zope.security Documentation

Release 6.2.dev0

Zope Foundation Contributors <zope-dev@zope.org>

Jan 18, 2023

Contents

CHAPTER 1

Narrative Documentation

1.1 Overview and Introduction

The Security framework provides a generic mechanism to implement security policies on Python objects. This introduction provides a tutorial of the framework explaining concepts, design, and going through sample usage from the perspective of a Python programmer using the framework outside of Zope.

1.1.1 Definitions

Principal

A generalization of a concept of a `user`. Further specializations include `groups of users` and principals that `know what groups they belong to`. All of these principals may interact with the system.

Permission

A kind of access, i.e. permission to READ vs. permission to WRITE. Fundamentally the whole security framework is organized around checking permissions on objects. Permissions are represented (and checked) as strings, with the exception of `a constant` that has the special meaning of "public", i.e., no checking needs to be done.

There are `permission objects` that can be registered as zope.component utilities for validation, introspection, and producing `lists of available permissions` to help users assign them to objects.

1.1.2 Purpose

The security framework's primary purpose is to guard and check access to Python objects. It does this by providing mechanisms for explicit and implicit security checks on attribute access for objects. Attribute names are mapped onto permission names when checking access and the implementation of the security check is defined by the security policy, which receives the object, the permission name, and an interaction.

Interactions are objects that represent the use of the system by one or more principals. An interaction contains a list of participations, which represents the way a single principal participates in the interaction. An HTTP request is one example of a participation.

Its important to keep in mind that the policy provided is just a default, and it can be substituted with one which doesn't care about principals or interactions at all.

1.1.3 Framework Components

Low Level Components

These components provide the infrastructure for guarding attribute access and providing hooks into the higher level security framework.

Checkers

A `checker` is associated with an object kind, and provides the hooks that map attribute checks onto permissions deferring to the security manager (which in turn defers to the policy) to perform the check.

Additionally, checkers provide for creating proxies of objects associated with the checker.

There are several implementation variants of checkers, such as checkers that grant access based on attribute names.

Proxies

`Wrappers around Python objects` that implicitly guard access to their wrapped contents by delegating to their associated checker. Proxies are also viral in nature, in that values returned by proxies are also proxied.

1.1.4 High Level Components

Security Management

Provides accessors for `setting up interactions` and the `global security policy`.

`Interaction`

An `interaction` represents zero or more principals manipulating or viewing (interacting with) the system.

Interactions also provide `a single method` that accepts the object and the permission of the access being checked and is used to implement the application logic for the security framework.

Participation

Stores information about a single principal `participating` in the `interaction`.

Security Policy

A `security policy` is used to create the interaction that will ultimately be responsible for security checking.

1.2 Example

As an example we take a look at constructing a multi-agent distributed system, and then adding a security layer using the Zope security model onto it.

1.2.1 Scenario

Our agent simulation consists of autonomous agents that live in various agent homes/sandboxes and perform actions that access services available at their current home. Agents carry around authentication tokens which signify their level of access within any given home. Additionally agents attempt to migrate from home to home randomly.

The agent simulation was constructed separately from any security aspects. Now we want to define and integrate a security model into the simulation. The full code for the simulation and the security model is available separately; we present only relevant code snippets here for illustration as we go through the implementation process.

For the agent simulation we want to add a security model such that we group agents into two authentication groups, "norse legends", including the principals thor, odin, and loki, and "greek men", including prometheus, archimedes, and thucydides.

We associate permissions with access to services and homes. We differentiate the homes such that certain authentication groups only have access to services or the home itself based on the local settings of the home in which they reside.

We define the homes/sandboxes

- origin - all agents start here, and have access to all services here.
- valhalla - only agents in the authentication group 'norse legend' can reside here.
- jail - all agents can come here, but only 'norse legend's can leave or access services.

1.2.2 Process

Loosely we define a process for implementing this security model

- mapping permissions onto actions
- mapping authentication tokens onto permissions
- implementing checkers and security policies that use our authentication tokens and permissions.
- binding checkers to our simulation classes
- inserting the hooks into the original simulation code to add proxy wrappers to automatically check security.
- inserting hooks into the original simulation to register the agents as the active principal in an interaction.

1.2.3 Defining a Permission Model

We define the following permissions:

```
NotAllowed = 'Not Allowed'
Public = Checker.CheckerPublic
TransportAgent = 'Transport Agent'
AccessServices = 'Access Services'
AccessAgents = 'Access Agents'
AccessTimeService = 'Access Time Services'
```

(continues on next page)

(continued from previous page)

```
AccessAgentService = 'Access Agent Service'
AccessHomeService = 'Access Home Service'
```

and create a dictionary database mapping homes to authentication groups which are linked to associated permissions.

