The Zeek Package Manager makes it easy for Zeek users to install and manage third party scripts as well as plugins for Zeek and ZeekControl. 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 3.6+
- 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

Note that following the zkg Installation process via pip3 will automatically install its dependencies for you.

1.2 Installation

- Zeek 4.0.0 or greater comes with a bundled zkg that is included as part of its installation. This is often the easiest choice since it comes pre-configured to work correctly for that particular Zeek installation and some Basic Configuration steps can be skipped. The directions to build and install Zeek from source are here: https://docs.zeek.org/en/current/install/install.html

Note that this method does require independent installation of zkg’s dependencies, which is usually easiest to do via pip3:

```sh
$ pip3 install gitpython semantic-version
```

- To install the latest release of zkg on PyPI:

```sh
$ pip3 install zkg
```

- To install the latest Git development version of zkg:
$ pip3 install git+git://github.com/zeek/package-manager@master

Note: If not using something like virtualenv to manage Python environments, the default user script directory is ~/.local/bin and you may have to modify your PATH to search there for zkg.

1.3 Basic Configuration

zkg supports four broad approaches to managing Zeek packages:

- Keep package metadata in $HOME/.zkg/ and maintain Zeek-relevant package content (such as scripts and plugins) in the Zeek installation tree. This is zkg’s "traditional" approach.
- Keep all state and package content within the Zeek installation tree. Zeek 4’s bundled zkg installation provides this by default. If you use multiple Zeek installations in parallel, this approach allows you to install different sets of Zeek packages with each Zeek version.
- Keep all state and package content in $HOME/.zkg/. This is the preferred approach when you’re running zkg and zeek as different users. zkg’s --user mode enables this approach.
- Custom configurations where you select your own state and content locations.

After installing via pip3, but not when using the zkg that comes pre-bundled with a Zeek installation, additional configuration is still required in the form of running a zkg autoconfig command, but in either case, do read onward to get a better understanding of how the package manager is configured, what directories it uses, etc.

To configure zkg for use with a given Zeek installation, make sure that the zeek-config script that gets installed with zeek is in your PATH. Then, as the user you want to run zkg with, do:

$ zkg autoconfig

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

- `script_dir`: set to the location of Zeek’s site scripts directory (e.g. <zeek_install_prefix>/share/zeek/site)
- `plugin_dir`: set to the location of Zeek’s default plugin directory (e.g. <zeek_install_prefix>/lib/zeek/plugins)
- `bin_dir`: set to the location where zkg installs executables that packages provide (e.g., <zeek_install_prefix>/bin).
- `zeek_dist`: set to the location of Zeek’s source code. If you didn’t build/install Zeek from source code, this field will not be set, but it’s only needed if you plan on installing packages that have uncompiled Zeek plugins.

With those settings, the package manager will install Zeek scripts, Zeek plugins, and ZeekControl plugins into directories where zeek and zeekctl will, by default, look for them. ZeekControl clusters will also automatically distribute installed package scripts/plugins to all nodes.

Note: If your Zeek installation is owned by "root" and you intend to run zkg as a different user, you have two options.

First, you can use zkg’s user mode (zkg --user). In user mode, zkg consults $HOME/.zkg/config for configuration settings. Creating this config file in user mode (zkg --user autoconfig) ensures that all state and content directories reside within $HOME/.zkg/. zkg reports according environment variables in the output of zkg --user env.
Second, you can grant "write" access to the directories specified by script_dir, plugin_dir, and bin_dir; perhaps using something like:

```
$ sudo chgrp $USER $(zeek-config --site_dir) $(zeek-config --plugin_dir) $(zeek-config --prefix)/bin
$ sudo chmod g+rwX $(zeek-config --site_dir) $(zeek-config --plugin_dir) $(zeek-config --prefix)/bin
```

The final step is to edit your site/local.zeek. If you want to have Zeek 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 zkg.

## 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 ~/.zkg 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 zeek or zeekctl will not detect scripts/plugins in those locations without additional configuration.

When using non-standard location, follow these steps to integrate with zeek and zeekctl:

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

  ```
  $ `zkg env`
  ```

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

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

  ```
  $ zkg config script_dir
  ```

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

  ```
  $ zkg config plugin_dir
  ```

  And set the SitePluginPath option in zeekctl.cfg based on the output you see.

- To have your shell find executables that packages provide, update your PATH:

  ```
  $ export PATH=$(zkg config bin_dir):$PATH
  ```

  (Executing `zkg env`, as described above, includes this already.)

## 1.5 Usage

Check the output of zkg –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 `zkg` 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 `zkg` treats each situation.

1.5.2 Offline Usage

It’s common to have limited network/internet access on the systems where Zeek is deployed. To accommodate those scenarios, `zkg` 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:

```
$ zkg install <package name>
```

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

```
$ zkg bundle zeek-packages.bundle
```

Then transfer `zeek-packages.bundle` to the Zeek deployment management host. For Zeek clusters using Zeek-Control, this will be the system acting as the "manager" node. Then on that system (assuming it already has `zkg` installed and configured):

```
$ zkg unbundle zeek-packages.bundle
```

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

```
$ zeekctl deploy
```
A command-line package manager for Zeek.

```
          [--extra-source NAME=URL]
          {test,install,bundle,unbundle,remove,purge,refresh,upgrade,load,unload,pin,
           unpin,list,search,info,config,autoconfig,env,create,template}

Options:

  --version                     show program’s version number and exit

  --configfile  Path to Zeek Package Manager config file. Precludes --user.

  See Config File.

  --user=False                  Store all state in user's home directory. Precludes --configfile.

  --verbose=0, -v=0             Increase program output for debugging. Use multiple times for more output
                                 (e.g. -vvv).

  --extra-source                Add an extra source.

Environment Variables:

  ZKG_CONFIG_FILE: Same as --configfile option, but has less precedence.
```
## 2.1 Commands

### 2.1.1 test

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

```bash
usage: zkg 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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/foo", "zeek/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 default branch, like "main" or "master".

### 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).

```bash
```

**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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/foo", "zeek/alice/foo".

**Options:**

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

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

```
usage: zkg remove [-h] [-f] [-N] 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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/foo", "zeek/alice/foo".

Options:

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

2.1.4 purge

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

```
usage: zkg purge [-h] [-f]
```

Options:

- `--force`  `False`
  Skip the confirmation prompt.
2.1.5 bundle

This command creates a bundle file containing a collection of Zeek 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:
```
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=off`: Skip the confirmation prompt.
- `--nodeps=off`: 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=off`: 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:
```
usage: zkg unbundle [-h] [--replace] [--force] [--user-var NAME=VAL]
filename.bundle
```

Positional arguments:
- `filename.bundle`: The path of the bundle file to install.

Options:
- `--replace=off`: Using this flag first removes all installed packages before then installing the packages from the bundle.
- `--force=off`: Don’t prompt for confirmation or user variables.
- `--user-var`: A user variable assignment. This avoids prompting for input and lets you provide a value when using --force. Use repeatedly as needed for multiple values.
2.1.7 refresh

Retrieves 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).

