Zeek Package Manager Documentation

Release 1.5.4

The Zeek Project

Mar 22, 2019

Contents

The Zeek Package Manager makes it easy for Zeek users to install and manage third party scripts as well as plugins for Zeek and BroControl. The command-line tool is preconfigured to download packages from the Zeek package source , a GitHub repository that has been set up such that any developer can request their Zeek package be included. See the `README` file of that repository for information regarding the package submission process.

> **note** It's left up to users to decide for themselves via code review, GitHub comments/stars, or other metrics whether any given package is trustworthy as there is no implied guarantees that it's secure just because it's been accepted into the default package source.

See the package manager documentation for further usage information, how-to guides, and walkthroughs. For offline reading, it's also available in the `doc/` directory of the source code distribution.

CHAPTER 1

Quickstart Guide

1.1 Dependencies

- Python 2.7+ or 3.0+
- git: https://git-scm.com
- GitPython: https://pypi.python.org/pypi/GitPython
- semantic_version: https://pypi.python.org/pypi/semantic_version
- btest: https://pypi.python.org/pypi/btest
- configparser backport (not needed when using Python 3.5+): https://pypi.python.org/pypi/configparser

Note that following the suggested *Installation* process via **pip** will automatically install dependencies for you.

1.2 Installation

Using the latest stable release on PyPI:

```
$ pip install bro-pkg
```

Using the latest git development version:

```
$ pip install git+git://github.com/bro/package-manager@master
```

1.3 Basic Configuration

After installing via **pip**, additional configuration is required. First, make sure that the **bro-config** script that gets installed with **bro** is in your PATH. Then, as the user you want to run **bro-pkg** with, do:

```
$ bro-pkg autoconfig
```

This automatically generates a config file with the following suggested settings that should work for most Bro deployments:

- *script_dir*: set to the location of Bro's `site` scripts directory (e.g. *<bro_install_prefix>*`/share/bro/site`)
- *plugin_dir*: set to the location of Bro's default plugin directory (e.g. *<bro_install_prefix>*`/lib/bro/plugins`)
- *bro_dist*: set to the location of Bro's source code. If you didn't build/install Bro from source code, this field will not be set, but it's only needed if you plan on installing packages that have uncompiled Bro plugins.

With those settings, the package manager will install Bro scripts, Bro plugins, and BroControl plugins into directories where **bro** and **broctl** will, by default, look for them. BroControl clusters will also automatically distribute installed package scripts/plugins to all nodes.

Note: If your Bro installation is owned by "root" and you intend to run **bro-pkg** as a different user, then you should grant "write" access to the directories specified by *script_dir* and *plugin_dir*. E.g. you could do something like:

```
$ sudo chgrp $USER $(bro-config --site_dir) $(bro-config --plugin_dir)
$ sudo chmod g+rwX $(bro-config --site_dir) $(bro-config --plugin_dir)
```

The final step is to edit your `site/local.bro`. If you want to have Bro automatically load the scripts from all *installed* packages that are also marked as "*loaded*" add:

```
@load packages
```

If you prefer to manually pick the package scripts to load, you may instead add lines like `@load` *<package_name>*, where *<package_name>* is the *shorthand name* of the desired package.

If you want to further customize your configuration, see the *Advanced Configuration* section and also check *here* for a full explanation of config file options. Otherwise you're ready to use *bro-pkg*.

1.4 Advanced Configuration

If you prefer to not use the suggested *Basic Configuration* settings for *script_dir* and *plugin_dir*, the default configuration will install all package scripts/plugins within `~/.bro-pkg` or you may change them to whatever location you prefer. These will be referred to as "non-standard" locations in the sense that vanilla configurations of either **bro** or **broctl** will not detect scripts/plugins in those locations without additional configuration.

When using non-standard location, follow these steps to integrate with **bro** and **broctl**:

- To get command-line **bro** to be aware of Bro scripts/plugins in a non-standard location, make sure the **bro-config** script (that gets installed along with **bro**) is in your `PATH` and run:

  ```
  $ `bro-pkg env`
  ```

  Note that this sets up the environment only for the current shell session.

- To get **broctl** to be aware of scripts/plugins in a non-standard location, run:

  ```
  $ bro-pkg config script_dir
  ```

  And set the *SitePolicyPath* option in `broctl.cfg` based on the output you see. Similarly, run:

```
$ bro-pkg config plugin_dir
```

And set the *SitePluginPath* option in `broctl.cfg` based on the output you see.

1.5 Usage

Check the output of *bro-pkg –help* for an explanation of all available functionality of the command-line tool.

1.5.1 Package Upgrades/Versioning

When installing packages, note that the *install command*, has a `--version` flag that may be used to install specific package versions which may either be git release tags or branch names. The way that **bro-pkg** receives updates for a package depends on whether the package is first installed to track stable releases or a specific git branch. See the *package upgrade process* documentation to learn how **bro-pkg** treats each situation.

1.5.2 Offline Usage

It's common to have limited network/internet access on the systems where Bro is deployed. To accomodate those scenarios, **bro-pkg** can be used as normally on a system that *does* have network access to create bundles of its package installation environment. Those bundles can then be transferred to the deployment systems via whatever means are appropriate (SSH, USB flash drive, etc).

For example, on the package management system you can do typical package management tasks, like install and update packages:

```
$ bro-pkg install <package name>
```

Then, via the *bundle command*, create a bundle file which contains a snapshot of all currently installed packages:

```
$ bro-pkg bundle bro-packages.bundle
```

Then transfer `bro-packages.bundle` to the Bro deployment management host. For Bro clusters using BroControl, this will be the system acting as the "manager" node. Then on that system (assuming it already as **bro-pkg** installed and configured):

```
$ bro-pkg unbundle bro-packages.bundle
```

Finally, if you're using BroControl, and the unbundling process was successful, you need to deploy the changes to worker nodes:

```
$ broctl deploy
```

CHAPTER 2

bro-pkg Command-Line Tool

A command-line package manager for Bro.

```
usage: bro-pkg [-h] [--version] [--configfile CONFIGFILE] [--verbose]
               {test,install,bundle,unbundle,remove,purge,refresh,upgrade,load,unload,
↪pin,unpin,list,search,info,config,autoconfig,env}
               ...
```

Options:

--version	show program's version number and exit
--configfile	Path to Bro Package Manager config file. See *Config File*.
--verbose=0, -v=0	Increase program output for debugging. Use multiple times for more output (e.g. -vvv).

Environment Variables:

`BRO_PKG_CONFIG_FILE`: Same as `--configfile` option, but has less precedence.

2.1 Commands

2.1.1 test

Runs the unit tests for the specified Bro packages. In most cases, the "bro" and "bro-config" programs will need to be in PATH before running this command.

```
usage: bro-pkg test [-h] [--version VERSION] package [package ...]
```

Positional arguments:

package The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo".

Options:

--version The version of the package to test. Only one package may be specified at a time when using this flag. A version tag, branch name, or commit hash may be specified here. If the package name refers to a local git repo with a working tree, then its currently active branch is used. The default for other cases is to use the latest version tag, or if a package has none, the "master" branch.

2.1.2 install

Installs packages from a configured package source or directly from a git URL. After installing, the package is marked as being "loaded" (see the `load` command).

```
usage: bro-pkg install [-h] [--force] [--skiptests] [--nodeps]
                       [--nosuggestions] [--version VERSION]
                       package [package ...]
```

Positional arguments:

package The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo".

Options:

--force=False Skip the confirmation prompt.

--skiptests=False Skip running unit tests for packages before installation.

--nodeps=False Skip all dependency resolution/checks. Note that using this option risks putting your installed package collection into a broken or unusable state.

--nosuggestions=False Skip automatically installing suggested packages.

--version The version of the package to install. Only one package may be specified at a time when using this flag. A version tag, branch name, or commit hash may be specified here. If the package name refers to a local git repo with a working tree, then its currently active branch is used. The default for other cases is to use the latest version tag, or if a package has none, the "master" branch.