1.2.4 Defining and Binding Checkers

`Checkers` are the foundational unit for the security framework. They define what attributes can be accessed or set on a given instance. They can be used implicitly via Proxy objects, to guard all attribute access automatically or explicitly to check a given access for an operation.

Checker construction expects two functions or dictionaries, one is used to map attribute names to permissions for attribute access and another to do the same for setting attributes.

We use the following checker factory function:

```
def PermissionMapChecker(permissions_map={},
                         setattr_permission_func=NoSetAttr):
    res = {}
    for k,v in permissions_map.items():
        for iv in v:
            res[iv]=k
    return checker.Checker(res.get, setattr_permission_func)

time_service_checker = PermissionMapChecker(
                               # permission : [methods]
                               {'AccessTimeService':['getTime']}
                               )
```

with the NoSetAttr function defined as a lambda which always return the permission `NotAllowed`.

To bind the checkers to the simulation classes we `register` our checkers with the security model's global checker registry:

```
import sandbox_simulation
from zope.security.checker import defineChecker
defineChecker(sandbox_simulation.TimeService, time_service_checker)
```

1.2.5 Defining a Security Policy

We implement our security policy such that it checks the current agent's authentication token against the given permission in the home of the object being accessed. (We extend a simple policy provided by the framework that will track participations for us):

```
from zope.security.simplepolicies import ParanoidSecurityPolicy

@provider(ISecurityPolicy)
@implementer(IInteraction)
class SimulationSecurityPolicy(ParanoidSecurityPolicy):

    def checkPermission(self, permission, object):

        home = object.getHome()
        db = getattr(SimulationSecurityDatabase, home.getId(), None)
```

(continues on next page)

(continued from previous page)

```
        if db is None:
            return False

        allowed = db.get('any', ())
        if permission in allowed or ALL in allowed:
            return True

        if not self.participations:
            return False
        for participation in self.participations:
            token = participation.principal.getAuthenticationToken()
            allowed = db.get(token, ())
            if permission not in allowed:
                return False

        return True
```

Since an interaction can have more than one principal, we check that *all* of them are given the necessary permission. This is not really necessary since we only create interactions with a single active principal.

There is some additional code present to allow for shortcuts in defining the permission database when defining permissions for all auth groups and all permissions.

1.2.6 Integration

At this point we have implemented our security model, and we need to integrate it with our simulation model. We do so in three separate steps.

First we make it such that agents only access homes that are wrapped in a security proxy. By doing this all access to homes and services (proxies have proxied return values for their methods) is implicitly guarded by our security policy.

The second step is that we want to associate the active agent with the security context so the security policy will know which agent's authentication token to validate against.

The third step is to set our security policy as the default policy for the Zope security framework. It is possible to create custom security policies at a finer grained than global, but such is left as an exercise for the reader.

1.2.7 Interaction Access

The `*default* implementation` of the interaction management interfaces defines interactions on a per thread basis with a function for an accessor. This model is not appropriate for all systems, as it restricts one to a single active interaction per thread at any given moment. Reimplementing the interaction access methods though is easily doable and is noted here for completeness.

1.2.8 Perspectives

It's important to keep in mind that there is a lot more that is possible using the security framework than what's been presented here. All of the interactions are interface based, such that if you need to re-implement the semantics to suite your application a new implementation of the interface will be sufficient. Additional possibilities range from restricted interpreters and dynamic loading of untrusted code to non Zope web application security systems. Insert imagination here ;-).

1.2.9 Zope Perspective

A Zope3 programmer will never commonly need to interact with the low level security framework. Zope3 defines a second security package over top the low level framework and authentication sources and checkers are handled via zcml registration. Still those developing Zope3 will hopefully find this useful as an introduction into the underpinnings of the security framework.

1.2.10 Authors

- Kapil Thangavelu <hazmat at objectrealms.net>
- Guido Wesdorp <guido at infrae.com>
- Marius Gedminas <marius at pov.lt>

1.3 Untrusted Interpreters and Security Proxies

Untrusted programs are executed by untrusted interpreters. Untrusted interpreters make use of security proxies to prevent un-mediated access to assets. An untrusted interpreter defines an environment for running untrusted programs. All objects within the environment are either:

- "safe" objects created internally by the environment or created in the course of executing the untrusted program, or
- "basic" objects
- security-proxied non-basic objects

The environment includes proxied functions for accessing objects outside of the environment. These proxied functions provide the only way to access information outside the environment. Because these functions are proxied, as described below, any access to objects outside the environment is mediated by the target security functions.