```
                 [--sources SOURCES [SOURCES ...]]
```

Options:

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

--fail-on-aggregate-problems=False  When using --aggregate, exit with error when any packages trigger metadata problems. Normally such problems only cause a warning.

--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

Upgrades 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.

```
                  [--user-var NAME=VAL]
                  [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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/foo", "zeek/alice/foo".

Options:

--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.
--force=False       Don’t prompt for confirmation or user variables.
--user-var          A user variable assignment. This avoids prompting for input and lets you provide a value when using --force. Use repeatedly as needed for multiple values.

2.1.9 load

The Zeek Package Manager keeps track of all packages that are marked as "loaded" and maintains a single Zeek script that, when loaded by Zeek (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: zkg load [-h] [--nodeps] package [package ...]
```

Positional arguments:

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

Options:

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

2.1.10 unload

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

```
```

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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/foo", "zeek/alice/foo".
```

Options:

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

2.1.11 pin

Pinned packages are ignored by the upgrade command.
usage: zkg 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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/foo", "zeek/alice/foo".

2.1.12 unpin

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

usage: zkg 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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/foo", "zeek/alice/foo".

2.1.13 list

Outputs a list of packages that match a given category.

usage: zkg list [-h] [--nodesc] ([all,installed,not_installed,loaded,unloaded,outdated])

Positional arguments:

category  Package category used to filter listing.

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: zkg 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: zkg 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 "zeek" that has a package named "foo" located in either "alice/zkg.index" or "alice/bro-pkg.index", the following names work: "foo", "alice/oo", "zeek/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 default branch, like "main" or "master", 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: zkg config [-h]
    [all,sources,user_vars,state_dir,script_dir,plugin_dir,bin_dir,
     →zeek_dist,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, bin_dir, zeek_dist, bro_dist

### 2.1.17 autoconfig

The output of this command is a valid package manager config file that is generated by using the `zeek-config` script that is installed along with Zeek. It is the suggested configuration to use for most Zeek installations. For this command to work, the `zeek-config` (or `bro-config`) script must be in `PATH`, unless the `--user` option is given, in which case this creates a config that does not touch the Zeek installation.

```
usage: zkg autoconfig [-h] [--force]
```

Options:

```
--force=False
```
Skip any confirmation prompt.

### 2.1.18 env

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

```
usage: zkg env [-h]
```

### 2.1.19 create

This command creates a new Zeek package in the directory provided via `--packagedir`. If this directory exists, `zkg` will not modify it unless you provide `--force`.

```
usage: zkg create [-h] --packagedir DIR [--version VERSION]
    [--features FEATURE [FEATURE ...]] [--template URL]
    [--force] [--user-var NAME=VAL]
```

Options:

```
--packagedir
```
Output directory into which to produce the new package. Required.

```
--version
```
The template version to use. A version tag, branch name, or commit hash may be specified here. If `--template` refers to a local git repo with a working tree, then `zkg` uses it as-is and the version is ignored. The default for other cases is to use the latest version tag, or if a template has none, the default branch, like "main" or "master".
Zeek Package Manager Documentation, Release 2.11.0

---features
Additional features to include in your package. Use the `template info` command for information about available features.

---template
By default, zkg uses its own package template. This makes it select an alternative.

--force=False
Don’t prompt for confirmation or user variables.

--user-var
A user variable assignment. This avoids prompting for input and lets you provide a value when using --force. Use repeatedly as needed for multiple values.

2.1.20 template info
This command shows versions and supported features for a given package.

usage: zkg template info [-h] [--json] [--jsonpretty SPACES] [--version VERSION] [URL]

Positional arguments:
  URL
  URL of a package template repository, or local path to one. When not provided, the configured default template is used.

Options:
  --json=False
  Output template information as JSON.
  --jsonpretty
  Optional number of spaces to indent for pretty-printed JSON output.
  --version
  The template version to report on. A version tag, branch name, or commit hash may be specified here. If the selected template refers to a local git repo, the version is ignored. The default for other cases is to use the latest version tag, or if a template has none, the default branch, like "main" or "master".

2.2 Config File
The zkg command-line tool uses an INI-format config file to allow users to customize their Package Sources, Package installation paths, Zeek executable/source paths, and other zkg options.

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

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

[sources]
# The default package source repository from which zkg fetches
```

(continues on next page)
# 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. The git URL may also use a
# suffix like "@branch-name" where "branch-name" is the name of a real
# branch to checkout (as opposed to the default branch, which is typically
# "main" or "master"). You can override the package source zkg puts
# in new config files (e.g. "zkg autoconfig") by setting the
# ZKG_DEFAULT_SOURCE environment variable.
zeek = https://github.com/zeek/packages

[paths]

# Directory where source repositories are cloned, packages are
# installed, and other package manager state information is
# maintained. If left blank or with --user this defaults to
# $HOME/.zkg. In Zeek-bundled installations, it defaults to
# <zeek_install_prefix>/var/lib/zkg/.
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 zkg.meta
# (or legacy bro-pkg.meta) file there.
# If left blank or with --user this defaults to <state_dir>/script_dir.
# In Zeek-bundled installations, it defaults to
# <zeek_install_prefix>/share/zeek/site.
# If you decide to change this location after having already
# installed packages, zkg will automatically relocate them
# the next time you run any zkg 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 zkg.meta
# (or legacy bro-pkg.meta) file there.
# If left blank or with --user this defaults to <state_dir>/plugin_dir.
# In Zeek-bundled installations, it defaults to
# <zeek_install_prefix>/lib/zeek/plugins.
# If you decide to change this location after having already
# installed packages, zkg will automatically relocate them
# the next time you run any zkg command.
plugin_dir =

# The directory where executables from packages are linked into upon
# installation. If left blank or with --user this defaults to <state_dir>/bin.
# In Zeek-bundled installations, it defaults to <zeek_install_prefix>/bin.
# If you decide to change this location after having already
# installed packages, zkg will automatically relocate them
# the next time you run any zkg command.
bin_dir =

# The directory containing Zeek distribution source code. This is only
# needed when installing packages that contain Zeek plugins that are
# not pre-built. The legacy name of this option is "bro_dist".
zeek_dist =
[templates]

# The URL of the package template repository that the "zkg create" command
# will instantiate by default.
default = https://github.com/zeek/package-template

[user_vars]

# For any key in this section that is matched for value interpolation
# in a package's zkg.meta (or legacy 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.
A Zeek package may contain Zeek scripts, Zeek plugins, or ZeekControl 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 `zkg.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 Zeek, the package manager, or other packages.

**Note:** `zkg.meta` is the canonical metadata file name used since `zkg v2.0`. The previous metadata file name of `bro-pkg.meta` is also accepted when no `zkg.meta` exists.

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