2.1.3 remove

Unloads (see the `unload` command) and uninstalls a previously installed package.

```
usage: bro-pkg remove [-h] [--force] package [package ...]
```

Positional arguments:

package The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo".

Options:

--force=False Skip the confirmation prompt.

2.1.4 purge

Unloads (see the `unload` command) and uninstalls all previously installed packages.

```
usage: bro-pkg purge [-h] [--force]
```

Options:

--force=False Skip the confirmation prompt.

2.1.5 bundle

This command creates a bundle file containing a collection of Bro packages. If `--manifest` is used, the user supplies the list of packages to put in the bundle, else all currently installed packages are put in the bundle. A bundle file can be unpacked on any target system, resulting in a repeatable/specific set of packages being installed on that target system (see the `unbundle` command). This command may be useful for those that want to manage packages on a system that otherwise has limited network connectivity. E.g. one can use a system with an internet connection to create a bundle, transport that bundle to the target machine using whatever means are appropriate, and finally unbundle/install it on the target machine.

```
usage: bro-pkg bundle [-h] [--force] [--nodeps] [--nosuggestions]
                      [--manifest MANIFEST [MANIFEST ...] --]
                      filename.bundle
```

Positional arguments:

filename.bundle The path of the bundle file to create. It will be overwritten if it already exists. Note that if –manifest is used before this filename is specified, you should use a double-dash, –, to first terminate that argument list.

Options:

--force=False Skip the confirmation prompt.

--nodeps=False Skip all dependency resolution/checks. Note that using this option risks creating a bundle of packages that is in a broken or unusable state.

--nosuggestions=False Skip automatically bundling suggested packages.

--manifest This may either be a file name or a list of packages to include in the bundle. If a file name is supplied, it should be in INI format with a single ‘‘[bundle]‘‘ section. The keys in that section correspond to package names and their values correspond to git version tags, branch names, or commit hashes. The values may be left blank to indicate that the latest available version should be used.

2.1.6 unbundle

This command unpacks a bundle file formerly created by the `bundle` command and installs all the packages contained within.

```
usage: bro-pkg unbundle [-h] [--force] [--replace] filename.bundle
```

Positional arguments:

filename.bundle The path of the bundle file to install.

Options:

--force=False Skip the confirmation prompt.

--replace=False Using this flag first removes all installed packages before then installing the packages from the bundle.

2.1.7 refresh

Retrieve latest package metadata from sources and checks whether any installed packages have available upgrades. Note that this does not actually upgrade any packages (see the `upgrade` command for that).

```
usage: bro-pkg refresh [-h] [--aggregate] [--push]
                       [--sources SOURCES [SOURCES ...]]
```

Options:

--aggregate=False Crawls the urls listed in package source bro-pkg.index files and aggregates the metadata found in their bro-pkg.meta files. The aggregated metadata is stored in the local clone of the package source that bro-pkg uses internally locating package metadata. For each package, the metadata is taken from the highest available git version tag or the master branch if no version tags exist

--push=False Push all local changes to package sources to upstream repos

--sources A list of package source names to operate on. If this argument is not used, then the command will operate on all configured sources.

2.1.8 upgrade

Uprades the specified package(s) to latest available version. If no specific packages are specified, then all installed packages that are outdated and not pinned are upgraded. For packages that are installed with `--version` using a git

branch name, the package is updated to the latest commit on that branch, else the package is updated to the highest available git version tag.

```
usage: bro-pkg upgrade [-h] [--force] [--skiptests] [--nodeps]
                       [--nosuggestions]
                       [package [package ...]]
```

Positional arguments:

package	The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo".

Options:

--force=False	Skip the confirmation prompt.
--skiptests=False	Skip running unit tests for packages before installation.
--nodeps=False	Skip all dependency resolution/checks. Note that using this option risks putting your installed package collection into a broken or unusable state.
--nosuggestions=False	Skip automatically installing suggested packages.

2.1.9 load

The Bro Package Manager keeps track of all packages that are marked as "loaded" and maintains a single Bro script that, when loaded by Bro (e.g. via `@load packages`), will load the scripts from all "loaded" packages at once. This command adds a set of packages to the "loaded packages" list.

```
usage: bro-pkg load [-h] package [package ...]
```

Positional arguments:

package	Name(s) of package(s) to load.

2.1.10 unload

The Bro Package Manager keeps track of all packages that are marked as "loaded" and maintains a single Bro script that, when loaded by Bro, will load the scripts from all "loaded" packages at once. This command removes a set of packages from the "loaded packages" list.

```
usage: bro-pkg unload [-h] package [package ...]
```

Positional arguments:

package	The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to

the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo".

2.1.11 pin

Pinned packages are ignored by the `upgrade` command.

```
usage: bro-pkg pin [-h] package [package ...]
```

Positional arguments:

package The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo".

2.1.12 unpin

Packages that are not pinned are automatically upgraded by the `upgrade` command

```
usage: bro-pkg unpin [-h] package [package ...]
```

Positional arguments:

package The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo".

2.1.13 list

Outputs a list of packages that match a given category.

```
usage: bro-pkg list [-h] [--nodesc]
                    [{all,installed,not_installed,loaded,unloaded,outdated}]
```

Positional arguments:

category Package category used to filter listing.

Possible choices: all, installed, not_installed, loaded, unloaded, outdated

Options:

--nodesc=False Do not display description text, just the package name(s).

2.1.14 search

Perform a substring search on package names and metadata tags. Surround search text with slashes to indicate it is a regular expression (e.g. `/text/`).

```
usage: bro-pkg search [-h] search_text [search_text ...]
```

Positional arguments:

search_text The text(s) or pattern(s) to look for.

2.1.15 info

Shows detailed information/metadata for given packages. If the package is currently installed, additional information about the status of it is displayed. E.g. the installed version or whether it is currently marked as "pinned" or "loaded."

```
usage: bro-pkg info [-h] [--version VERSION] [--nolocal] [--json]
                    [--jsonpretty SPACES] [--allvers]
                    package [package ...]
```

Positional arguments:

package The name(s) of package(s) to operate on. The package may be named in several ways. If the package is part of a package source, it may be referred to by the base name of the package (last component of git URL) or its path within the package source. If two packages in different package sources have conflicting paths, then the package source name may be prepended to the package path to resolve the ambiguity. A full git URL may also be used to refer to a package that does not belong to a source. E.g. for a package source called "bro" that has a package named "foo" located in "alice/bro-pkg.index" the following names work: "foo", "alice/foo", "bro/alice/foo". If a single name is given and matches one of the same categories as the "list" command, then it is automatically expanded to be the names of all packages which match the given category.

Options:

--version The version of the package metadata to inspect. A version tag, branch name, or commit hash and only one package at a time may be given when using this flag. If unspecified, the behavior depends on whether the package is currently installed. If installed, the metadata will be pulled from the installed version. If not installed, the latest version tag is used, or if a package has no version tags, the "master" branch is used.

--nolocal=False Do not read information from locally installed packages. Instead read info from remote GitHub.

--json=False Output package information as JSON.

--jsonpretty Optional number of spaces to indent for pretty-printed JSON output.

--allvers=False When outputting package information as JSON, show metadata for all versions. This option can be slow since remote repositories may be cloned multiple times. Also, installed packages will show metadata only for the installed version unless the –nolocal option is given.

2.1.16 config

The default output of this command is a valid package manager config file that corresponds to the one currently being used, but also with any defaulted field values filled in. This command also allows for only the value of a specific field to be output if the name of that field is given as an argument to the command.

```
usage: bro-pkg config [-h]
                      [{all,sources,user_vars,state_dir,script_dir,plugin_dir,bro_
↪dist}]
```

Positional arguments:

config_param	Name of a specific config file field to output. Possible choices: all, sources, user_vars, state_dir, script_dir, plugin_dir, bro_dist

2.1.17 autoconfig

The output of this command is a valid package manager config file that is generated by using the `bro-config` script that is installed along with Bro. It is the suggested configuration to use for most Bro installations. For this command to work, the `bro-config` script must be in `PATH`.

```
usage: bro-pkg autoconfig [-h]
```

2.1.18 env

This command returns shell commands that, when executed, will correctly set `BROPATH` and `BRO_PLUGIN_PATH` to utilize the scripts and plugins from packages installed by the package manager. For this command to function properly, either have the `bro-config` script (installed by bro) in `PATH`, or have the `BROPATH` and `BRO_PLUGIN_PATH` environment variables already set so this command can append package-specific paths to them.

```
usage: bro-pkg env [-h]
```

2.2 Config File

The **`bro-pkg`** command-line tool uses an INI-format config file to allow users to customize their *Package Sources*, *Package* installation paths, Bro executable/source paths, and other **`bro-pkg`** options.