Safe objects are objects whose operations, except for attribute retrieval, and methods access only information stored within the objects or passed as arguments. Safe objects contained within the interpreter environment can contain only information that is already in the environment or computed directly from information that is included in the environment. For this reason, safe objects created within the environment cannot be used to directly access information outside the environment.

Safe objects have some attributes that could (very) indirectly be used to access assets. For this reason, an untrusted interpreter always proxies the results of attribute accesses on a safe objects.

Basic objects are safe objects that are used to represent elemental data values such as strings and numbers. Basic objects require a lower level of protection than non-basic objects, as will be described detail in a later section.

Security proxies mediate all object operations. Any operation access is checked to see whether a subject is authorized to perform the operation. All operation results other than basic objects are, in turn, security proxied. Security proxies will be described in greater detail in a later section. Any operation on a security proxy that results in a non-basic object is also security proxied.

All external resources needed to perform an operation are security proxied.

Let's consider the trusted interpreter for evaluating URLs. In operation 1 of the example, the interpreter uses a proxied method for getting the system root object. Because the method is proxied, the result of calling the method and the operation is also proxied.

The interpreter has a function for traversing objects. This function is proxied. When traversing an object, the function is passed an object and a name. In operation 2, the function is passed the result of operation 1, which is the proxied root object and the name 'A'. We may traverse an object by invoking an operation on it. For example, we may use an

operation to get a sub-object. Because any operation on a proxied object returns a proxied object or a basic object, the result is either a proxied object or a basic object. Traversal may also look up a component. For example, in operation 1, we might look up a presentation component named "A" for the root object. In this case, the external object is not proxied, but, when it is returned from the traversal function, it is proxied (unless it is a a basic object) because the traversal function is proxied, and the result of calling a proxied function is proxied (unless the result is a basic object). Operation 3 proceeds in the same way.

When we get to operation 4, we use a function for computing the default presentation of the result of operation 3. As with traversal, the result of getting the default presentation is either a proxied object or a basic object because the function for getting the default presentation is proxied.

When we get to the last operation, we have either a proxied object or a basic object. If the result of operation 4 is a basic object, we simply convert it to a string and return it as the result page. If the result of operation 4 is a non-basic object, we invoke a render operation on it and return the result as a string.

Note that an untrusted interpreter may or may not provide protection against excessive resource usage. Different interpreters will provide different levels of service with respect to limitations on resource usage.

If an untrusted interpreter performs an attribute access, the trusted interpreter must proxy the result unless the result is a basic object.

In summary, an untrusted interpreter assures that any access to assets is mediated through security proxies by creating an environment to run untrusted code and making sure that:

- The only way to access anything from outside of the environment is to call functions that are proxied in the environment.
- Results of any attribute access in the environment are proxied unless the results are basic objects.

1.3.1 Security proxies

Security proxies are objects that wrap and mediate access to objects.

The Python programming language used by Zope defines a set of specific named low-level operations. In addition to operations, Python objects can have attributes, used to represent data and methods. Attributes are accessed using a dot notation. Applications can, and usually do, define methods to provide extended object behaviors. Methods are accessed as attributes through the low-level operation named "__getattribute__". The Python code:

```
a.b()
```

invokes 2 operations:

1. Use the low-level `__getattribute__` operation with the name "b".
2. Use the low-level `__call__` operation on the result of the first operation.

For all operations except the `__getattribute__` and `__setattribute__` operations, security proxies have a permission value defined by the permission-declaration subsystem. Two special permission values indicate that access is either forbidden (never allowed) or public (always allowed). For all other permission values, the authorization subsystem is used to decide whether the subject has the permission for the proxied object. If the subject has the permission, then access to the operation is allowed. Otherwise, access is denied.

For getting or setting attributes, a proxy has permissions for getting and a permission for setting attribute values for a given attribute name. As described above, these permissions may be one of the two special permission values indicating forbidden or public access, or another permission value that must be checked with the authorization system.

For all objects, Zope defines the following operations to be always public:

comparison "__lt__", "__le__", "__eq__", "__gt__", "__ge__", "__ne__"

hash "__hash__"

boolean value "__nonzero__"

class introspection "__class__"

interface introspection "__providedBy__", "__implements__"

adaptation "__conform__"

low-level string representation "__repr__"

The result of an operation on a proxied object is a security proxy unless the result is a basic value.