```
@load foo
```

### 3.1 Bootstrapping packages with `zkg`

The easiest way to start a new Zeek package is via `zkg` itself: its `zkg create` command lets you generate new Zeek packages from the command line.

This functionality is available since since `zkg v2.9`. See the *Walkthroughs* section for step-by-step processes that show how to manually create packages (e.g. perhaps when using older `zkg` versions).
3.1.1 Concepts

*zkg* instantiates new packages from a *package template*. Templates are standalone git repositories. The URL of *zkg*’s default template is [https://github.com/zeek/package-template](https://github.com/zeek/package-template), but you can provide your own.

**Note:** At *zkg* configuration time, the `ZKG_DEFAULT_TEMPLATE` environment variable lets you override the default, and the `--template` argument to *zkg create* allows overrides upon instantiation. You can review the template *zkg* will use by default via the *zkg config* command’s output.

A template provides a basic *package* layout, with optional added *features* that enhance the package. For example, the default template lets you add a native-code plugin and support for GitHub actions.

Templates are parameterized via *user variables*. These variables provide the basic configuration required when instantiating the template, for example to give the package a name. A template uses resolved user variables to populate internal *parameters* that the template requires. Think of parameters as derivatives of the user variables, for example to provide different capitalizations or suffixes.

A template operates as a *zkg* plugin, including runnable Python code. This code has full control over how a package gets instantiated, defining required user variables and features, and possibly customizing content production.

3.1.2 The *create* command

When using the *zkg create* command, you specify an output directory for the new package tree, name the features you’d like to add, and optionally define user variables. *zkg* will prompt you for any variables it still needs to resolve, and guides you through the package creation. A basic invocation might look as follows:

```
$ zkg create --packagedir foobar --feature plugin
```

"package-template" requires a "name" value (the name of the package, e.g. "FooBar"):  
name: Foobar  
"package-template" requires a "namespace" value (a namespace for the package, e.g. "MyOrg"):  
namespace: MyOrg

The resulting package now resides in the `foobar` directory. Unless you provide `--force`, *zkg* will not overwrite an existing package. When the requested output directory exists, it will prompt for permission to delete the existing directory.

After instantiation, the package is immediately installable via *zkg*. You’ll see details of how it got generated in its initial commit, and the newly minted *zkg.meta* has details of the provided user variables:

```
$ cat foobar/zkg.meta
...
[template]
source = package-template
version = master
zkg_version = 2.8.0
features = plugin

[template_vars]
name = Foobar
namespace = MyOrg
```

This information is currently informational only, but in the future will enable baselining changes in package templates to assist with package modernization.
To keep templates in sync with `zkg` versions, templates employ semantic API versioning. An incompatible template will refuse to load and lead to an according error message. Much like Zeek packages, templates support git-level versioning to accommodate compatibility windows.

See the output of `zkg create --help` for a complete summary of the available options.

### 3.1.3 Obtaining information about a template

The best source for the capabilities of a template is its documentation, but to get a quick overview of a given template’s features and user variables, consider the `zkg template info` command, which summarizes a template in plain text, or in JSON when invoked with the `--json` argument.

### 3.2 Walkthroughs

For historical reference, the following sections cover manual ways of establishing Zeek packages.

#### 3.2.1 Pure Zeek Script Package

1. Create a git repository:

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

2. Create a package metadata file, `zkg.meta`:

   ```bash
   $ echo '[package]' > zkg.meta
   ```

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

   ```bash
   $ echo 'event zeek_init() { print "foo is loaded"; }' > __load__.zeek
   ```

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

   ```bash
   $ mkdir scripts && mv __load__.zeek scripts
   $ echo 'script_dir = scripts' >> zkg.meta
   ```

5. Commit everything to git:

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

6. (Optional) Test that Zeek correctly loads the script after installing the package with `zkg`:

   ```bash
   $ zkg install .
   $ zeek foo
   $ zkg remove .
   ```

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

See `Zeek Scripting` for more information on developing Zeek scripts.

#### 3.2.2 Binary Zeek Plugin Package

See `Zeek Plugins` for more complete information on developing Zeek plugins, though the following steps are the essentials needed to create a package.
1. Create a plugin skeleton using `aux*/zeek-aux/plugin-support/init-plugin` from Zeek's source distribution:

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

2. Create a git repository

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

3. Create a package metadata file, `zkg.meta`:

```
[package]
script_dir = scripts/Demo/Rot13
build_command = ./configure && make
```

**Note:** See [Supporting Older Bro Versions](#) for notes on configuring packages to support Bro 2.5 or earlier.

4. Add example script code:

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

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

```
module Demo;

function rot13%(s: string%) : string {
  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 Zeek correctly loads the plugin after installing the package with `zkg`:

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

8. (Optional) [Create a release version tag](#).
3.2.3 ZeekControl Plugin Package

1. Create a git repository:

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

2. Create a package metadata file, zkg.meta:

   ```bash
   $ echo '[package]' > zkg.meta
   ```

3. Create an example ZeekControl plugin, foo.py:

   ```python
   import ZeekControl.plugin
   from ZeekControl import config

   class Foo(ZeekControl.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:

   ```bash
   $ echo 'plugin_dir = .' >> zkg.meta
   ```

5. Commit everything to git:

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

6. (Optional) Test that ZeekControl correctly loads the plugin after installing the package with zkg:

   ```bash
   $ zkg install .
   $ zeekctl
   $ zkg remove .
   ```

7. (Optional) Create a release version tag.

See ZeekControl Plugins for more information on developing ZeekControl plugins.

If you want to distribute a ZeekControl plugin along with a Zeek plugin in the same package, you may need to add the ZeekControl plugin’s python script to the `zeek_plugin_dist_files()` macro in the `CMakeLists.txt` of the Zeek plugin so that it gets copied into build/ along with the built Zeek 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 Zeek plugin and the ZeekControl plugin.

3.2.4 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/zeek/packages

3.3 Package Metadata

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

3.3.1 description field

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

3.3.2 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.

3.3.3 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 (zkg 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:
### 3.3.4 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 `zkg search` command will inspect these tags. An example `zkg.meta` using tags:

```yaml
[package]
tags = zeek plugin, zeekctl plugin, scan detection, intel
```

#### Suggested Tags

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

- zeek scripting
  - conn
  - intel
  - geolocation
  - file analysis
  - sumstats, summary statistics
  - input
  - log, logging
  - notices
- `<network protocol name>`
- `<file format name>`
- signatures
- zeek plugin
  - protocol analyzer
  - file analyzer
  - bifs
  - packet source
  - packet dumper
  - input reader
  - log writer
- zeekctl plugin
3.3.5 script_dir field

The `script_dir` field is a path relative to the root of the package that contains a file named `__load__.zeek` and possibly other Zeek 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 `<zeek_install_prefix>/share/zeek/site`).

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

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

An example `zkg.meta`:

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

For a `zkg.meta` that looks like the above, the package should have a file called `scripts/__load__.zeek`.

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

3.3.6 plugin_dir field

The `plugin_dir` field is a path relative to the root of the package that contains either pre-built Zeek Plugins, ZeekControl Plugins, or both.

An example `zkg.meta`:

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

For the above example, Zeek and ZeekControl will load any plugins found in the installed package’s `plugins` directory.

If the `plugin_dir` field is not present in `zkg.meta`, it defaults to a directory named `build/` at the top-level of the package. This is the default location where Zeek 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 Zeek plugin. The default CMake skeleton for Zeek 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.

3.3.7 executables field

The `executables` field is a whitespace-delimited list of shell scripts or other executables that the package provides. The package manager will make these executables available inside the user’s `bin_dir` directory as specified in the `config file`.

An example `zkg.meta`, if the Rot13 example plugin were also building an executable `a.out`:
The package manager makes executables available by maintaining symbolic links referring from `bin_dir` to the actual files.

Available since `bro-pkg v2.8`.

### 3.3.8 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 Zeek 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 `zkg.meta`:

