results detail will show TBD for both the target and dependency licenses. Now that there is data in the system, you can go back and define these relationships and update the test data.

There is a Print Results button on the detail page that should open the browser print dialog to print to a physical printer or to a file. Some parts of the GUI are hidden in the printed output so that only the test results show up in the printed report.

Review Results

Figure 4 illustrates how users of the tool can access the test results via the Review Results page. This is a tabular list of all the test runs, sorted by test id/date. The far-right column has the information entered from the Check Dependencies tab. Clicking on the link for the target file or directory will open the detail tab. If you want to delete test results, you can select the checkboxes and delete them from here, using the Delete Selected Tests button.



Check Dependencies | Review Results | Licenses | License Bindings | License Policies | Settings | Documentation

		Delete Selected Tests	
	Target	Date	Information
	Jsu	Jul 15, 2010 12:40:10	*Target Directory:*N/A *Recursion Level:* 1 *Check Static Deps:* Yes *User:* Anthony *Project:* Test Project *Comments:* We're taking screen shots

Licenses

The License tab lets you enter license names, versions, abbreviations and aliases. For example, the GNU General Public License v 2.0:

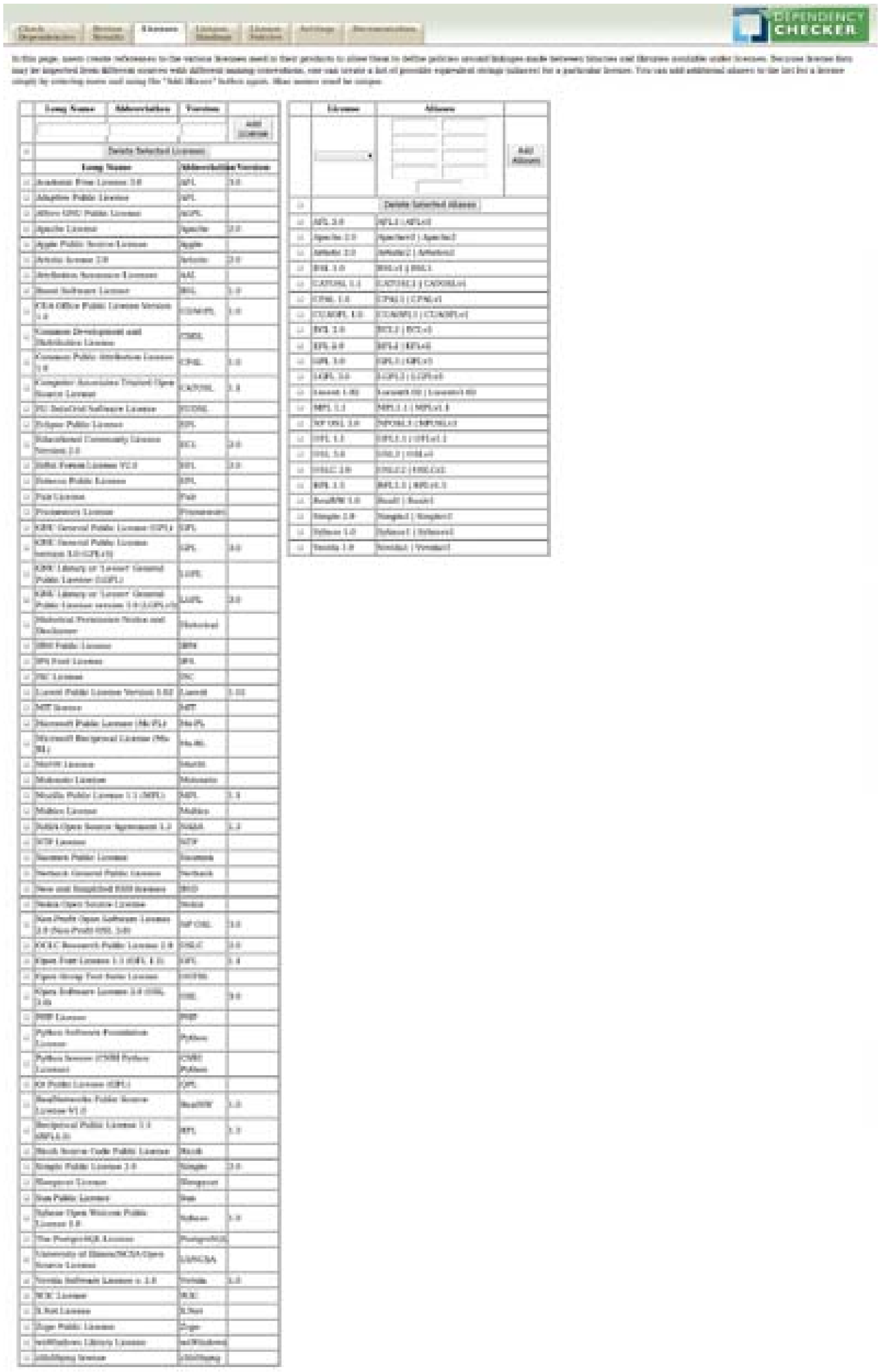
- Long name = GNU General Public License
- Abbreviation = GPL
- Version number = 2.0
- Possible aliases = GPL2.0 | GPL v2 | GPL 2 | GPL v2