1.3.2 Basic objects

Basic objects are safe immutable objects that contain only immutable subobjects. Examples of basic objects include:

- Strings,
- Integers (long and normal),
- Floating-point objects,
- Date-time objects,
- Boolean objects (True and False), and
- The special (nil) object, None.

Basic objects are safe, so, as described earlier, operations on basic objects, other than attribute access, use only information contained within the objects or information passed to them. For this reason, basic objects cannot be used to access information outside of the untrusted interpreter environment.

The decision not to proxy basic objects is largely an optimization. It allows low-level safe computation to be performed without unnecessary overhead,

Note that a basic object could contain sensitive information, but such a basic object would need to be obtained by making a call on a proxied object. Therefore, the access to the basic object in the first place is mediated by the security functions.

1.3.3 Rationale for mutable safe objects

Some safe objects are not basic. For these objects, we proxy the objects if they originate from outside of the environment. We do this for two reasons:

1. Non-basic objects from outside the environment need to be proxied to prevent unauthorized access to information.
2. We need to prevent un-mediated change of information from outside of the environment.

We don't proxy safe objects created within the environment. This is safe to do because such safe objects can contain and provide access to information already in the environment. Sometimes the interpreter or the interpreted program needs to be able to create simple data containers to hold information computed in the course of the program execution. Several safe container types are provided for this purpose.

1.3.4 Known Issues With Proxies

Security proxies (proxies in general) are not perfect in Python. There are some things that they cannot transparently proxy.

isinstance and proxies

A proxied object cannot proxy its type (although it does proxy its `__class__`):

```
>>> from zope.security.proxy import ProxyFactory
>>> class Object(object):
...     pass
>>> target = Object()
>>> target.__class__
<class 'Object'>
>>> type(target)
<class 'Object'>
>>> proxy = ProxyFactory(target, None)
>>> proxy.__class__
<class 'Object'>
>>> type(proxy)
<... 'zope.security...Proxy...'>
```

This means that the builtin `isinstance()` may return unexpected results:

```
>>> isinstance(target, Object)
True
>>> isinstance(proxy, Object)
False
```

There are two workarounds. The safest is to use `zope.security.proxy.isinstance()`, which takes specifically this into account (in modules that will be dealing with a number of proxies, it is common to simply place `from zope.security.proxy import isinstance` at the top of the file to override the builtin `isinstance()`; we won't show that here for clarity):

```
>>> import zope.security.proxy
>>> zope.security.proxy.isinstance(target, Object)
True
>>> zope.security.proxy.isinstance(proxy, Object)
True
```

Alternatively, you can manually remove the security proxy (or indeed, all proxies) with `zope.security.proxy.removeSecurityProxy()` or `zope.proxy.removeAllProxies()`, respectively, before calling `isinstance()`:

```
>>> from zope.security.proxy import removeSecurityProxy
>>> isinstance(removeSecurityProxy(target), Object)
True
>>> isinstance(removeSecurityProxy(proxy), Object)
True
```

issubclass and proxies

Security proxies will proxy the return value of `__class__`: it will be a proxy around the real class of the proxied value. This causes failures with `issubclass`:

```
>>> from zope.security.proxy import ProxyFactory
>>> class Object(object):
...     pass
>>> target = Object()
>>> target.__class__ is Object
```

(continues on next page)

(continued from previous page)

```
True
>>> proxy = ProxyFactory(target, None)
>>> proxy.__class__
<class 'Object'>
>>> proxy.__class__ is Object
False
>>> issubclass(proxy.__class__, Object)
Traceback (most recent call last):
...
TypeError: issubclass() arg 1 must be a class
```

Although the above is a contrived example, using `abstract base classes` can cause it to arise quite unexpectedly:

```
>>> try:
...     from collections.abc import Mapping
... except ImportError:  # PY2
...     from collections import Mapping
>>> from abc import ABCMeta
>>> isinstance(Mapping, ABCMeta)
True
>>> isinstance(proxy, Mapping)
Traceback (most recent call last):
...
TypeError: issubclass() arg 1 must be a class
```

In this case, the workarounds described *above* also work:

```
>>> zope.security.proxy.isinstance(proxy, Mapping)
False
>>> isinstance(removeSecurityProxy(proxy), Mapping)
False
```

logging

Starting with Python 2.7.7, the `logging.LogRecord` makes exactly the above `isinstance` call:

```
>>> from logging import LogRecord
>>> LogRecord("name", 1, "/path/to/file", 1,
...     "The message %s", (proxy,), None)
Traceback (most recent call last):
...
TypeError: issubclass() arg 1 must be a class
```