```
[package]
script_dir = scripts/Demo/Rot13
build_command = ./configure && make
```

**Note:** See *Supporting Older Bro Versions* for notes on configuring packages to support Bro 2.5 or earlier.

The default CMake skeleton for Zeek 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 Zeek plugin and Zeek script components, the "plugin" part is always unconditionally loaded by Zeek, but the "script" components must either be explicitly loaded (e.g. `@load <package_name>`) or the package marked as `loaded`.

**Supporting Older Bro Versions**

Plugin skeletons generated before Bro v2.6 and also any packages that generally want to support such Bro versions need to pass an additional configuration option such as:

```
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, but a brief explanation is that it will expand to a path containing the Bro source-code on the user’s system. For newer versions of Bro, packages are able to work entirely with the installation path and don’t require original source code.

Also note that other various Zeek scripting and CMake infrastructure may have changed between Bro v2.6 and Zeek v3.0. So if you plan to support older version of Bro (before the Zeek rename), then you should keep an eye out for various things that got renamed. For example, the `zeek_init` event won’t exist in any version before Zeek v3.0, nor will any CMake macros that start with `zeek_plugin`.

---

**3.3. Package Metadata**

27
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 zkg 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. Note that `since zkg v2.0`, `zeek_dist` is the canonical name for `bro_dist` within the zkg config file, but either one means the same thing and should work. To support older versions of `bro-pkg`, you’d want to use `bro_dist` in package metadata files.

Besides the `bro_dist/zeek_dist` config keys, 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.

Another pre-defined config key is `package_base`, which points to the top-level directory where zkg stores all installed packages (i.e. clones of each package’s git repository). This can be used to gain access to the content of another package that was installed as a dependency. Note that `package_base` is only available `since zkg v2.3`

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.

3.3.9 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 zkg.meta:

```conf
[package]
build_command = ./configure --with-librdkafka=%(LIBRDKAFKA_ROOT)s --with-libdbus=--%(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:

```bash
$ zkg install zeek-test-package
The following packages will be INSTALLED:
    zeek/jsiwek/zeek-test-package (1.0.5)
Proceed? [Y/n] y
zeek/jsiwek/zeek-test-package asks for LIBRDKAFKA_ROOT (Path to librdkafka, --installation)? [/usr] /usr/local
Saved answers to config file: /Users/jon/.zkg/config
Installed "zeek/jsiwek/zeek-test-package" (master)
Loaded "zeek/jsiwek/zeek-test-package"
```
The `zkg` 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 `zkg` 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 user may also provide `user_vars` via `--user-var NAME=VAL` command-line arguments. These arguments are given priority over environment variables, which in turn take precedence over any values in the user’s package manager config file.

Available since bro-pkg v1.1.

### 3.3.10 `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 `zkg.meta`:

```plaintext
[pkg]
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.

### 3.3.11 `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, `zkg` can instead save a backup and possibly prompt the user to review the differences.

An example `zkg.meta`:

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

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`.

### 3.3.12 `depends` field

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

An example `zkg.meta`:

```plaintext
[package]
depends =
  zeek >=2.5.0
  foo *
  https://github.com/zeek/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 *zeek*, *zkg*, *bro*, *bro-pkg*, a full git URL of the package, or a *package shorthand name*.

- The special *zeek* and *bro* dependencies refers not to a package, but the version of Zeek that the package requires in order to function. If the user has *zeek-config* or *bro-config* in their PATH when installing/upgrading a package that specifies a *zeek* or *bro* dependency, then *zkg* will enforce that the requirement is satisfied.

**Note:** In this context, *zeek* and *bro* mean the same thing – the later is maintained for backwards compatibility while the former became available since zkg v2.0.

- The special *zkg* and *bro-pkg* dependencies 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:** This feature itself, via use of a *bro-pkg* dependency, is only available since bro-pkg v1.2 while a *zkg* dependency is only recognized since zkg v2.0. Otherwise, *zkg* and *bro-pkg* mean the same thing in this context.

- 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 *zkg* 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 \textit{vX.Y.Z} format for its \textit{git version tags}, do not use the ‘v’ prefix in the version specifications here as that is not part of the semantic version.

### 3.3.13 external\_depends field

The \textit{external\_depends} field follows the same format as the \textit{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 \texttt{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 \texttt{bro-pkg v1.1}.

### 3.3.14 suggests field

The \textit{suggests} field follows the same format as the \textit{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 \textit{suggests} is functionally equivalent to a package in \textit{depends} except in the way it’s presented to users in various prompts during \texttt{zkg} operations. Users also have the option to ignore suggestions by supplying an additional \texttt{--nosuggestions} flag to \texttt{zkg} commands.

Available since \texttt{bro-pkg v1.3}.

### 3.4 Package Versioning

#### 3.4.1 Creating New Package Release Versions

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

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

The tag name may also be of the \textit{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:

```bash
$ 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 default branch (typically named \texttt{main} or \texttt{master}). Users will receive package updates differently depending on whether you decide to use release version tags or not. See the \textit{package upgrade process} documentation for more details on the differences.

#### 3.4.2 Package Upgrade Process

The \texttt{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 `zkg` 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 default branch (typically named `main` or `master`).

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.
zkg, by default, is configured to obtain packages from a single "package source", the Zeek Packages Git Repository, which is hosted by and loosely curated by the Zeek Team. However, users may configure zkg 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 Zeek Packages Git Repository organizes package index files hierarchically based on package author names such as alice/zkg.index or bob/zkg.index where alice and bob are usually GitHub usernames or some unique way of identifying the organization/person that maintains Zeek packages. However, a source is free to use a flat organization with a single, top-level zkg.index.

Note: The magic index file name of zkg.index is available since zkg v2.0. For compatibility purposes, the old index file name of bro-pkg.index is also still supported.

After creating a git repo for the package source and adding package index files to it, it’s ready to be used by zkg.