You can add as many aliases as you need for any given license.
This information is needed when importing license information from a third party tool. Since we are unaware of how the license name and version will be reported, we have the facility to define various aliases for any given License Name / License Version combination and still link that to the license policies. Following this approach, if we have a policy that states,

Binaries licensed under license A v1.0 can link statically or dynamically to libraries licensed under license B v2.0.

we will still be able to discover and flag the combination even if the license are imported differently than what is stated in the policy. You can select a abbreviation/version from the system and then provide up to 9 alternate names (aliases) that will be considered equivalent when examining test results for policy violations. Additional aliases can be added to an existing list by simply selecting the license, entering just the new alias and clicking "Add" again. Alias entries must be unique, and you'll be warned when trying to add an alias that already exists.

Figure 5 is an illustration of the License tab and shows the default open source licenses (OSI approved) licenses that are pre-loaded by default in the tool.



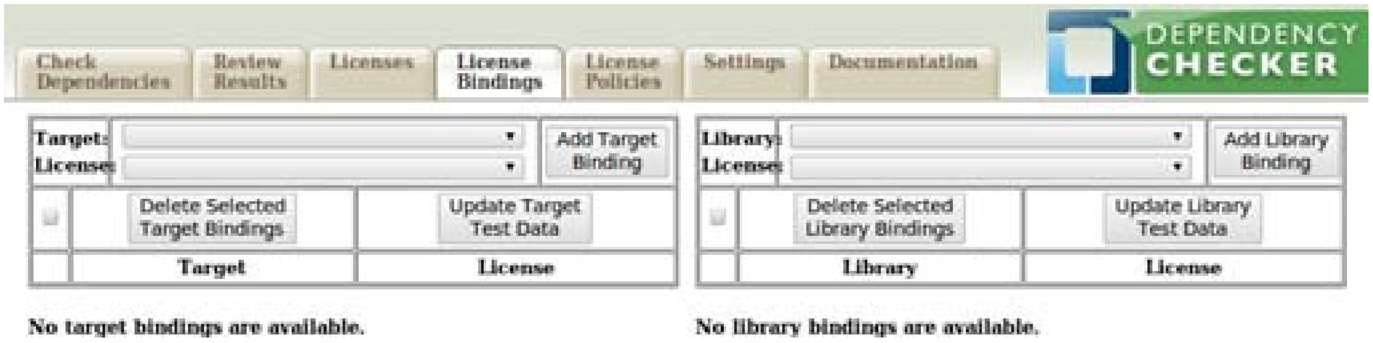
In this page, users create references to the various licenses used in their products to allow them to define policies around linkages made between libraries and libraries available under licenses. Because license lists may be imported from different sources with different naming conventions, one can create a list of possible equivalent strings (aliases) for a particular license. You can add additional aliases to the list for a license simply by entering more and using the "Add Aliases" button again. Alias names must be unique.