Possible workarounds include:

- Carefully removing security proxies of objects before passing them to the logging system.
- Monkey-patching the logging system to use `zope.security.proxy.isinstance()` which does this automatically:

  ```
  import zope.security.proxy
  import logging
  logging.isinstance = zope.security.proxy.isinstance
  ```

- Using `logging.setLogRecordfactory()` to set a custom `LogRecord` subclass that unwraps any security proxies before they are given to the super class. Note that this is only available on Python 3. On Python 2, it might be possible to achieve a similar result with a custom `logger class`:

```
>>> from zope.security.proxy import removeSecurityProxy
>>> class UnwrappingLogRecord(LogRecord):
...     def __init__(self, name, level, pathname, lineno,
...                  msg, args, exc_info, *largs, **kwargs):
...         args = [removeSecurityProxy(x) for x in args]
...         LogRecord.__init__(self, name, level, pathname,
...                            lineno, msg, args, exc_info, *largs, **kwargs)
...     def __repr__(self):
...         return '<UnwrappingLogRecord>'
>>> UnwrappingLogRecord("name", 1, "/path/to/file", 1,
...                     "The message %s", (proxy,), None)
<UnwrappingLogRecord>
```

Each specific application will have to determine what solution is correct for its security model.

1.4 Hacking on `zope.security`

1.4.1 Getting the Code

The main repository for `zope.security` is in the Zope Foundation Github repository:

https://github.com/zopefoundation/zope.security

You can get a read-only checkout from there:

```
$ git clone https://github.com/zopefoundation/zope.security.git
```

or fork it and get a writeable checkout of your fork:

```
$ git clone git@github.com/jrandom/zope.security.git
```

The project also mirrors the trunk from the Github repository as a Bazaar branch on Launchpad:

https://code.launchpad.net/zope.security

You can branch the trunk from there using Bazaar:

```
$ bzr branch lp:zope.security
```

1.4.2 Working in a `virtualenv`

Installing

If you use the `virtualenv` package to create lightweight Python development environments, you can run the tests using nothing more than the `python` binary in a virtualenv. First, create a scratch environment:

```
$ /path/to/virtualenv --no-site-packages /tmp/hack-zope.security
```

Next, get this package registered as a "development egg" in the environment:

```
$ /tmp/hack-zope.security/bin/pip install -e .[test]
```

Running the tests

Then, you can run the tests using the zope.testrunner (or a test runner of your choice):

```
$ /tmp/hack-zope.security/bin/zope-testrunner --test-path=src
Running zope.testrunner.layer.UnitTests tests:
  Set up zope.testrunner.layer.UnitTests in 0.000 seconds.
  Running:

  Ran 742 tests with 0 failures, 0 errors, 36 skipped in 0.253 seconds.
Tearing down left over layers:
  Tear down zope.testrunner.layer.UnitTests in 0.000 seconds.
```

If you have the `coverage` package installed in the virtualenv, you can see how well the tests cover the code:

```
$ /tmp/hack-zope.security/bin/pip install coverage
...
$ coverage run -m zope.testrunner --test-path=src
...
$ coverage report
Name                                Stmts   Miss  Cover   Missing
-----------------------------------------------------------------
zope/security.py                        4      0   100%
zope/security/_compat.py                9      0   100%
zope/security/_definitions.py          11      0   100%
zope/security/adapter.py               45      0   100%
zope/security/checker.py              333      0   100%
zope/security/decorator.py             33      0   100%
zope/security/i18n.py                   4      0   100%
zope/security/interfaces.py            65      0   100%
zope/security/management.py            62      0   100%
zope/security/metaconfigure.py        108      0   100%
zope/security/metadirectives.py        38      0   100%
zope/security/permission.py            46      0   100%
zope/security/protectclass.py          39      0   100%
zope/security/proxy.py                164     19    88%   55, 86, 97, 119-121, 127-129,␣
↪143-144, 153-154, 158-159, 163-164, 298, 330
zope/security/simplepolicies.py        32      0   100%
zope/security/zcml.py                  43      0   100%
-----------------------------------------------------------------
TOTAL                                1036     19    98%
-----------------------------------------------------------------------
Ran 655 tests in 0.000s

OK
```