4.2 Package Index Files

Files named zkg.index (or the legacy bro-pkg.index) are used to describe the Zeek Packages found within the package source. They are simply a list of git URLs pointing to the git repositories of packages. For example:

```plaintext
https://github.com/zeek/foo
https://github.com/zeek/bar
https://github.com/zeek/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 committing/pushing those changes to the package source git repository. 

zkg 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 zkg 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 packages’ zkg.meta (and legacy bro-pkg.meta) files. The zkg refresh is used to perform the task. For example:

```
$ zkg 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 zkg 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.
This package defines a Python interface for installing, managing, querying, and performing other operations on Zeek 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 `zeekpkg` public interface:

### 5.1 `zeekpkg.manager` module

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

```python
class zeekpkg.manager.Manager
    (state_dir, script_dir, plugin_dir, zeek_dist='', user_vars=None, bin_dir='')

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`

- **zeek_dist**
  - path to the Zeek source code distribution. This is needed for packages that contain Zeek plugins that need to be built from source code.
Type str

**state_dir**
the directory where the package manager will 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 `zkg.meta` or `bro-pkg.meta`). 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 `zkg.meta` or `bro-pkg.meta`). Each package gets a subdirectory within *plugin_dir* associated with its name.

Type str

**bin_dir**
the directory where the package manager will link executables into that are provided by an installed package through *executables* (as given by its `zkg.meta` or `bro-pkg.meta`)

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 Zeek script named packages.zeek 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 Zeek __load__.zeek script which is just a symlink to autoload_script. It’s always located in a directory named packages, so as long as ZEEKPATH is configured correctly, @load packages will load all installed packages that have been marked as loaded.

Type str
class SourceAggregationResults (refresh_error=", package_issues=[])
Bases: object
The return value of a call to Manager.aggregate_source().

refresh_error
an empty string if no overall error occurred in the "refresh" operation, else a description of what wrong

Type str
package_issues
a list of reasons for failing to collect metadata per packages/repository. The first tuple element gives the repository URL in which the problem occurred and the second tuple element describes the failure.

Type list of (str, 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
aggregate_source (name, push=False)
Pull latest git info from a package source and aggregate metadata.

This is like calling refresh_source() with the aggregate arguments set to True.

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().
• push (bool) – whether to push local changes to the aggregated metadata to the remote package source.
Returns

the results of the refresh/aggregation.

Return type `Manager.SourceAggregationResults`

`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()`

Same as `zeek_plugin_path()`.

Using `zeek_plugin_path()` is preferred since this may later be deprecated.

`bropath()`

Same as `zeekpath()`.

Using `zeekpath()` is preferred since this may later be deprecated.

`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 "zeek" with package named "foo" in `alice/zkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/alice/foo".

get_installed_package_dependencies *(pkg_path)*

Return a set of tuples of dependent package names and their version number if `pkg_path` is 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 "zeek" with package named "foo" in `alice/zkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/alice/foo".

has_plugin *(installed_pkg)*

Return whether a `package.InstalledPackage` installed a plugin.

Parameters `installed_pkg` *(package.InstalledPackage)* – the installed package to check for whether it has installed a Zeek plugin.

Returns True if the package has installed a Zeek plugin.

Return type bool

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 Zeek scripts.

Returns True if the package has installed Zeek 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 "zeek" with package named "foo" in `alice/zkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/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 default branch like "main" or "master" 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 "zeek" with package named
"foo" in alice/zkg.index, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/alice/foo".

- **version** *(str)* – if not given, then the latest git version tag is installed (or if no version tags exist, the default branch like "main" or "master" 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

```python
def installed_package_dependencies()
    Return dict of 'package' -> dict of 'dependency' -> 'version'.
    Package-name / dependency-name / and version-requirement values are all strings.
```

```python
def installed_packages()
    Return list of package.InstalledPackage.
```

```python
def list_depender_pkgs(pkg_path)
    List of depender packages.
    If C depends on B and B depends on A, we represent the dependency chain as C -> B -> A. Thus, package C is dependent on A and B, while package B is dependent on just A. Example representation:

```python
    {
        'A': set(),
        'B': set([A, version_of_A])
        'C': set([B, version_of_B])
    }
```

Further, package A is a direct dependee for B (and implicitly for C), while B is a direct depender (and C is an implicit depender) for A.

**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 "zeek" with package named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/alice/foo".