See the default/example config file below for explanations of the available options and how to customize them:

```
# This is an example config file for bro-pkg to explain what
# settings are possible as well as their default values.
# The order of precedence for how bro-pkg finds/reads config files:
#
# (1) bro-pkg --configfile=/path/to/custom/config
# (2) the BRO_PKG_CONFIG_FILE environment variable
# (3) a config file located at $HOME/.bro-pkg/config
# (4) if none of the above exist, then bro-pkg uses builtin/default
#     values for all settings shown below

[sources]
```

(continues on next page)

(continued from previous page)

```
# The default package source repository from which bro-pkg fetches
# packages.  The default source may be removed, changed, or
# additional sources may be added as long as they use a unique key
# and a value that is a valid git URL.
bro = https://github.com/bro/packages

[paths]

# Directory where source repositories are cloned, packages are
# installed, and other package manager state information is
# maintained.  If left blank, this defaults to $HOME/.bro-pkg
state_dir =

# The directory where package scripts are copied upon installation.
# A subdirectory named "packages" is always created within the
# specified path and the package manager will copy the directory
# specified by the "script_dir" option of each package's bro-pkg.meta
# file there.
# If left blank, this defaults to <state_dir>/script_dir
# A typical path to set here is <bro_install_prefix>/share/bro/site
# If you decide to change this location after having already
# installed packages, bro-pkg will automatically relocate them
# the next time you run any bro-pkg command.
script_dir =

# The directory where package plugins are copied upon installation.
# A subdirectory named "packages" is always created within the
# specified path and the package manager will copy the directory
# specified by the "plugin_dir" option of each package's bro-pkg.meta
# file there.
# If left blank, this defaults to <state_dir>/plugin_dir
# A typical path to set here is <bro_install_prefix>/lib/bro/plugins
# If you decide to change this location after having already
# installed packages, bro-pkg will automatically relocate them
# the next time you run any bro-pkg command.
plugin_dir =

# The directory containing Bro distribution source code.  This is only
# needed when installing packages that contain Bro plugins that are
# not pre-built.
bro_dist =

[user_vars]

# For any key in this section that is matched for value interpolation
# in a package's bro-pkg.meta file, the corresponding value is
# substituted during execution of the package's `build_command`.
# This section is typically automatically populated with the
# the answers supplied during package installation prompts
# and, as a convenience feature, used to recall the last-used settings
# during subsequent operations (e.g. upgrades) on the same package.
```

CHAPTER 3

How-To: Create a Package

A Bro package may contain Bro scripts, Bro plugins, or BroControl plugins. Any number or combination of those components may be included within a single package.

The minimum requirement for a package is that it be in its own git repository and contain a metadata file named `bro-pkg.meta` at its top-level that begins with the line:

```
[package]
```

This is the package's metadata file in INI file format and may contain *additional fields* that describe the package as well as how it inter-operates with Bro, the package manager, or other packages.

Note that the shorthand name for your package that may be used by *bro-pkg* and Bro script `@load <package_name>` directives will be the last component of its git URL. E.g. a package at `https://github.com/bro/foo` may be referred to as **foo** when using **`bro-pkg`** and a Bro script that wants to load all the scripts within that package can use:

```
@load foo
```

3.1 Walkthroughs

3.1.1 Pure Bro Script Package

1. Create a git repository:

   ```
   $ mkdir foo && cd foo && git init
   ```

2. Create a package metadata file, `bro-pkg.meta`:

   ```
   $ echo '[package]' > bro-pkg.meta
   ```

3. Create a `__load__.bro` script with example code in it:

```
$ echo 'event bro_init() { print "foo is loaded"; }' > __load__.bro
```

4. (Optional) Relocate your `__load__.bro` script to any subdirectory:

```
$ mkdir scripts && mv __load__.bro scripts
$ echo 'script_dir = scripts' >> bro-pkg.meta
```

5. Commit everything to git:

```
$ git add * && git commit -m 'First commit'
```

6. (Optional) Test that Bro correctly loads the script after installing the package with **`bro-pkg`**:

```
$ bro-pkg install .
$ bro foo
$ bro-pkg remove .
```

7. (Optional) *Create a release version tag*.

See Bro Scripting for more information on developing Bro scripts.

3.1.2 Binary Bro Plugin Package

See Bro Plugins for more complete information on developing Bro plugins, though the following step are the essentials needed to create a package.

1. Create a plugin skeleton using `aux/bro-aux/plugin-support/init-plugin` from Bro's source distribution:

```
$ init-plugin ./rot13 Demo Rot13
```

2. Create a git repository

```
$ cd rot13 && git init
```

3. Create a package metadata file, `bro-pkg.meta`:

```
[package]
script_dir = scripts/Demo/Rot13
build_command = ./configure --bro-dist=%(bro_dist)s && make
```

   See the *Value Interpolation* section for more information on what the `%(bro_dist)s` string does.

4. Add example script code:

```
$ echo 'event bro_init() { print "rot13 plugin is loaded"; }' >> scripts/__load__.
↪bro
$ echo 'event bro_init() { print "rot13 script is loaded"; }' >> scripts/Demo/
↪Rot13/__load__.bro
```

5. Add an example builtin-function in `src/rot13.bif`:

```
module Demo;

function rot13%(s: string%) : string
    %{
```

(continues on next page)

(continued from previous page)

```
    char* rot13 = copy_string(s->CheckString());

    for ( char* p = rot13; *p; p++ )
        {
        char b = islower(*p) ? 'a' : 'A';
        *p  = (*p - b + 13) % 26 + b;
        }

    BroString* bs = new BroString(1, reinterpret_cast<byte_vec>(rot13),
                                  strlen(rot13));
    return new StringVal(bs);
    %}
```

6. Commit everything to git:

```
$ git add * && git commit -m 'First commit'
```

7. (Optional) Test that Bro correctly loads the plugin after installing the package with **bro-pkg**:

```
$ bro-pkg install .
$ bro rot13 -e 'print Demo::rot13("Hello")'
$ bro-pkg remove .
```

8. (Optional) *Create a release version tag*.

3.1.3 BroControl Plugin Package

1. Create a git repository:

```
$ mkdir foo && cd foo && git init
```

2. Create a package metadata file, `bro-pkg.meta`:

```
$ echo '[package]' > bro-pkg.meta
```

3. Create an example BroControl plugin, `foo.py`:

```
import BroControl.plugin
from BroControl import config

class Foo(BroControl.plugin.Plugin):
    def __init__(self):
        super(Foo, self).__init__(apiversion=1)

    def name(self):
        return "foo"

    def pluginVersion(self):
        return 1

    def init(self):
        self.message("foo plugin is initialized")
        return True
```

4. Set the *plugin_dir* metadata field to directory where the plugin is located:

```
$ echo 'plugin_dir = .' >> bro-pkg.meta
```

5. Commit everything to git:

```
$ git add * && git commit -m 'First commit'
```

6. (Optional) Test that BroControl correctly loads the plugin after installing the package with **bro-pkg**:

```
$ bro-pkg install .
$ broctl
$ bro-pkg remove .
```

7. (Optional) *Create a release version tag*.

See BroControl Plugins for more information on developing BroControl plugins.

If you want to distribute a BroControl plugin along with a Bro plugin in the same package, you may need to add the BroControl plugin's python script to the `bro_plugin_dist_files()` macro in the `CMakeLists.txt` of the Bro plugin so that it gets copied into `build/` along with the built Bro plugin. Or you could also modify your *build_command* to copy it there, but what ultimately matters is that the *plugin_dir* field points to a directory that contains both the Bro plugin and the BroControl plugin.