Building the documentation

`zope.security` uses the nifty `Sphinx` documentation system for building its docs. Using the same virtualenv you set up to run the tests, you can build the docs:

```
$ /tmp/hack-zope.security/bin/pip install -e .[docs]
...
$ cd docs
$ PATH=/tmp/hack-zope.security/bin:$PATH make html
sphinx-build -b html -d _build/doctrees   . _build/html
...
build succeeded.

Build finished. The HTML pages are in _build/html.
```

You can also test the code snippets in the documentation:

```
$ PATH=/tmp/hack-zope.security/bin:$PATH make doctest
sphinx-build -b doctest -d _build/doctrees   . _build/doctest
...
running tests...

Document: api/zcml
------------------
1 items passed all tests:
  23 tests in default
23 tests in 1 items.
23 passed and 0 failed.
Test passed.

Document: api/proxy
-------------------
1 items passed all tests:
  11 tests in default
11 tests in 1 items.
11 passed and 0 failed.
Test passed.
1 items passed all tests:
   1 tests in default (cleanup code)
1 tests in 1 items.
1 passed and 0 failed.
Test passed.

Document: api/permission
------------------------
1 items passed all tests:
  35 tests in default
35 tests in 1 items.
35 passed and 0 failed.
Test passed.
1 items passed all tests:
   1 tests in default (cleanup code)
1 tests in 1 items.
1 passed and 0 failed.
Test passed.

Document: api/checker
---------------------
1 items passed all tests:
 356 tests in default
356 tests in 1 items.
356 passed and 0 failed.
```

(continues on next page)

(continued from previous page)

```
Test passed.

Document: api/decorator
-----------------------
1 items passed all tests:
  53 tests in default
53 tests in 1 items.
53 passed and 0 failed.
Test passed.
1 items passed all tests:
   1 tests in default (cleanup code)
1 tests in 1 items.
1 passed and 0 failed.
Test passed.

Doctest summary
===============
  478 tests
    0 failures in tests
    0 failures in setup code
    0 failures in cleanup code
```

1.4.3 Using `zc.buildout`

Setting up the buildout

`zope.security` ships with its own `buildout.cfg` file and `bootstrap.py` for setting up a development buildout:

```
$ /path/to/python2.6 bootstrap.py
...
Generated script '.../bin/buildout'
$ bin/buildout
Develop: '/home/jrandom/projects/Zope/BTK/security/.'
...
```

Running the tests

You can now run the tests:

```
$ bin/test --all
Running zope.testing.testrunner.layer.UnitTests tests:
  Set up zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
  Ran 643 tests with 0 failures and 0 errors in 0.000 seconds.
Tearing down left over layers:
  Tear down zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
```

1.4.4 Using `tox`

Running Tests on Multiple Python Versions

tox is a Python-based test automation tool designed to run tests against multiple Python versions. It creates a

`virtualenv` for each configured version, installs the current package and configured dependencies into each `virtualenv`, and then runs the configured commands.

`zope.security` configures the following `tox` environments via its `tox.ini` file:

- The `py27`, `py34`, `py35`, `pypy`, etc, environments builds a `virtualenv` with the appropriate interpreter, installs `zope.security` and dependencies, and runs the tests.
- The `py27-pure` and `py33-pure` environments build a `virtualenv` with the appropriate interpreter, installs `zope.security` and dependencies **without compiling C extensions**, and runs the tests via `python setup.py test -q`.
- The `coverage` environment builds a `virtualenv`, runs all the tests under `coverage`, and prints a report to stdout.
- The `docs` environment builds a virtualenv and then builds the docs and exercises the doctest snippets.

This example requires that you have a working `python2.7` on your path, as well as installing `tox`:

```
$ tox -e py27
GLOB sdist-make: .../zope.security/setup.py
py27 sdist-reinst: .../zope.security/.tox/dist/zope.security-4.0.2dev.zip
py27 runtests: commands[0]
.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.....................................................
----------------------------------------------------------------------
Ran 643 tests in 0.000s

OK
_____________________________________ summary ______________________________________
py26: commands succeeded
congratulations :)
```

Running `tox` with no arguments runs all the configured environments, including building the docs and testing their snippets:

```
$ tox
GLOB sdist-make: .../zope.security/setup.py
py26 sdist-reinst: .../zope.security/.tox/dist/zope.security-4.0.2dev.zip
py26 runtests: commands[0]
...
Doctest summary
===============
 478 tests
   0 failures in tests
   0 failures in setup code
   0 failures in cleanup code
build succeeded.
_____________________________________ summary ______________________________________
py26: commands succeeded
py27: commands succeeded
py27-pure: commands succeeded
pypy: commands succeeded
py32: commands succeeded
```

(continues on next page)

(continued from previous page)

```
py33: commands succeeded
py33-pure: commands succeeded
py34: commands succeeded
coverage: commands succeeded
docs: commands succeeded
congratulations :)
```

1.4.5 Contributing to `zope.security`

Submitting a Bug Report

`zope.security` tracks its bugs on Github:

https://github.com/zopefoundation/zope.security/issues

Please submit bug reports and feature requests there.