	Long Name	Abbreviation	Version
☐	Academic Free License 3.0	AFL	3.0
☐	Adaptive Public License	APL	
☐	Affero GNU Public License	AGPL	
☐	Apache License	Apache	2.0
☐	Apple Public Source License	Apple	
☐	Artistic license 2.0	Artistic	2.0
☐	Attribution Assurance License	AAL	
☐	Boost Software License	BSL	1.0
☐	CUA Office Public License Version 1.0	CUA-OPL	1.0
☐	Common Development and Distribution License	CDDL	
☐	Common Public Attribution License 1.0	CPAL	1.0
☐	Computer Associates Trusted Open Source License	CATOSL	1.1
☐	EU DataGrid Software License	EUDSL	
☐	Eclipse Public License	EPL	
☐	Educational Community License Version 2.0	ECL	2.0
☐	Eiffel Forum License V2.0	EFL	2.0
☐	Entessa Public License	EPL	
☐	Fair License	Fair	
☐	Frameworx License	Frameworx	
☐	GNU General Public License (GPL)	GPL	
☐	GNU General Public License version 3.0 (GPLv3)	GPL	3.0
☐	GNU Library or "Lesser" General Public License (LGPL)	LGPL	
☐	GNU Library or "Lesser" General Public License version 3.0 (LGPLv3)	LGPL	3.0
☐	Historical Permission Notice and Disclaimer	Historical	
☐	IBM Public License	IBM	
☐	IPA Font License	IPA	
☐	ISC License	ISC	
☐	Lucent Public License Version 1.02	Lucent	1.02
☐	MIT license	MIT	
☐	Microsoft Public License (Ms-PL)	Ms-PL	
☐	Microsoft Reciprocal License (Ms-RL)	Ms-RL	
☐	MirOS License	MirOS	
☐	Motosoto License	Motosoto	
☐	Mozilla Public License 1.1 (MPL)	MPL	1.1
☐	Multics License	Multics	
☐	NASA Open Source Agreement 1.3	NASA	1.3
☐	NTP License	NTP	
☐	Naumen Public License	Naumen	
☐	Nethack General Public License	Nethack	
☐	New and Simplified BSD licenses	BSD	
☐	Nokia Open Source License	Nokia	
☐	Non-Profit Open Software License 3.0 (Non-Profit OSL 3.0)	NPOSL	3.0
☐	OCLC Research Public License 2.0	OSLC	2.0
☐	Open Font License 1.1 (OFL 1.1)	OFL	1.1
☐	Open Group Test Suite License	OGTSL	
☐	Open Software License 3.0 (OSL 3.0)	OSL	3.0
☐	PHP License	PHP	
☐	Python Software Foundation License	Python	
☐	Python license (CNRI Python License)	CNRI Python	
☐	Qt Public License (QPL)	QPL	
☐	RealNetworks Public Source License V1.0	RealNW	1.0
☐	Reciprocal Public License 1.5 (RPL1.5)	RPL	1.5
☐	Ricoh Source Code Public License	Ricoh	
☐	Simple Public License 2.0	Simple	2.0
☐	Sleepycat License	Sleepycat	
☐	Sun Public License	Sun	
☐	Sybase Open Watcom Public License 1.0	Sybase	1.0
☐	The PostgreSQL License	PostgreSQL	
☐	University of Illinois/NCSA Open Source License	UI/NCSA	
☐	Vovida Software License v. 1.0	Vovida	1.0
☐	W3C License	W3C	
☐	X.Net License	X.Net	
☐	Zope Public License	Zope	
☐	wxWindows Library License	wxWindows	
☐	zlib/libpng license	zlib/libpng	

	License	Aliases
☐	AFL 3.0	AFL3 | AFLv3
☐	Apache 2.0	Apachev2 | Apache2
☐	Artistic 2.0	Artistic2 | Artisticv2
☐	BSL 1.0	BSLv1 | BSL1
☐	CATOSL 1.1	CATOSL1.1 | CATOSLv1
☐	CPAL 1.0	CPAL1 | CPALv1
☐	CUAOPL 1.0	CUAOPL1 | CUAOPLv1
☐	ECL 2.0	ECL2 | ECLv2
☐	EFL 2.0	EFL2 | EFLv2
☐	GPL 3.0	GPL3 | GPLv3
☐	LGPL 3.0	LGPL3 | LGPLv3
☐	Lucent 1.02	Lucent1.02 | Lucentv1.02
☐	MPL 1.1	MPL1.1 | MPLv1.1
☐	NPOSL 3.0	NPOSL3 | NPOSLv3
☐	OFL 1.1	OFL1.1 | OFLv1.1
☐	OSL 3.0	OSL3 | OSLv3
☐	OSLC 2.0	OSLC2 | OSLCv2
☐	RPL 1.5	RPL1.5 | RPLv1.5
☐	RealNW 1.0	Real1 | Realv1
☐	Simple 2.0	Simple2 | Simplev2
☐	Sybase 1.0	Sybase1 | Sybasev1
☐	Vovida 1.0	Vovida1 | Vovidav1