3.1.4 Registering to a Package Source

Registering a package to a package source is always the following basic steps:

1) Create a *Package Index File* for your package.

2) Add the index file to the package source's git repository.

The full process and conventions for submitting to the default package source can be found in the `README` at:

https://github.com/bro/packages

3.2 Package Metadata

See the following sub-sections for a full list of available fields that may be used in `bro-pkg.meta` files.

3.2.1 *description* field

The description field may be used to give users a general overview of the package and its purpose. The *bro-pkg list* will display the first sentence of description fields in the listings it displays. An example `bro-pkg.meta` using a description field:

```
[package]
description = Another example package.
    The description text may span multiple
    line: when adding line breaks, just
    indent the new lines so they are parsed
    as part of the 'description' value.
```

3.2.2 *aliases* field

The *aliases* field can be used to specify alternative names for a package. Users can then use `@load <package_alias>` for any alias listed in this field. This may be useful when renaming a package's repository on GitHub while still supporting users that already installed the package under the previous name. For example, if package *foo* were renamed to *foo2*, then the *aliases* for it could be:

```
[package]
aliases = foo2 foo
```

Currently, the order does not matter, but you should specify the canonical/current alias first. The list is delimited by commas or whitespace. If this field is not specified, the default behavior is the same as if using a single alias equal to the package's name.

The low-level details of the way this field operates is that, for each alias, it simply creates a symlink of the same name within the directory associated with the `script_dir` path in the *config file*.

Available **since bro-pkg v1.5**.

3.2.3 *credits* field

The *credits* field contains a comma-delimited set of author/contributor/maintainer names, descriptions, and/or email addresses.

It may be used if you have particular requirements or concerns regarding how authors or contributors for your package are credited in any public listings made by external metadata scraping tools (**bro-pkg** does not itself use this data directly for any functional purpose). It may also be useful as a standardized location for users to get contact/support info in case they encounter problems with the package. For example:

```
[package]
credits = A. Sacker <ace@sacker.com>.,
    JSON support added by W00ter (Acme Corporation)
```

3.2.4 *tags* field

The *tags* field contains a comma-delimited set of metadata tags that further classify and describe the purpose of the package. This is used to help users better discover and search for packages. The *bro-pkg search* command will inspect these tags. An example `bro-pkg.meta` using tags:

```
[package]
tags = bro plugin, broctl plugin, scan detection, intel
```

Suggested Tags

Some ideas for what to put in the *tags* field for packages:

- bro scripting
  - conn
  - intel
  - geolocation
  - file analysis

    - sumstats, summary statistics
    - input
    - log, logging
    - notices
- *<network protocol name>*
- *<file format name>*
- signatures
- bro plugin
    - protocol analyzer
    - file analyzer
    - bifs
    - packet source
    - packet dumper
    - input reader
    - log writer
- broctl plugin

3.2.5 *script_dir* field

The *script_dir* field is a path relative to the root of the package that contains a file named `__load__.bro` and possibly other Bro scripts. The files located in this directory are copied into *<user_script_dir>*`/packages/`*<package>*`/`, where **<*user_script_dir*>** corresponds to the *script_dir* field of the user's *config file* (typically *<bro_install_prefix>*`/share/bro/site`).

When the package is *loaded*, an `@load` *<package_name>* directive is added to *<user_script_dir>*`/packages/packages.bro`.

You may place any valid Bro script code within `__load__.bro`, but a package that contains many Bro scripts will typically have `__load__.bro` just contain a list of `@load` directives to load other Bro scripts within the package. E.g. if you have a package named **foo** installed, then it's `__load__.bro` will be what Bro loads when doing `@load foo` or running `bro foo` on the command-line.

An example `bro-pkg.meta`:

```
[package]
script_dir = scripts
```

For a `bro-pkg.meta` that looks like the above, the package should have a file called `scripts/__load__.bro`.

If the *script_dir* field is not present in `bro-pkg.meta`, it defaults to checking the top-level directory of the package for a `__load__.bro` script. If it's found there, **bro-pkg** use the top-level package directory as the value for *script_dir*. If it's not found, then **bro-pkg** assumes the package contains no Bro scripts (which may be the case for some plugins).

3.2.6 *plugin_dir* field

The *plugin_dir* field is a path relative to the root of the package that contains either pre-built Bro Plugins, BroControl Plugins, or both.

An example `bro-pkg.meta`:

```
[package]
script_dir = scripts
plugin_dir = plugins
```

For the above example, Bro and BroControl will load any plugins found in the installed package's `plugins/` directory.

If the *plugin_dir* field is not present in `bro-pkg.meta`, it defaults to a directory named `build/` at the top-level of the package. This is the default location where Bro binary plugins get placed when building them from source code (see the *build_command field*).

This field may also be set to the location of a tarfile that has a single top- level directory inside it containing the Bro plugin. The default CMake skeleton for Bro plugins produces such a tarfile located at `build/<namespace>_<plugin>.tgz`. This is a good choice to use for packages that will be published to a wider audience as installing from this tarfile contains the minimal set of files needed for the plugin to work whereas some extra files will get installed to user systems if the *plugin_dir* uses the default `build/` directory.

3.2.7 *build_command* field

The *build_command* field is an arbitrary shell command that the package manager will run before installing the package.

This is useful for distributing Bro Plugins as source code and having the package manager take care of building it on the user's machine before installing the package.

An example `bro-pkg.meta`:

```
[package]
script_dir = scripts/Demo/Rot13
build_command = ./configure --bro-dist=%(bro_dist)s && make
```

In the above example, the `%(bro_dist)s` string is *substituted* for the path the user has set for the *bro_dist* field in the *package manager config file*.

The default CMake skeleton for Bro plugins will use `build/` as the directory for the final/built version of the plugin, which matches the defaulted value of the omitted *plugin_dir* metadata field.

The *script_dir* field is set to the location where the author has placed custom scripts for their plugin. When a package has both a Bro plugin and Bro script components, the "plugin" part is always unconditionally loaded by Bro, but the "script" components must either be explicitly loaded (e.g. `@load <package_name>`) or the package marked as *loaded*.

Value Interpolation

The *build_command field* may reference the settings any given user has in their customized *package manager config file*.

For example, if a metadata field's value contains the `%(bro_dist)s` string, then **`bro-pkg`** operations that use that field will automatically substitute the actual value of *bro_dist* that the user has in their local config file. Note the trailing

's' character at the end of the interpolation string, `%(bro_dist)s` is intended/necessary for all such interpolation usages.

Besides the *bro_dist* config key, any key inside the *user_vars* sections of their *package manager config file* that matches the key of an entry in the package's *user_vars field* will be interpolated.

Internally, the value substitution and metadata parsing is handled by Python's configparser interpolation. See its documentation if you're interested in the details of how the interpolation works.

3.2.8 *user_vars* field

The *user_vars* field is used to solicit feedback from users for use during execution of the *build_command field*.

An example `bro-pkg.meta`:

```
[package]
build_command = ./configure --bro-dist=%(bro_dist)s --with-librdkafka=%(LIBRDKAFKA_
↪ROOT)s --with-libdub=%(LIBDBUS_ROOT)s && make
user_vars =
  LIBRDKAFKA_ROOT [/usr] "Path to librdkafka installation"
  LIBDBUS_ROOT [/usr] "Path to libdbus installation"
```

The format of the field is a sequence entries of the format:

```
key [value] "description"
```

The *key* is the string that should match what you want to be interpolated within the *build_command field*.

The *value* is provided as a convenient default value that you'd typically expect to work for most users.

The *description* is provided as an explanation for what the value will be used for.