**Returns** list of depender packages.

**Return type** list

```python
load(pkg_path)
    Mark an installed package as being "loaded".
    The collection of "loaded" packages is a convenient way for Zeek 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 "zeek" with package named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/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
load_with_dependencies(pkg_name, visited={})
Mark dependent (but previously installed) packages as being "loaded".

Parameters

• pkg_name(str) – name of the package.

• visited(set(str)) – set of packages visited along the recursive loading

Returns list of tuples containing dependent package name and whether it was marked as loaded
or else an explanation of why the loading failed.

Return type list(str, str)

loaded_package_states()
Save "loaded" state for all installed packages.

Returns dictionary of "loaded" status for installed packages

Return type dict

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 "zeek" with package
named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo",
"alice/foo", or "zeek/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 Zeek 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 "zeek" with package
named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo",
"alice/foo", or "zeek/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 “zeek” with package named “foo” in `alice/zkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/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 “zeek” with package named “foo” in `alice/zkg.index`, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/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

### `restore_loaded_package_states(saved_state)`

Restores state for installed packages.
Parameters

- **saved_state** (dict) – dictionary of saved "loaded" state for installed

- **packages** –

**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 "zeek" with package named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/alice/foo".

- **version** (str) – if not given, then the latest git version tag is used (or if no version tags exist, the default branch like "main" or "master" 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 Zeek 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 "zeek" with package
named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/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

unload_with_unused_dependers(pkg_name)
Unmark dependent (but previously installed packages) as being "loaded".

Parameters pkg_name (str) – name of the package.

Returns list of tuples containing dependent package name and whether it was marked as unloaded or else an explanation of why the unloading failed.

Return type list(str, str)

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 "zeek" with package named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/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 "zeek" with package named "foo" in alice/zkg.index, the following inputs may refer to the package: "foo", "alice/foo", or "zeek/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))

zeek_plugin_path()
Return the path where installed package plugins are located.
This path can be added to ZEEK_PLUGIN_PATH for interoperability with Zeek.

zeekpath()
Return the path where installed package scripts are located.
This path can be added to ZEEKPATH for interoperability with Zeek.

5.2 zeekpkg.package module

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

class zeekpkg.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

zeekpkg.package.METADATA_FILENAME = "zkg.meta"
The name of files used by packages to store their metadata.

class zeekpkg.package.Package (git_url, source="", directory="", metadata=None, name=None, canonical=False)
    Bases: object
    A Zeek 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 Zeek 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 zkg.index containing this package is located. E.g. if the package source has a package named "foo" declared in alice/zkg.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 zkg.index file.

Type str
metadata
the contents of the package’s zkg.meta or 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). The names 'zeek' or 'zkg' may also be keys that indicate a dependency on a particular Zeek or zkg 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 "zeek/alice/foo", the following inputs will match: "foo", "alice/foo", "zeek/alice/foo"
name_with_source_directory()
Return the package’s name within its package source.
E.g. for a package source with a package named "foo" in alice/zkg.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 zeekpkg.package.PackageInfo (package=None, status=None, metadata=None, versions=None, metadata_version="", invalid_reason="", version_type="", metadata_file=None, default_branch=None)

Bases: object

Contains information on an arbitrary package.

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

package
the relevant Zeek package
    Type Package

status
this attribute is set for installed packages
    Type PackageStatus

metadata
the contents of the package’s zkg.meta or bro-pkg.meta
    Type dict of str -> str

versions
a list of the package’s available 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

metadata_file
the absolute path to the zkg.meta or bro-pkg.meta for this package. Use this if you’d like to parse the metadata yourself. May not be defined, in which case the value is None.

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 default branch like 'main' or 'master'.

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

The keys indicate the name of a package (shorthand name or full git URL). The names 'zeek' or 'zkg' may also be keys that indicate a dependency on a particular Zeek or zkg 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 user variables parsed from metadata’s 'user_vars' field.

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

**class zeekpkg.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

zeekpkg.package.aliases(metadata_dict)
Return a list of package aliases found in metadata’s ‘aliases’ field.

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

zeekpkg.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). The names ‘zeek’ or ‘zkg’ may also be keys that indicate a dependency on a particular Zeek or zkg 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.

zeekpkg.package.is_valid_name(name)
Returns True if name is a valid package name, else False.

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

zeekpkg.package.short_description(metadata_dict)
Returns the first sentence of the metadata’s ‘desciption’ field.

zeekpkg.package.tags(metadata_dict)
Return a list of tag strings found in the metadata’s ‘tags’ field.

zeekpkg.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, its 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 zeekpkg.source module

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

zeekpkg.source.AGREGATE_DATA_FILE = 'aggregate.meta'
The name of the package source file where package metadata gets aggregated.

zeekpkg.source.INDEX_FILENAME = 'zkg.index'
The name of package index files.

class zeekpkg.source.Source(name, clone_path, git_url, version=None)
    Bases: object
    A Zeek 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 its [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.

### 5.4 zeekpkg.template module

A module for instantiating different types of Zeek packages.

**exception zeekpkg.template.Error**
*Bases: Exception*

Base class for any template-related errors.

**class zeekpkg.template.Feature**
*Bases: zeekpkg.template._Content*

Features overlay additional functionality onto a package.

This class fills in feature-specific functionality but it still abstract. At a minimum, your template’s Feature derivative needs to implement `contentdir()`.

**name()**
A name for this feature. Defaults to its content directory.

**exception zeekpkg.template.GitError**
*Bases: zeekpkg.template.LoadError*

There’s git trouble while producing template output.

**exception zeekpkg.template.InputError**
*Bases: zeekpkg.template.Error*

Something’s amiss in the input arguments for a package.

**exception zeekpkg.template.LoadError**
*Bases: zeekpkg.template.Error*

Something’s going wrong while retrieving a template.

**exception zeekpkg.template.OutputError**
*Bases: zeekpkg.template.Error*

Something’s going wrong while producing template output.
class zeepkg.template.Package
Bases: zeepkg.template._Content

Template content for a Zeek package.

This class fills in package-specific functionality but it still abstract. At a minimum, your template’s Package
derivative needs to implement contentdir().

do_instantiate(tmpl, packagedir, use_force=False)
Main driver for instantiating template content.

zkg calls this internally as part of template instantiation. You’ll likely want to focus on instantiate() for
customization.

Parameters

• tmpl (zeepkg.template.Template) – template context
• packagedir (str) – output folder for the instantiation
• use_force (bool) – whether to overwrite/recreate files as needed

class zeepkg.template.Template(templatedir, api_version, version=None, repo=None)
Bases: object

Base class for any template.

Templates need to derive from this class in their toplevel __init__.py. Instances of this class pull together the
components in a given template and capture their parameterization.

api_version()
The template API version string declared in this instance’s module.

apply_user_vars(user_vars)
Apply the user variables to this template.

Override this by invoking self.define_param() as needed to create template parameters based on the pro-
vided user vars. The relationship of user vars to template parameters is up to you. They can be a 1:1
mapping, you can derive additional parameters from a single user var (e.g. to accommodate string suf-
fixes), or you can use a combination of user vars to define a resulting parameter.

Parameters user_vars (list of zeepkg.uservar.UserVar) – input values for
the template.

define_param(name, val)
Defines a parameter of the given name and value.

define_user_vars() Defines the full set of user vars supported by this template.

This function defines the complete set of user vars supported by the template content. Instances of
zeepkg.template.Package and zeepkg.template.Feature declare which of these user vars they require
by implementing the needed_user_vars() method, returning the names of those variables.

The default implementation declares no user variables.

Returns list of zeepkg.uservar.Uservar instances

features() Provides any additional features templates supported.

If the template provides extra features, return each as an instance of zeepkg.template.Feature instance in
a list. By default, a template offers no features.

Returns list of zeepkg.template.Feature instances
info()  
Returns a dict capturing information about this template  
This is usable when rendered as JSON, and also serves as the input for our human-readable template information.

static is_api_compatible(tmpl_ver)  
Validate template API compatibility.  
Given a semver string describing the API version for which a template was written, verifies that we are compatible with it according to semantic versioning rules:  
MAJOR version changes when we make incompatible API changes MINOR version changes when you add backwards-compatible functionality PATCH version changes when you make backwards-compatible bug fixes.  
Returns bool indicating whether template is compatible.  
Raisess ValueError when given version isn't semver-parseable—

static load(config, template, version=None)  
Template loader.  
This function instantiates a zeekpkg.template.Template from a template source present either locally on disk or provided as a repo URL.  
It first uses the template’s __init__.py to bootstrap a module on the fly, then instantiates the zeekpkg.template.Template derivative that must be present in it.  
Parameters
• config(configparser.ConfigParser) – a zkg configuration  
• template(str) – template source repo, as directory or git URL  
• version(str) – if provided, a specific version tag to use. Ignored when "template" is a local directory. Otherwise, the same logic applies as with packages: the most recent version tag is picked, and if no version tags are available, the default branch.  
Returns zeekpkg.template.Template derivative  
Raises
• zeekpkg.template.GitError – git hit a problem during cloning/checkout  
• zeekpkg.template.LoadError – the template Python code does not load cleanly

lookup_param(name, default="")  
Looks up a parameter, falling back to the given default.

name()  
A name for this template, derived from the repo URL.

package()  
Provides a package template to instantiate.  
If the template provides a Zeek package, return a Package instance from this method.  
Returns zeekpkg.template.Package instance

params()  
Returns current str->str template parameter dict.

templatedir()  
Returns the path to the template’s source tree on disk.
version()
A version string for the template.

This can be a git tag, branch, commit hash, or None if we’re using the latest version on the default branch.

5.5 zeekpkg.uservar module

A module for zkg’s notion of "user variables": named values required by packages that the user can provide in a variety of ways, including responses to zkg’s input prompting.

class zeekpkg.uservar.UserVar
Bases: object

A class representing a single user variable.

User variables have a name and an optional description. They resolve to a value using a cascade of mechanisms, including command-line arguments (via –user-var), environment variables, cached previous values, and user input. They may come with a default value.

default()
desc()
name()

static parse_arg(arg)
Parser for NAME=VAL format string used in command-line args.

static parse_dict(metadata_dict)
Returns list of UserVars from the metadata’s ‘user_vars’ field.

Parameters
metadata_dict (dict of str->str) – the input metadata, e.g. from a configparser entry value.

Returns list of UserVar. If the ‘user_vars’ field is not present, an empty list is returned. If malformed, returns None.

resolve(name, config, user_var_args=None, force=False)
Populates user variables with updated values and returns them.

This function resolves the variable in the following order:
(1) Use any value provided on the command line via –user-var
(2) If force is not used, prompt the user for input
(3) use an environment variables of the same name,
(4) retrieve from the provided config parser’s "user_vars" section,
(5) use the default value of the user variable.

The resolved value is stored with the instance (to be retrieved via .val() in the future) and also returned.

Parameters
- name (str) – the requesting entity, e.g. a package name
- config (configparser.ConfigParser) – the zkg configuration
- user_var_args (list of UserVar) – user-var instances provided via command line
- force (bool) – whether to skip prompting for input
**Returns**  the resulting variable value

**Return type**  str

**Raises**  `ValueError` – when we couldn’t produce a value for the variable