License Bindings

Since the module to import license information from third party tools is not available yet, we needed a way to test the tool and the defined policy. The solution was to create a way to attach license information to binaries and libraries. This solution, described below, is a temporary fix until we are able to import license information attached to binaries and libraries from other tools.
The License Bindings tab illustrated in Figure 6 allows the user to define the license binding for the target files (i.e. the files that are being tested for dependencies). The same type of bindings can be done for the dependency libraries (i.e. the libraries linked to by the binaries).
The drop down under "Target" will show all the files captured in the target directory scanned.
The drop down under "License" will show all the licenses defined in the License tab. If there is no test-data or no licenses, the drop-downs will be empty. If there is test data in the system, you can update the license information for current data using the Update Target Test Data button.



No target bindings are available.

No library bindings are available.

The drop down under "Library" will show all libraries in the test data. The drop down under "License" will show all the licenses defined in the License tab. The License selector does not differentiate between static and dynamic versions; both will be treated the same. If there are no test data or no licenses, the drop-downs will be empty. If there are test data in the system, you can update the license information for current data using the Update Library Test Data button. The bindings will also get updated as part of the next test run.

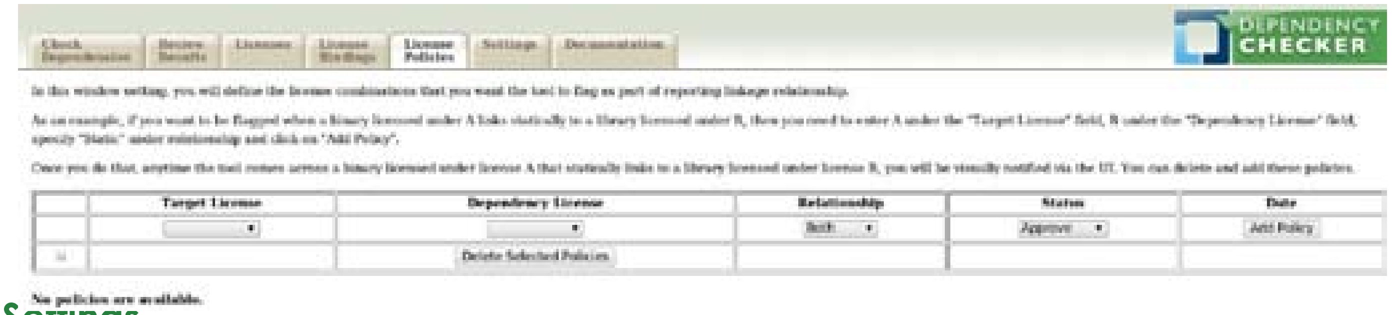
License Policies

The License Policies, shown in Figure 7, allow the user to define pairing of target/library licenses that could have potential issues. You select the Target License and Library License from the drop downs and then select the relationship, either Static, Dynamic, or both. When a test is run, violations of these policy settings will appear in the report printed in red with a red flag after the License name. Like the other tabs, you can select and delete policies using the check-boxes and the Delete Selected Policies button.

If the target (file) or library is using a license naming convention that is not defined in the application licenses tab, but has a naming convention defined as equivalent in the aliases table, the license violation will look like:

alias name (real name) [graphical flag]

Undefined license relationships will be flagged with orange text and an orange flag.



In this window setting, you will define the license combinations that you want the tool to flag as part of reporting linkage relationship.

As an example, if you want to be flagged when a binary licensed under A links statically to a library licensed under B, then you need to enter A under the "Target License" field, B under the "Dependency License" field, specify "Static" under relationship and click on "Add Policy".

Once you do that, anytime the tool comes across a binary licensed under license A that statically links to a library licensed under license B, you will be visually notified via the UI. You can delete and add these policies.