Here's what a typical user would see:

```
$ bro-pkg install bro-test-package
The following packages will be INSTALLED:
  bro/jsiwek/bro-test-package (1.0.5)

Proceed? [Y/n] y
bro/jsiwek/bro-test-package asks for LIBRDKAFKA_ROOT (Path to librdkafka␣
↪installation) ? [/usr] /usr/local
Saved answers to config file: /Users/jon/.bro-pkg/config
Installed "bro/jsiwek/bro-test-package" (master)
Loaded "bro/jsiwek/bro-test-package"
```

The **bro-pkg** command will iterate over the *user_vars* field of all packages involved in the operation and prompt the user to provide a value that will work for their system.

If a user is using the `--force` option to **bro-pkg** commands or they are using the Python API directly, it will first look within the *user_vars* section of the user's *package manager config file* and, if it can't find the key there, it will fallback to use the default value from the package's metadata.

In any case, the user may choose to supply the value of a *user_vars* key via an environment variable, in which case, prompts are skipped for any keys located in the environment. The environment is also given priority over any values in the user's *package manager config file*.

Available **since bro-pkg v1.1**.

3.2.9 *test_command* field

The *test_command* field is an arbitrary shell command that the package manager will run when a user either manually runs the *test command* or before the package is installed or upgraded.

An example `bro-pkg.meta`:

```
[package]
test_command = cd testing && btest -d tests
```

The recommended test framework for writing package unit tests is btest. See its documentation for further explanation and examples.

3.2.10 *config_files* field

The *config_files* field may be used to specify a list of files that users are intended to directly modify after installation. Then, on operations that would otherwise destroy a user's local modifications to a config file, such as upgrading to a newer package version, **`bro-pkg`** can instead save a backup and possibly prompt the user to review the differences.

An example `bro-pkg.meta`:

```
[package]
script_dir = scripts
config_files = scripts/foo_config.bro, scripts/bar_config.bro
```

The value of *config_files* is a comma-delimited string of config file paths that are relative to the root directory of the package. Config files should either be located within the *script_dir* or *plugin_dir*.

3.2.11 *depends* field

The *depends* field may be used to specify a list of dependencies that the package requires.

An example `bro-pkg.meta`:

```
[package]
depends =
  bro >=2.5.0
  foo *
  https://github.com/bro/bar >=2.0.0
  package_source/path/bar branch=name_of_git_branch
```

The field is a list of dependency names and their version requirement specifications.

A dependency name may be either *bro*, *bro-pkg*, a full git URL of the package, or a *package shorthand name*.

- The special *bro* dependency refers not to a package, but the version of Bro that the package requires in order to function. If the user has **`bro-config`** in their `PATH` when installing/upgrading a package that specifies a *bro* dependency, then **`bro-pkg`** will enforce that the requirement is satisfied.
- The special *bro-pkg* dependency refers to the version of the package manager that is required by the package. E.g. if a package takes advantage of new features that are not present in older versions of the package manager, then it should indicate that so users of those old version will see an error message an know to upgrade instead of seeing a cryptic error/exception, or worse, seeing no errors, but without the desired functionality being performed. Note that this feature itself is only available **`since bro-pkg v1.2`**.

- The full git URL may be directly specified in the *depends* metadata if you want to force the dependency to always resolve to a single, canonical git repository. Typically this is the safe approach to take when listing package dependencies and for publicly visible packages.
- When using shorthand package dependency names, the user's **bro-pkg** will try to resolve the name into a full git URL based on the package sources they have configured. Typically this approach may be most useful for internal or testing environments.

A version requirement may be either a git branch name or a semantic version specification. When using a branch as a version requirement, prefix the branchname with `branch=`, else see the Semantic Version Specification documentation for the complete rule set of acceptable version requirement strings. Here's a summary:

- `*`: any version (this will also satisfy/match on git branches)
- `<1.0.0`: versions less than 1.0.0
- `<=1.0.0`: versions less than or equal to 1.0.0
- `>1.0.0`: versions greater than 1.0.0
- `>=1.0.0`: versions greater than or equal to 1.0.0
- `==1.0.0`: exactly version 1.0.0
- `!=1.0.0`: versions not equal to 1.0.0
- `^1.3.4`: versions between 1.3.4 and 2.0.0 (not including 2.0.0)
- `~1.2.3`: versions between 1.2.3 and 1.3.0 (not including 1.3.0)
- `~=2.2`: versions between 2.2.0 and 3.0.0 (not included 3.0.0)
- `~=1.4.5`: versions between 1.4.5 and 1.5.0 (not including 3.0.0)
- Any of the above may be combined by a separating comma to logically "and" the requirements together. E.g. `>=1.0.0,<2.0.0` means "greater or equal to 1.0.0 and less than 2.0.0".

Note that these specifications are strict semantic versions. Even if a given package chooses to use the `vX.Y.Z` format for its *git version tags*, do not use the 'v' prefix in the version specifications here as that is not part of the semantic version.

3.2.12 *external_depends* field

The *external_depends* field follows the same format as the *depends field*, but the dependency names refer to external/third-party software packages. E.g. these would be set to typical package names you'd expect the package manager from any given operating system to use, like 'libpng-dev'. The version specification should also generally be given in terms of semantic versioning where possible. In any case, the name and version specification for an external dependency are only used for display purposes – to help users understand extra pre-requisites that are needed for proceeding with package installation/upgrades.

Available **since bro-pkg v1.1**.

3.2.13 *suggests* field

The *suggests* field follows the same format as the *depends field*, but it's used for specifying optional packages that users may want to additionally install. This is helpful for suggesting complementary packages that aren't strictly required for the suggesting package to function properly.

A package in *suggests* is functionaly equivalent to a package in *depends* except in the way it's presented to users in various prompts during **bro-pkg** operations. Users also have the option to ignore suggestions by supplying an additional `--nosuggestions` flag to **bro-pkg** commands.

Available **since bro-pkg v1.3**.

3.3 Package Versioning

3.3.1 Creating New Package Release Versions

Package's should use git tags for versioning their releases. Use the Semantic Versioning numbering scheme here. For example, to create a new tag for a package:

```
$ git tag -a 1.0.0 -m 'Release 1.0.0'
```

The tag name may also be of the `vX.Y.Z` form (prefixed by 'v'). Choose whichever you prefer.

Then, assuming you've already set up a public/remote git repository (e.g. on GitHub) for your package, remember to push the tag to the remote repository:

```
$ git push --tags
```

Alternatively, if you expect to have a simple development process for your package, you may choose to not create any version tags and just always make commits directly to your package's *master* branch. Users will receive package updates differently depending on whether you decide to use release version tags or not. See the *package upgrade process* documentation for more details on the differences.

3.3.2 Package Upgrade Process

The *install command* will either install a stable release version or the latest commit on a specific git branch of a package.

The default installation behavior of **bro-pkg** is to look for the latest release version tag and install that. If there are no such version tags, it will fall back to installing the latest commit of the package's *master* branch

Upon installing a package via a *git version tag*, the *upgrade command* will only upgrade the local installation of that package if a greater version tag is available. In other words, you only receive stable release upgrades for packages installed in this way.

Upon installing a package via a git branch name, the *upgrade command* will upgrade the local installation of the package whenever a new commit becomes available at the end of the branch. This method of tracking packages is suitable for testing out development/experimental versions of packages.

If a package was installed via a specific commit hash, then the package will never be eligible for automatic upgrades.

CHAPTER 4

How-To: Create a Package Source

bro-pkg, by default, is configured to obtain packages from a single "package source", the Bro Packages Git Repository, which is hosted by and loosely curated by the Bro Team. However, users may *configure bro-pkg* to use other package sources: either ones they've set up themselves for organization purposes or those hosted by other third parties.

4.1 Package Source Setup

In order to set up such a package source, one simply has to create a git repository and then add *Package Index Files* to it. These files may be created at any path in the package source's git repository. E.g. the Bro Packages Git Repository organizes package index files hierarchically based on package author names such as `alice/bro-pkg.index` or `bob/bro-pkg.index` where `alice` and `bob` are usually GitHub usernames or some unique way of identifying the organization/person that maintains Bro packages. However, a source is free to use a flat organization with a single, top-level `bro-pkg.index`.

After creating a git repo for the package source and adding package index files to it, it's ready to be used by *bro-pkg*.