Sharing Your Changes

Note: Please ensure that all tests are passing before you submit your code. If possible, your submission should include new tests for new features or bug fixes, although it is possible that you may have tested your new code by updating existing tests.

If have made a change you would like to share, the best route is to fork the Githb repository, check out your fork, make your changes on a branch in your fork, and push it. You can then submit a pull request from your branch:

https://github.com/zopefoundation/zope.security/pulls

1.5 Changes

1.5.1 6.2 (unreleased)

6.1 (2023-01-18)

- Remove more proxying code for names that no longer exist in Python 3. (#92)

6.0 (2023-01-16)

- Remove proxying code for names that no longer exist in Python 3. (#92)
- Drop support for Python 2.7, 3.5, 3.6.

5.8 (2022-11-30)

- The extra `untrustedpython` now for Python 3, too, installs `zope.untrustedpython`.

5.7 (2022-11-17)

- Release to rebuild full set of binary wheels.

5.6 (2022-11-16)

- Add support for building arm64 wheels on macOS.

5.5 (2022-11-06)

- Add support for final release of Python 3.11.

5.4 (2022-09-15)

- Disable unsafe math optimizations in C code. See pull request 89.

5.3 (2022-04-27)

- Allow calling bound methods of some built-in objects such as `().__repr__` and `{}.__repr__` by default. This worked on Python 2, but raised `ForbiddenAttribute` on Python 3. See issue 75.
- Remove usage of `unittest.makeSuite` as it is deprecated in Python 3.11+. See issue 83.
- Add support for Python 3.11 (as of 3.11.0a7).

5.2 (2022-03-10)

- Add support for Python 3.9 and 3.10.

5.1.1 (2020-03-23)

- Ensure all objects have consistent interface resolution orders (if all dependencies are up-to-date). See issue 71.

5.1.0 (2020-02-14)

- Let proxied interfaces be iterated on Python 3. This worked on Python 2, but raised `ForbiddenAttribute` an Python 3. See zope.interface issue 141.
- Allow to use a common Sphinx version for Python 2 and 3.

5.0.0 (2019-11-11)

- Drop support for Python 3.4.
- Add support for Python 3.8.
- Properly declare dependency on zope.schema >= 4.2.0, introduced in zope.security 4.2.1.
- Fix dict item view iteration on PyPy3 7.x.

4.3.1 (2019-01-03)

- Fix the decimal.Decimal checker, `__truediv__` was missing causing `ForbiddenAttribute` on a `ProxyFactory(Decimal('1')) / 1` operation

4.3.0 (2018-08-24)

- Add the interface `ISystemPrincipal` and make `zope.security.management.system_user` a regular object that implements this interface. This facilitates providing adapter registrations specifically for the `system_user`.

4.2.3 (2018-08-09)

- Add support for Python 3.7.

4.2.2 (2018-01-11)

- Make the pure-Python proxy on Python 2 *not* check permissions for `__unicode__` just like the C implementation. Note that `__str__` is checked for both implementations on both Python 2 and 3, but if there is no `__unicode__` method defined, Python 2's automatic fallback to `__str__` is **not** checked when `unicode` is called. See issue 10.

4.2.1 (2017-11-30)

- Fix the default values for `Permission` fields `title` and `description` under Python 2. See issue 48.
- Change the `IPermission.id` from `Text` (unicode) to a `NativeStringLine`. This matches what ZCML creates and what is usually written in source code.

4.2.0 (2017-09-20)

- Fix the extremely rare potential for a crash when the C extensions are in use. See issue 35.
- Fix issue 7: The pure-Python proxy didn't propagate `TypeError` from `__repr__` and `__str__` like the C implementation did.
- Fix issue 27: iteration of `zope.interface.providedBy()` is now allowed by default on all versions of Python. Previously it only worked on Python 2. Note that `providedBy` returns unproxied objects for backwards compatibility.
- Fix `__length_hint__` of proxied iterator objects. Previously it was ignored.
- Drop support for Python 3.3.
- Enable coveralls.io for coverage measurement and run doctests on all supported Python versions.
- Fix issue 9: iteration of `itertools.groupby` objects is now allowed by default. In addition, iteration of all the custom iterator types defined in itertools are also allowed by default.
- Simplify the internal `_compat.py` module now that we only run on newer Python versions. See PR 32.
- Respect `PURE_PYTHON` at runtime. At build time, always try to build the C extensions on supported platforms, ignoring `PURE_PYTHON`. See issue 33.
- Fix watching checkers (`ZOPE_WATCH_CHECKERS=1`) in pure-Python mode. See issue 8.