	Target License	Dependency License	Relationship	Status	Date
			Both	Approve	Add Policy
☐		Delete Selected Policies			

No policies are available.

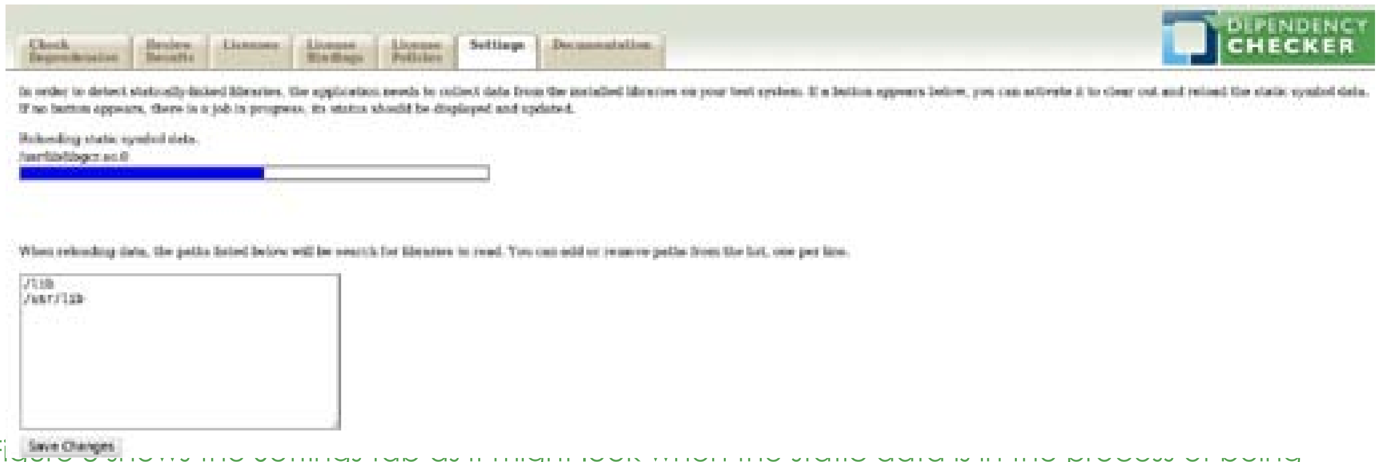
Settings

The Settings tab provides settings for managing the static symbol data used for detecting static library dependencies.

The Reload Static Data button clears out any data in the static symbol tables, and loads new data into them based on the libraries available on the system. It should be activated at least once: when running the Dependency Checker for the first time. Both this page and the Check Dependencies page warn you when no static symbol data is present.

If a job is being run, such as a dependency check or a symbol data reload, the status for that job will be displayed instead of the Reload Static Data button. When the job is finished and it is possible to reload the data again, the button will return.

By default, when loading symbol data, only libraries found in the common system paths are considered. You can change this by adding or removing paths from the list under the Reload Static Data button, and then clicking Save Changes. Once the paths have been changed, click the Reload Static Data button to reload the data from the new paths.



In order to detect statically-linked libraries, the application needs to collect data from the installed libraries on your test system. If a button appears below, you can activate it to clear out and reload the static symbol data. If no button appears, there is a job in progress, its status should be displayed and updated.

Reloading static symbol data.
/usr/lib/libgcc.so.0

When reloading data, the paths listed below will be search for libraries to read. You can add or remove paths from the list, one per line.

```
/lib
/usr/lib
```

Figure [illegible] shows the Settings tab as it might look when the static data is in the process of being reloaded.

Installation

From Packages

The program is packaged as an rpm package, with dependencies on python-django. If your system does not provide django, or it's named differently, you may need to install using --nodeps:

```
rpm -Uvh dep-checker-0.0.5-1.noarch.rpm --nodeps
```

Note: If you had to use --nodeps, then you must make sure django is installed and functional on your system. Both the command line program and the GUI depend on django.
The installation creates a compliance user/group and should create a desktop menu entry to launch the server and open the GUI in your web browser.