```python
set(val)
```

```python
val (fallback=None)
```

`zeekpkg.uservar.slugify(string)`

Returns file-system-safe, lower-case version of the input string.

Any character sequence outside of `[a-zA-Z0-9_]` + gets replaced by a single underscore. If the variable has no value or the value is an empty string, returns the given default.
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 `zkg` 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 'zeek' 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 zkg usage
GitPython
semantic_version
btest
```

(continues on next page)
# Requirements for development (e.g. building docs)

Sphinx
sphinxcontrib-napoleon
sphinx_rtd_theme

They can be installed like:

```
pip3 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, or `make doc` to build them both. To view the generated HTML output, open `doc/_build/index.html`. The generated man page is located in `doc/man/zkg.1`.

If you have also installed `sphinx-autobuild` (e.g. via `pip3`), 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.

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

Python API docstrings roughly follow the Google Style Docstrings format.
Z
zeekpkg, 35
zeekpkg.manager, 35
zeekpkg.package, 45
zeekpkg.source, 49
zeekpkg.template, 50
zeekpkg.uservar, 53
Index

A
add_source() (zeekpkg.manager.Manager method), 37
AGGREGATE_DATA_FILE (in module zeekpkg.source), 49
aggregate_source() (zeekpkg.manager.Manager method), 37
aliases() (in module zeekpkg.package), 49
aliases() (zeekpkg.package.Package method), 46
aliases() (zeekpkg.package.PackageInfo method), 47
api_version() (zeekpkg.template.Template method), 51
apply_user_vars() (zeekpkg.template.Template method), 51
autoload_package (zeekpkg.manager.Manager attribute), 37
autoload_script (zeekpkg.manager.Manager attribute), 37
B
backup_dir (zeekpkg.manager.Manager attribute), 36
backup_modified_files() (zeekpkg.manager.Manager method), 38
best_version() (zeekpkg.package.PackageInfo method), 47
bin_dir (zeekpkg.manager.Manager attribute), 36
bro_plugin_path() (zeekpkg.manager.Manager method), 38
bropath() (zeekpkg.manager.Manager method), 38
bundle() (zeekpkg.manager.Manager method), 38
bundle_info() (zeekpkg.manager.Manager method), 38
C
canonical_url() (in module zeekpkg.package), 49
close (zeekpkg.source.Source attribute), 50
current_hash (zeekpkg.package.PackageStatus attribute), 49
current_version (zeekpkg.package.PackageStatus attribute), 48
D
default() (zeekpkg.uservar.UserVar method), 53
define_param() (zeekpkg.template.Template method), 51
define_user_vars() (zeekpkg.template.Template method), 51
dependencies() (in module zeekpkg.package), 49
dependencies() (zeekpkg.package.Package method), 46
dependencies() (zeekpkg.package.PackageInfo method), 48
desc() (zeekpkg.uservar.UserVar method), 53
directory (zeekpkg.package.Package attribute), 46
do_instantiate() (zeekpkg.template.Package method), 51
E
environment variable
VAR, 56
environment variable
PATH, 4, 5, 30
ZEEK_PLUGIN_PATH, 45
ZEEKPATH, 37, 45
Error, 50
F
Feature (class in zeekpkg.template), 50
features() (zeekpkg.template.Template method), 51
find_installed_package() (zeekpkg.manager.Manager method), 38
G
get_installed_package_dependencies() (zeekpkg.manager.Manager method), 39
git_url (zeekpkg.package.Package attribute), 45
git_url (zeekpkg.source.Source attribute), 50
GitError, 50
Zeek Package Manager Documentation, Release 2.11.0

H
has_plugin() (zeekpkg.manager.Manager method), 39
has_scripts() (zeekpkg.manager.Manager method), 39

I
INDEX_FILENAME (in module zeekpkg.source), 49
info() (zeekpkg.manager.Manager method), 39
info() (zeekpkg.template.Template method), 51
InputError, 50
install() (zeekpkg.manager.Manager method), 39
installed_package_dependencies() (zeekpkg.manager.Manager method), 40
installed_packages() (zeekpkg.manager.Manager method), 40
installed_pkgs (zeekpkg.manager.Manager method), 35
InstalledPackage (class in zeekpkg.package), 45
invalid_reason (zeekpkg.package.PackageInfo attribute), 47
is_api_compatible() (zeekpkg.template.Template static method), 52
is_loaded (zeekpkg.package.PackageStatus attribute), 48
is_outdated (zeekpkg.package.PackageStatus attribute), 48
is_pinned (zeekpkg.package.PackageStatus attribute), 48
is_valid_name() (in module zeekpkg.package), 49
list_depender_pkgs() (zeekpkg.manager.Manager method), 40
load() (zeekpkg.manager.Manager method), 40
load() (zeekpkg.template.Template static method), 52
load_with_dependencies() (zeekpkg.manager.Manager method), 40
loaded_package_states() (zeekpkg.manager.Manager method), 41
loaded_packages() (zeekpkg.manager.Manager method), 41
LoadError, 50
log_dir (zeekpkg.manager.Manager attribute), 36
lookup_param() (zeekpkg.template.Template method), 52

M
Manager (class in zeekpkg.manager), 35
Manager.SourceAggregationResults (class in zeekpkg.manager), 37
manifest (zeekpkg.manager.Manager attribute), 37
match_source_packages() (zeekpkg.manager.Manager method), 41
matches_path() (zeekpkg.package.Package method), 46
metadata (zeekpkg.package.Package attribute), 46
metadata (zeekpkg.package.PackageInfo attribute), 47
metadata_file (zeekpkg.package.PackageInfo attribute), 47
METADATA_FILENAME (in module zeekpkg.package), 45
metadata_version (zeekpkg.package.PackageInfo attribute), 47
modified_config_files() (zeekpkg.manager.Manager method), 41

N
name (zeekpkg.package.Package attribute), 45
name (zeekpkg.package.Source attribute), 50
name() (zeekpkg.template.Feature method), 50
name() (zeekpkg.template.Template method), 52
name() (zeekpkg.uservar.UserVar method), 53
name_from_path() (in module zeekpkg.package), 49
name_with_source_directory() (zeekpkg.package.Package method), 46

O
OutputError, 50

P
Package (class in zeekpkg.package), 45
Package (class in zeekpkg.template), 50
package (zeekpkg.package.InstalledPackage attribute), 45