4.2 Package Index Files

Files named `bro-pkg.index` are used to describe the *Bro Packages* found within the package source. They are simply a list of git URLs pointing to the git repositories of packages. For example:

```
https://github.com/bro/foo
https://github.com/bro/bar
https://github.com/bro/baz
```

Local filesystem paths are also valid if the package source is only meant for your own private usage or testing.

4.3 Adding Packages

Adding packages is as simple as adding new *Package Index Files* or extending existing ones with new URLs and then commiting/pushing those changes to the package source git repository.

bro-pkg will see new packages listed the next time it uses the *refresh command*.

4.4 Removing Packages

Just remove the package's URL from the *Package Index File* that it's contained within.

After the next time **bro-pkg** uses the *refresh command*, it will no longer see the now-removed package when viewing package listings via by the *list command*.

Users that had previously installed the now-removed package may continue to use it and receive updates for it.

4.5 Aggregating Metadata

The maintainer/operator of a package source may choose to periodically aggregate the metadata contained in its package's `bro-pkg.meta` files. The *bro-pkg refresh* is used to perform the task. For example:

```
$ bro-pkg refresh --aggregate --push --sources my_source
```

The optional `--push` flag is helpful for setting up cron jobs to automatically perform this task periodically, assuming you've set up your git configuration to push changesets without interactive prompts. E.g. to set up pushing to remote servers you could set up SSH public key authentication.

Aggregated metadata gets written to a file named `aggregate.meta` at the top-level of a package source and the *list*, *search*, and *info* all may access this file. Having access to the aggregated metadata in this way is beneficial to all **bro-pkg** users because they then will not have to crawl the set of packages listed in a source in order to obtain this metadata as it will have already been pre-aggregated by the operator of the package source.

CHAPTER 5

Python API Reference

This package defines a Python interface for installing, managing, querying, and performing other operations on Bro Packages and Package Sources. The main entry point is the *Manager* class.

This package provides a logger named *LOG* to which logging stream handlers may be added in order to help log/debug applications.

The following Python modules are all provided as part of the `bropkg` public interface:

5.1 bropkg.manager module

A module defining the main Bro Package Manager interface which supplies methods to interact with and operate on Bro packages.

class bropkg.manager.**Manager**(*state_dir*, *script_dir*, *plugin_dir*, *bro_dist=''*, *user_vars=None*)

Bases: object

A package manager object performs various operations on packages.

It uses a state directory and a manifest file within it to keep track of package sources, installed packages and their statuses.

sources

dictionary package sources keyed by the name given to *add_source()*

Type dict of str -> *source.Source*

installed_pkgs

a dictionary of installed packaged keyed on package names (the last component of the package's git URL)

Type dict of str -> *package.InstalledPackage*

bro_dist

path to the Bro source code distribution. This is needed for packages that contain Bro plugins that need to be built from source code.

Type str

state_dir
the directory where the package manager will a maintain manifest file, package/source git clones, and other persistent state the manager needs in order to operate

Type str

user_vars
dictionary of key-value pairs where the value will be substituted into package build commands in place of the key.

Type dict of str -> str

backup_dir
a directory where the package manager will store backup files (e.g. locally modified package config files)

Type str

log_dir
a directory where the package manager will store misc. logs files (e.g. package build logs)

Type str

scratch_dir
a directory where the package manager performs miscellaneous/temporary file operations

Type str

script_dir
the directory where the package manager will copy each installed package's *script_dir* (as given by its `bro-pkg.meta` file). Each package gets a subdirectory within *script_dir* associated with its name.

Type str

plugin_dir
the directory where the package manager will copy each installed package's *plugin_dir* (as given by its `bro-pkg.meta` file). Each package gets a subdirectory within *plugin_dir* associated with its name.

Type str

source_clonedir
the directory where the package manager will clone package sources. Each source gets a subdirectory associated with its name.

Type str

package_clonedir
the directory where the package manager will clone installed packages. Each package gets a subdirectory associated with its name.

Type str

package_testdir
the directory where the package manager will run tests. Each package gets a subdirectory associated with its name.

Type str

manifest
the path to the package manager's manifest file. This file maintains a list of installed packages and their status.

Type str

autoload_script
path to a Bro script named `packages.bro` that the package manager maintains. It is a list of `@load` for each installed package that is marked as loaded (see *load()*).

Type str

autoload_package
path to a Bro `__load__.bro` script which is just a symlink to *autoload_script*. It's always located in a directory named `packages`, so as long as `BROPATH` is configured correctly, `@load packages` will load all installed packages that have been marked as loaded.

Type str

add_source(*name*, *git_url*)
Add a git repository that acts as a source of packages.

Parameters

- **name** (*str*) – a short name that will be used to reference the package source.
- **git_url** (*str*) – the git URL of the package source

Returns empty string if the source is successfully added, else the reason why it failed.

Return type str

backup_modified_files(*backup_subdir*, *modified_files*)
Creates backups of modified config files

Parameters

- **modified_files** (*list of (str, str)*) – the return value of *modified_config_files()*.
- **backup_subdir** (*str*) – the subdir of *backup_dir* in which

Returns paths indicating the backup locations. The order of the returned list corresponds directly to the order of *modified_files*.

Return type list of str

bro_plugin_path()
Return the path where installed package plugins are located.

This path can be added to `BRO_PLUGIN_PATH` for interoperability with Bro.

bropath()
Return the path where installed package scripts are located.

This path can be added to `BROPATH` for interoperability with Bro.

bundle(*bundle_file*, *package_list*, *prefer_existing_clones=False*)
Creates a package bundle.

Parameters

- **bundle_file** (*str*) – filesystem path of the zip file to create.
- **package_list** (*list of (str, str)*) – a list of (git URL, version) string tuples to put in the bundle. If the version string is empty, the latest available version of the package is used.
- **prefer_existing_clones** (*bool*) – if True and the package list contains a package at a version that is already installed, then the existing git clone of that package is put into the bundle instead of cloning from the remote repository.

Returns empty string if the bundle is successfully created, else an error string explaining what failed.

Return type str

bundle_info(*bundle_file*)
Retrieves information on all packages contained in a bundle.

Parameters **bundle_file** (*str*) – the path to the bundle to inspect.

Returns a tuple with the the first element set to an empty string if the information successfully retrieved, else an error message explaining why the bundle file was invalid. The second element of the tuple is a list containing information on each package contained in the bundle: the exact git URL and version string from the bundle's manifest along with the package info object retrieved by inspecting git repo contained in the bundle.

Return type (str, list of (str, str, *package.PackageInfo*))

find_installed_package(*pkg_path*)
Return an *package.InstalledPackage* if one matches the name.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

A package's name is the last component of it's git URL.

has_scripts(*installed_pkg*)
Return whether a *package.InstalledPackage* installed scripts.

Parameters **installed_pkg** (*package.InstalledPackage*) – the installed package to check for whether it has installed any Bro scripts.

Returns True if the package has installed Bro scripts.

Return type bool

info(*pkg_path*, *version=''*, *prefer_installed=True*)
Retrieves information about a package.

Parameters

- **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".
- **version** (*str*) – may be a git version tag, branch name, or commit hash from which metadata will be pulled. If an empty string is given, then the latest git version tag is used (or the "master" branch if no version tags exist).
- **prefer_installed** (*bool*) – if this is set, then the information from any current installation of the package is returned instead of retrieving the latest information from the package's git repo. The *version* parameter is also ignored when this is set as it uses whatever version of the package is currently installed.

Returns A *package.PackageInfo* object.

install(*pkg_path*, *version=''*)
Install a package.

Parameters

- **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".
- **version** (*str*) – if not given, then the latest git version tag is installed (or if no version tags exist, the "master" branch is installed). If given, it may be either a git version tag, a git branch name, or a git commit hash.

Returns empty string if package installation succeeded else an error string explaining why it failed.

Return type str

Raises **IOError** – if the manifest can't be written

installed_packages()
Return list of *package.InstalledPackage*.

load(*pkg_path*)
Mark an installed package as being "loaded".