From Git

You can also checkout the project from git and run it in place:

```
git clone http://git.linuxfoundation.org/dep-checker.git

cd dep-checker
```

Alternately, you can get the latest tarball from the git web page by clicking on the snapshot link in the upper right-hand part of the page. Once you have the tarball, unpack it (example, the numbers of your download may differ):

```
tar -xf dep-checker-3af829ae0cc5aba33192c000ef0365ef6bced843.tar.gz

cd dep-checker
```

Create the application database and the documentation (you will need w3m to create README.txt):

```
make
```

Running the Tool

```
Usage: dep-checker.py [options] start | stop

Options:

    --force-root             allow running as root

    --server-only            don't open a browser

    --interface=INTERFACE    listen on network interface (port or ip:port)

    -h, --help               show this help message and exit
```

Step 1

The first step in running the tool is to:

```
~/dep-checker$ make
```

You will see output something like this:

```
cd package && make clean
make[1]: Entering directory `/home/stew/git/dep-checker/package'
make[1]: Leaving directory `/home/stew/git/dep-checker/package'
cd compliance/media/docs && make clean
make[1]: Entering directory `/home/stew/git/dep-checker/compliance/media/docs'
rm -f index.html index.html.addons
make[1]: Leaving directory `/home/stew/git/dep-checker/compliance/media/docs'
rm -f README.txt
rm -f compliance/compliance
rm -f compliance/compliance
cd compliance && python manage.py syncdb --noinput
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_user_permissions
Creating table auth_user_groups
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table linkage_test
Creating table linkage_file
Creating table linkage_lib
Creating table linkage_license
Creating table linkage_aliases
Creating table linkage_liblicense
Creating table linkage_filelicense
Creating table linkage_policy
Creating table linkage_staticsymbol
Creating table linkage_staticlibsearchpath
Creating table linkage_meta
Installing index for auth.Permission model
Installing index for auth.Group_permissions model
Installing index for auth.User_user_permissions model
Installing index for auth.User_groups model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for linkage.File model
Installing index for linkage.Lib model
Installing index for linkage.StaticSymbol model
Installing index for linkage.Meta model
Installing xml fixture 'initial_data' from '/home/stew/git/dep-checker/compliance/../compliance/linkage/
fixtures'.
Installed 202 object(s) from 1 fixture(s)
cd compliance/media/docs && make
make[1]: Entering directory `/home/stew/git/dep-checker/compliance/media/docs'
./text-docs-to-html > index.html.addons
cat index.html.base index.html.addons index.html.footer > index.html
make[1]: Leaving directory `/home/stew/git/dep-checker/compliance/media/docs'
w3m -dump compliance/media/docs/index.html > README.txt
```

If you've already been using the tool, you do not want to make clean, as it will destroy your existing data.

Step 2

The second step in running the tool is to:

```
~/dep-checker$ ./bin/dep-checker.py start
Waiting for the server to start...
Starting a web browser.
Created a new window in existing browser session
```

Stopping the Server

```
./bin/dep-checker.py stop
```

Known Limitations and Areas to Improve

Debugging Information

The process of detecting statically linked libraries has some limitations because the normal build process does not embed any information about symbols linked from static libraries, as it does for symbols linked from dynamic libraries. The process is therefore less about gathering the data already embedded in the application, and more about attempting to derive the data using hints in the application that are not directly relevant to the linking process.

The process of detecting statically linked libraries assumes that no deliberate deception is being attempted. A malicious developer could easily tweak a build to avoid detection of certain static libraries. The process relies absolutely on the presence of debugging information. If a software provider strip out debugging information, the tool cannot detect static library usage in most released code. Therefore, we recommend incorporating the tool at an earlier phase of the project's life-cycle, and verifying that debugging information is available when running the tool to ensure all statically linked libraries are discovered.

Software Development Practices

Static library detection also relies on common developer and distribution vendor practices. Developers tend to create debugging information in anything they build themselves, and to not care about debugging information in libraries not being built by the developer. Distribution vendors tend to ship debugging libraries for dynamic libraries only. These are trends, not absolute rules; developers and distribution vendors may change their practices for perfectly valid reasons. A compliance engineer relying on this tool's static library analysis should be familiar with the common practices of both the company's own developers and their preferred distribution.