package (zeekpkg.package.PackageInfo attribute), 47
package() (zeekpkg.template.Template method), 52
package_build_log() (zeekpkg.manager.Manager method), 41
package_clonedir (zeekpkg.manager.Manager attribute), 36
package_index_files() (zeekpkg.source.Source method), 36
package_issues (zeekpkg.manager.Manager.SourceAggregationResults attribute), 37
package_testdir (zeekpkg.manager.Manager attribute), 36
package_versions() (zeekpkg.manager.Manager method), 41
PackageInfo (class in zeekpkg.package), 47
packages() (zeekpkg.source.Source method), 50
PackageStatus (class in zeekpkg.package), 48
params() (zeekpkg.template.Template method), 52
parse_arg() (zeekpkg.uservar.UserVar static method), 53
parse_dict() (zeekpkg.uservar.UserVar static method), 53
PATH, 4, 5, 30
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin()</td>
<td>zeekpkg.manager.Manager method</td>
<td>41</td>
</tr>
<tr>
<td>plugin_dir</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>36</td>
</tr>
<tr>
<td>qualified_name()</td>
<td>(zeekpkg.package.Package method)</td>
<td>46</td>
</tr>
<tr>
<td>refresh_error</td>
<td>(zeekpkg.manager.Manager.SourceAggregationResults attribute)</td>
<td>37</td>
</tr>
<tr>
<td>refresh_installed_packages()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>42</td>
</tr>
<tr>
<td>refresh_source()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>42</td>
</tr>
<tr>
<td>remove()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>42</td>
</tr>
<tr>
<td>resolve()</td>
<td>(zeekpkg.uservar.UserVar method)</td>
<td>53</td>
</tr>
<tr>
<td>restore_loaded_package_states()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>42</td>
</tr>
<tr>
<td>save_temporary_config_files()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>43</td>
</tr>
<tr>
<td>scratch_dir</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>36</td>
</tr>
<tr>
<td>script_dir</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>36</td>
</tr>
<tr>
<td>set()</td>
<td>(zeekpkg.uservar.UserVar method)</td>
<td>54</td>
</tr>
<tr>
<td>short_description()</td>
<td>(in module zeekpkg.package)</td>
<td>49</td>
</tr>
<tr>
<td>short_description()</td>
<td>(zeekpkg.package.Package method)</td>
<td>46</td>
</tr>
<tr>
<td>short_description()</td>
<td>(zeekpkg.package.PackageInfo method)</td>
<td>48</td>
</tr>
<tr>
<td>slugify()</td>
<td>(in module zeekpkg.uservar)</td>
<td>54</td>
</tr>
<tr>
<td>Source</td>
<td>(class in zeekpkg.source)</td>
<td>49</td>
</tr>
<tr>
<td>source</td>
<td>(zeekpkg.package.Package attribute)</td>
<td>46</td>
</tr>
<tr>
<td>source_clonedir</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>36</td>
</tr>
<tr>
<td>source_packages()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>43</td>
</tr>
<tr>
<td>sources</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>35</td>
</tr>
<tr>
<td>state_dir</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>36</td>
</tr>
<tr>
<td>status</td>
<td>(zeekpkg.package.InstalledPackage attribute)</td>
<td>45</td>
</tr>
<tr>
<td>status</td>
<td>(zeekpkg.package.PackageInfo attribute)</td>
<td>47</td>
</tr>
<tr>
<td>tags()</td>
<td>(in module zeekpkg.package)</td>
<td>49</td>
</tr>
<tr>
<td>tags()</td>
<td>(zeekpkg.package.Package method)</td>
<td>46</td>
</tr>
<tr>
<td>tags()</td>
<td>(zeekpkg.package.PackageInfo method)</td>
<td>48</td>
</tr>
<tr>
<td>Template</td>
<td>(class in zeekpkg.template)</td>
<td>51</td>
</tr>
<tr>
<td>templatedir()</td>
<td>(zeekpkg.template.Template method)</td>
<td>52</td>
</tr>
<tr>
<td>test()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>43</td>
</tr>
<tr>
<td>tracking_method</td>
<td>(zeekpkg.package.PackageStatus attribute)</td>
<td>48</td>
</tr>
<tr>
<td>unbundle()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>43</td>
</tr>
<tr>
<td>unload()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>43</td>
</tr>
<tr>
<td>unload_with_unused_dependers()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>44</td>
</tr>
<tr>
<td>unpin()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>44</td>
</tr>
<tr>
<td>upgrade()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>44</td>
</tr>
<tr>
<td>user_vars</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>36</td>
</tr>
<tr>
<td>user_vars()</td>
<td>(in module zeekpkg.package)</td>
<td>49</td>
</tr>
<tr>
<td>user_vars()</td>
<td>(zeekpkg.package.Package method)</td>
<td>47</td>
</tr>
<tr>
<td>user_vars()</td>
<td>(zeekpkg.package.PackageInfo method)</td>
<td>48</td>
</tr>
<tr>
<td>UserVar</td>
<td>(class in zeekpkg.uservar)</td>
<td>53</td>
</tr>
<tr>
<td>val()</td>
<td>(zeekpkg.uservar.UserVar method)</td>
<td>54</td>
</tr>
<tr>
<td>validate_dependencies()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>44</td>
</tr>
<tr>
<td>VAR</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>version()</td>
<td>(zeekpkg.template.Template method)</td>
<td>52</td>
</tr>
<tr>
<td>version_type</td>
<td>(zeekpkg.package.PackageInfo attribute)</td>
<td>47</td>
</tr>
<tr>
<td>versions</td>
<td>(zeekpkg.package.PackageInfo attribute)</td>
<td>47</td>
</tr>
<tr>
<td>zeek_dist</td>
<td>(zeekpkg.manager.Manager attribute)</td>
<td>35</td>
</tr>
<tr>
<td>ZEEK_PLUGIN_PATH</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>zeek_plugin_path()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>45</td>
</tr>
<tr>
<td>ZEEKPATH</td>
<td>37, 45</td>
<td></td>
</tr>
<tr>
<td>zeekpath()</td>
<td>(zeekpkg.manager.Manager method)</td>
<td>45</td>
</tr>
<tr>
<td>zeekpkg</td>
<td>(module)</td>
<td>35</td>
</tr>
<tr>
<td>zeekpkg.manager</td>
<td>(module)</td>
<td>35</td>
</tr>
<tr>
<td>zeekpkg.package</td>
<td>(module)</td>
<td>45</td>
</tr>
<tr>
<td>zeekpkg.source</td>
<td>(module)</td>
<td>49</td>
</tr>
<tr>
<td>zeekpkg.template</td>
<td>(module)</td>
<td>50</td>
</tr>
<tr>
<td>zeekpkg.uservar</td>
<td>(module)</td>
<td>53</td>
</tr>
</tbody>
</table>