The collection of "loaded" packages is a convenient way for Bro to more simply load a whole group of packages installed via the package manager.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

Returns empty string if the package is successfully marked as loaded, else an explanation of why it failed.

Return type str

Raises **IOError** – if the loader script or manifest can't be written

loaded_packages()
Return list of loaded *package.InstalledPackage*.

match_source_packages(*pkg_path*)
Return a list of *package.Package* that match a given path.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

modified_config_files(*installed_pkg*)
Return a list of package config files that the user has modified.

Parameters **installed_pkg** (*package.InstalledPackage*) – the installed package to check for whether it has installed any Bro scripts.

Returns tuples that describe the modified config files. The first element is the config file as specified in the package metadata (a file path relative to the package's root directory). The second element is an absolute file system path to where that config file is currently installed.

Return type list of (str, str)

package_build_log(*pkg_path*)
Return the path to the package manager's build log for a package.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

package_versions(*installed_package*)
Returns a list of version number tags available for a package.

Parameters **installed_package** (*package.InstalledPackage*) – the package for which version number tags will be retrieved.

Returns the version number tags.

Return type list of str

pin(*pkg_path*)
Pin a currently installed package to the currently installed version.

Pinned packages are never upgraded when calling *upgrade()*.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

Returns None if no matching installed package could be found, else the installed package that was pinned.

Return type *package.InstalledPackage*

Raises **IOError** – when the manifest file can't be written

refresh_installed_packages()
Fetch latest git information for installed packages.

This retrieves information about outdated packages, but does not actually upgrade their installations.

Raises **IOError** – if the package manifest file can't be written

refresh_source(*name*, *aggregate=False*, *push=False*)
Pull latest git information from a package source.

This makes the latest pre-aggregated package metadata available or performs the aggregation locally in order to push it to the actual package source. Locally aggregated data also takes precedence over the source's pre-aggregated data, so it can be useful in the case the operator of the source does not update their pre-aggregated data at a frequent enough interval.

Parameters

- **name** (*str*) – the name of the package source. E.g. the same name used as a key to *add_source()*.
- **aggregate** (*bool*) – whether to perform a local metadata aggregation by crawling all packages listed in the source's index files.
- **push** (*bool*) – whether to push local changes to the aggregated metadata to the remote package source. If the *aggregate* flag is set, the data will be pushed after the aggregation is finished.

Returns an empty string if no errors occurred, else a description of what went wrong.

Return type str

remove(*pkg_path*)
Remove an installed package.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

Returns True if an installed package was removed, else False.

Return type bool

Raises

- **IOError** – if the package manifest file can't be written
- **OSError** – if the installed package's directory can't be deleted

save_temporary_config_files(*installed_pkg*)
Return a list of temporary package config file backups.

Parameters **installed_pkg** (*package.InstalledPackage*) – the installed package to save temporary config file backups for.

Returns tuples that describe the config files backups. The first element is the config file as specified in the package metadata (a file path relative to the package's root directory). The second element is an absolute file system path to where that config file has been copied. It should be considered temporary, so make use of it before doing any further operations on packages.

Return type list of (str, str)

source_packages()
Return a list of *package.Package* within all sources.

test(*pkg_path*, *version=''*)
Test a package.

Parameters

- **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".
- **version** (*str*) – if not given, then the latest git version tag is used (or if no version tags exist, the "master" branch is used). If given, it may be either a git version tag or a git branch name.

Returns a tuple containing an error message string, a boolean indicating whether the tests passed, as well as a path to the directory in which the tests were run. In the case where tests failed, the directory can be inspected to figure out what went wrong. In the case where the error message string is not empty, the error message indicates the reason why tests could not be run.

Return type (str, bool, str)

unbundle(*bundle_file*)
Installs all packages contained within a bundle.

Parameters **bundle_file** (*str*) – the path to the bundle to install.

Returns an empty string if the operation was successful, else an error message indicated what went wrong.

Return type str

unload(*pkg_path*)
Unmark an installed package as being "loaded".

The collection of "loaded" packages is a convenient way for Bro to more simply load a whole group of packages installed via the package manager.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

Returns True if a package is successfully unmarked as loaded.

Return type bool

Raises **IOError** – if the loader script or manifest can't be written

unpin(*pkg_path*)
Unpin a currently installed package and allow it to be upgraded.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

Returns None if no matching installed package could be found, else the installed package that was unpinned.

Return type *package.InstalledPackage*

Raises **IOError** – when the manifest file can't be written

upgrade(*pkg_path*)
Upgrade a package to the latest available version.

Parameters **pkg_path** (*str*) – the full git URL of a package or the shortened path/name that refers to it within a package source. E.g. for a package source called "bro" with package named "foo" in `alice/bro-pkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "bro/alice/foo".

Returns an empty string if package upgrade succeeded else an error string explaining why it failed.

Return type str

Raises **IOError** – if the manifest can't be written

validate_dependencies(*requested_packages*, *ignore_installed_packages=False*, *ignore_suggestions=False*)
Validates package dependencies.

Parameters

- **requested_packages** (*list of (str, str)*) – a list of (package name or git URL, version) string tuples validate. If the version string is empty, the latest available version of the package is used.
- **ignore_installed_packages** (*bool*) – whether the dependency analysis should consider installed packages as satisfying dependency requirements.
- **ignore_suggestions** (*bool*) – whether the dependency analysis should consider installing dependencies that are marked in another package's 'suggests' metadata field.

Returns the first element of the tuple is an empty string if dependency graph was successfully validated, else an error string explaining what is invalid. In the case it was validated, the second element is a list of tuples where the first elements are dependency packages that would need to be installed in order to satisfy the dependencies of the requested packages (it will not include any packages that are already installed or that are in the *requested_packages* argument). The second element of tuples in the list is a version string of the associated package that satisfies dependency requirements. The third element of the tuples in the list is a boolean value indicating whether the package is included in the list because it's merely suggested by another package.

Return type (str, list of (*package.PackageInfo*, str, bool))

5.2 bropkg.package module

A module with various data structures used for interacting with and querying the properties and status of Bro packages.

class bropkg.package.**InstalledPackage**(*package*, *status*)

Bases: object

An installed package and its current status.

package
the installed package

Type *Package*

status
the status of the installed package

Type *PackageStatus*

bropkg.package.**METADATA_FILENAME = 'bro-pkg.meta'**
The name of files used by packages to store their metadata.

class bropkg.package.**Package**(*git_url*, *source=''*, *directory=''*, *metadata=None*, *name=None*, *canonical=False*)

Bases: object

A Bro package.

This class contains properties of a package that are defined by the package git repository itself and the package source it came from.

git_url
the git URL which uniquely identifies where the Bro package is located

Type str

name
the canonical name of the package, which is always the last component of the git URL path

Type str

source
the package source this package comes from, which may be empty if the package is not a part of a source (i.e. the user is referring directly to the package's git URL).

Type str

directory
the directory within the package source where the `bro-pkg.index` containing this package is located. E.g. if the package source has a package named "foo" declared in `alice/bro-pkg.index`, then *dir* is equal to "alice". It may also be empty if the package is not part of a package source or if it's located in a top-level `bro-pkg.index` file.

Type str

metadata
the contents of the package's `bro-pkg.meta` file. If the package has not been installed then this information may come from the last aggregation of the source's `aggregate.meta` file (it may not be accurate/up-to-date).

Type dict of str -> str

aliases()
Return a list of package name aliases.

The canonical one is listed first.

dependencies(*field='depends'*)
Returns a dictionary of dependency -> version strings.

The keys indicate the name of a package (shorthand name or full git URL) or just 'bro' to indicate a dependency on a particular bro version.

The values indicate a semantic version requirement.

If the dependency field is malformed (e.g. number of keys not equal to number of values), then None is returned.

matches_path(*path*)
Return whether this package has a matching path/name.

E.g for a package with *qualified_name()* of "bro/alice/foo", the following inputs will match: "foo", "alice/foo", "bro/alice/foo"

name_with_source_directory()
Return the package's within its package source.