Accuracy

Detection of actual libraries depends on a table of symbols that maps symbol names found to the library the symbol comes from. Accuracy in detection depends on having high-quality data in this table. Because of this, the tool generates the data from the system actually running the tests, instead of shipping a canonical database that can become stale. Ideally, the tool would be running on the same system used to build the software under test. If this is not possible, the tool should be run on a system set up to match the build system as closely as possible.

Additionally, if the development team builds static libraries of their own, those static libraries should also be reflected in the table of symbols. Make sure that the search paths for libraries in the tool include the directories where those libraries are installed. Either static or dynamic versions of the libraries will work.

Recursion

Because dynamic library linkage is much more explicit, the tool reports dynamic dependencies with high confidence. The limitation comes when using recursion, because this will often result in an examination of libraries shipped by the distribution vendor. In many cases, the dependencies uncovered may not be relevant to the compliance/legal status of the application under test. For

example, if an application dynamically links to library A on the system, and that library is shipped by the distribution to itself link to library B for some optional functionality not used by the application, the legal status of the application may be different than if the application linked to both libraries A and B directly. The tool provides the mechanism to uncover such relationship through recursion, but does not make any statement regarding the compliance status.
Further, dependencies of system libraries can change between distributions and even between versions of the same distribution. Thus, the recursive dependencies as reported by the tool can be different depending on which distribution the test is being run under.

Areas for Contributions

Being a first release, there are several areas that can be improved or augmented with additional functionalities. The listing below (in random order) provides a TO DO list for the next phase of the tool development:

- Plugin import license information from source code scanning tools
- Improve the GUI
- Improve detection of static linkages
- Improve the CLI to provide similar functionalities to those available through the GUI
- Improve the speed of load table of symbols
- Improve the speed of running the dependency checker
- Provide the application as a web application

Availability and Participation

- You can view the development source from git at http://git.linuxfoundation.org/?p=dep-checker.git;a=summary, or check it out using standard git commands:
  % git clone http://git.linuxfoundation.org/dep-checker.git
- To file bugs or request features, go to http://bugs.linuxfoundation.org/ (under the Compliance product).
- To subscribe to the mailing list or view the archives: https://lists.linux-foundation.org/mailman/listinfo/dep-checker-dev.

Conclusion

The Dependency Checker Tool is an Open Source project initiated by the Linux Foundation to help companies with their compliance efforts. Please consider this article as an invitation to learn more about the project and to consider contributing to it.

About the Authors

Stew Benedict is a Member of Technical Staff at the Linux Foundation. He previously worked as a Distribution Developer for Mandrake/Mandriva, working on PPC/IA64 ports and various security initiatives. Prior to that, he held several roles with an automotive parts manufacturer in Cleveland, Ohio, and with GE Lighting as part of their Electronic Products group. He has written several Linux and Solaris articles for Linux Journal and TechRepublic.
Jeff Licquia is a Member of Technical Staff at the Linux Foundation. Previously, he worked at Progeny Linux Systems, and was the project lead for the Progeny Debian distribution. He is also a member of the Debian project, and has contributed to several books on Linux software development.

Over his career, he has worked on a diverse set of problems, from industrial automation to system administration, and on platforms ranging from MS-DOS and Ultrix to Windows and Linux.

About the Open Compliance Program

The Linux Foundation's Open Compliance Program is the industry's only neutral, comprehensive software compliance initiative. By marshaling the resources of its members and leaders in the compliance community, the Linux Foundation brings together the individuals, companies and legal entities needed to expand the use of open source software while decreasing legal costs and FUD. The Open Compliance Program offers comprehensive training and informational materials, open source tools, an online community (FOSSBazaar), a best practices checklist, a rapid alert directory of company's compliance officers and a standard to help companies uniformly tag and report software used in their products. The Open Compliance Program is led by experts in the compliance industry and backed by such organizations as the Adobe, AMD, ARM Limited, Cisco Systems, Google, HP, IBM, Intel, Motorola, NEC, Novell, Samsung, Software Freedom Law Center, Sony Electronics and many more. More information can be found at http://www.linuxfoundation.org/programs/legal/compliance.

The Linux Foundation promotes, protects and standardizes Linux by providing unified resources and services needed for open source to successfully compete with closed platforms.

To learn more about The Linux Foundation, the Open Compliance Program or our other initiatives please visit us at http://www.linuxfoundation.org/.

THE LINUX FOUNDATION