E.g. for a package source with a package named "foo" in `alice/bro-pkg.index`, this method returns "alice/foo". If the package has no source or sub-directory within the source, then just the package name is returned.

qualified_name()
Return the shortest name that qualifies/distinguishes the package.

If the package is part of a source, then this returns "source_name/*name_with_source_directory()*", else the package's git URL is returned.

short_description()
Return a short description of the package.

This will be the first sentence of the package's 'description' field and may return results from the source's aggregated metadata if the package has not been installed yet.

tags()
Return a list of keyword tags associated with the package.

This will be the contents of the package's *tags* field and may return results from the source's aggregated metadata if the package has not been installed yet.

user_vars()
Returns a list of (str, str, str) from metadata's 'user_vars' field.

Each entry in the returned list is a the name of a variable, it's value, and its description.

If the 'user_vars' field is not present, an empty list is returned. If it is malformed, then None is returned.

class bropkg.package.**PackageInfo**(*package=None*, *status=None*, *metadata=None*, *versions=None*, *metadata_version=''*, *invalid_reason=''*, *version_type=''*)

Bases: `object`

Contains information on an arbitrary package.

If the package is installed, then its status is also available.

package
the relevant Bro package

Type *Package*

status
this attribute is set for installed packages

Type *PackageStatus*

metadata
the contents of the package's `bro-pkg.meta` file

Type dict of str -> str

versions
a list of the package's availabe git version tags

Type list of str

metadata_version
the package version that the metadata is from

version_type
either 'version', 'branch', or 'commit' to indicate whether the package info/metadata was taken from a release version tag, a branch, or a specific commit hash.

invalid_reason
this attribute is set when there is a problem with gathering package information and explains what went wrong

Type str

aliases()
Return a list of package name aliases.

The canonical one is listed first.

best_version()
Returns the best/latest version of the package that is available.

If the package has any git release tags, this returns the highest one, else it returns the 'master' branch.

dependencies(*field='depends'*)
Returns a dictionary of dependency -> version strings.

The keys indicate the name of a package (shorthand name or full git URL) or just 'bro' to indicate a dependency on a particular bro version.

The values indicate a semantic version requirement.

If the dependency field is malformed (e.g. number of keys not equal to number of values), then None is returned.

short_description()
Return a short description of the package.

This will be the first sentence of the package's 'description' field.

tags()
Return a list of keyword tags associated with the package.

This will be the contents of the package's *tags* field.

user_vars()
Returns a list of (str, str, str) from metadata's 'user_vars' field.

Each entry in the returned list is a the name of a variable, it's value, and its description.

If the 'user_vars' field is not present, an empty list is returned. If it is malformed, then None is returned.

class bropkg.package.**PackageStatus**(*is_loaded=False*, *is_pinned=False*, *is_outdated=False*, *tracking_method=None*, *current_version=None*, *current_hash=None*)
Bases: object

The status of an installed package.

This class contains properties of a package related to how the package manager will operate on it.

is_loaded
whether a package is marked as "loaded".

Type bool

is_pinned
whether a package is allowed to be upgraded.

Type bool

is_outdated
whether a newer version of the package exists.

Type bool

tracking_method
either "branch", "version", or "commit" to indicate (respectively) whether package upgrades should stick to a git branch, use git version tags, or do nothing because the package is to always use a specific git commit hash.

Type str

current_version
the current version of the installed package, which is either a git branch name or a git version tag.

Type str

current_hash
the git sha1 hash associated with installed package's current version/commit.

Type str

bropkg.package.**aliases**(*metadata_dict*)
Return a list of package aliases found in metadata's 'aliases' field.

bropkg.package.**canonical_url**(*path*)
Returns the url of a package given a path to its git repo.

bropkg.package.**dependencies**(*metadata_dict*, *field='depends'*)
Returns a dictionary of (str, str) based on metadata's dependency field.

The keys indicate the name of a package (shorthand name or full git URL) or just 'bro' to indicate a dependency on a particular bro version.

The values indicate a semantic version requirement.

If the dependency field is malformed (e.g. number of keys not equal to number of values), then None is returned.

bropkg.package.**name_from_path**(*path*)
Returns the name of a package given a path to its git repository.

bropkg.package.**short_description**(*metadata_dict*)
Returns the first sentence of the metadata's 'desciption' field.

bropkg.package.**tags**(*metadata_dict*)
Return a list of tag strings found in the metadata's 'tags' field.

bropkg.package.**user_vars**(*metadata_dict*)
Returns a list of (str, str, str) from metadata's 'user_vars' field.

Each entry in the returned list is a the name of a variable, it's value, and its description.

If the 'user_vars' field is not present, an empty list is returned. If it is malformed, then None is returned.

5.3 bropkg.source module

A module containing the definition of a "package source": a git repository containing a collection of `bro-pkg.index` files. These are simple INI files that can describe many Bro packages. Each section of the file names a Bro package along with the git URL where it is located and metadata tags that help classify/describe it.

bropkg.source.**AGGREGATE_DATA_FILE = 'aggregate.meta'**
The name of the package source file where package metadata gets aggregated.

bropkg.source.**INDEX_FILENAME = 'bro-pkg.index'**
The name of package index files.

class bropkg.source.**Source**(*name*, *clone_path*, *git_url*)
Bases: `object`

A Bro package source.

This class contains properties of a package source like its name, remote git URL, and local git clone.

name
The name of the source as given by a config file key in it's `[sources]` section.

Type str

git_url
The git URL of the package source.

Type str

clone
The local git clone of the package source.

Type git.Repo

package_index_files()
Return a list of paths to package index files in the source.

packages()
Return a list of *package.Package* in the source.

CHAPTER 6

Developer's Guide

This a guide for developers working on the Zeek Package Manager itself.

6.1 Versioning/Releases

After making a commit to the *master* branch, you can use the **`update-changes`** script in the zeek-aux repository to automatically adapt version numbers and regenerate the **`bro-pkg`** man page. Make sure to install the *documentation dependencies* before using it.

Releases are hosted at PyPi. To build and upload a release:

1. Finalize the git repo tag and version with `update-changes -R <version>` if not done already.
2. Upload the distribution (you will need the credentials for the 'bro' account on PyPi):

```
$ make upload
```

6.2 Documentation

Documentation is written in reStructuredText (reST), which Sphinx uses to generate HTML documentation and a man page.

6.2.1 Dependencies

To build documentation locally, find the requirements in `requirements.txt`:

```
# Requirements for general bro-pkg usage
GitPython
semantic_version
configparser
```

(continues on next page)

(continued from previous page)

```
btest
# Requirements for development (e.g. building docs)
Sphinx
sphinxcontrib-napoleon
sphinx_rtd_theme
```

They can be installed like:

pip install -r requirements.txt

6.2.2 Local Build/Preview

Use the `Makefile` targets `make html` and `make man` to build the HTML and man page, respectively. To view the generated HTML output, open `doc/_build/index.html`. The generated man page is located in `doc/man/bro-pkg.1`.

If you have also installed **`sphinx-autobuild`** (e.g. via **`pip`**), there's a `Makefile` target, `make livehtml`, you can use to help preview documentation changes as you edit the reST files.

6.2.3 Remote Hosting

The GitHub repository has a webhook configured to automatically rebuild the HTML documentation hosted at Read the Docs whenever a commit is pushed.

6.2.4 Style Conventions

The following style conventions are (generally) used.

Documentation Subject	reST Markup	Preview
File Path	`:file:`path``	path
File Path w/ Substitution	`:file:`{<replace_me>}/path``	*<replace_me>*/path
OS-Level Commands	`:command:`cmd``	**cmd**
Program Names	`:program:`prog``	**prog**
Environment Variables	`:envvar:`VAR``	VAR
Literal Text (e.g. code)	``` ``code`` ```	code
Substituted Literal Text	`:samp:`code {<replace_me>}``	code *<replace_me>*
Variable/Type Name	`` `x` ``	*x*
INI File Option	`` `name` ``	*name*

Python API docstrings roughly follow the Google Style Docstrings format.

Python Module Index

b

Index

V