- Remove unused internal files from `tests/`.
- Remove `zope.security.setup`. It was unused and did not work anyway.
- Fix the pure-Python proxy on Python 2 letting `__getslice__` and `__setslice__` fall through to `__getitem__` or `__setitem__`, respectively, if it raised an error.
- Fix the pure-Python proxy calling a wrapped `__getattr__` or `__getattribute__` more than once in situations where the C implementation only called it one time (when it raised an AttributeError).
- Reach 100% test coverage and maintain it via automated checks.

4.1.1 (2017-05-17)

- Fix issue 23: iteration of `collections.OrderedDict` and its various views is now allowed by default on all versions of Python.
- As a further fix for issue 20, iteration of `BTree` itself is now allowed by default.

4.1.0 (2017-04-24)

- When testing `PURE_PYTHON` environments under `tox`, avoid poisoning the user's global wheel cache.
- Drop support for Python 2.6 and 3.2.
- Add support for Python 3.5 and 3.6.
- Fix issue 20: iteration of pure-Python `BTrees.items()`, and also creating a list from `BTrees.items()` on Python 3. The same applies for `keys()` and `values()`.

4.0.3 (2015-06-02)

- Fix iteration over security proxies in Python 3 using the pure-Python implementation.

4.0.2 (2015-06-02)

- Fix compatibility with `zope.proxy` 4.1.5 under PyPy.
- Fix the very first call to `removeSecurityProxy` returning incorrect results if given a proxy under PyPy.

4.0.1 (2014-03-19)

- Add support for Python 3.4.

4.0.0 (2013-07-09)

- Update `boostrap.py` to version 2.2.
- Bugfix: ZOPE_WATCH_CHECKERS=2 used to incorrectly suppress unauthorized/forbidden warnings.
- Bugfix: ZOPE_WATCH_CHECKERS=1 used to miss most of the checks.

4.0.0b1 (2013-03-11)

- Add support for PyPy.
- Fix extension compilation on windows python 3.x

4.0.0a5 (2013-02-28)

- Undo changes from 4.0.0a4. Instead, `zope.untrustedpython` is only included during Python 2 installs.

4.0.0a4 (2013-02-28)

- Remove `untrustedpython` extra again, since we do not want to support `zope.untrustedpython` in ZTK 2.0. If BBB is really needed, we will create a 3.10.0 release.

4.0.0a3 (2013-02-15)

- Fix test breakage in 4.0.0a2 due to deprecation strategy.

4.0.0a2 (2013-02-15)

- Add back the `untrustedpython` extra: now pulls in `zope.untrustedpython`. Restored deprecated backward-compatible imports for `zope.security.untrustedpython.{builtins, interpreter,rcompile}` (the extra and the imports are to be removed in version 4.1).

4.0.0a1 (2013-02-14)

- Add support for Python 3.2 and 3.3.
- Bring unit test coverage to 100%.
- `zope.security.untrustedpython` moved to separate project: `zope.untrustedpython`
- Convert use of `assert` in non-test code to apprpriate error types:
  - Non-dict's passed to `Checker.__init__`.
- Remove dprecattion of `zope.security.adapter.TrustedAdapterFactory`. Although it has been marked as deprectaed since before Zope3 3.2, current versions of `zope.compoent` still rely on it.
- Convert doctests to Sphinx documentation in 'docs'.
- Add `setup.py docs` alias (installs `Sphinx` and dependencies).
- Add `setup.py dev` alias (runs `setup.py develop` plus installs `nose` and `coverage`).
- Make non-doctest tests fully independent of `zope.testing`.

  Two modules, `zope.security.checker` and `zope.security.management`, register cleanups with `zope.testing` IFF it is importable, but the tests no longer rely on it.
- Enable building extensions without the `svn:external` of the `zope.proxy` headers into our `include` dir.
- Bump `zope.proxy` dependency to ">= 4.1.0" to enable compilation on Py3k.
- Replace deprecated `zope.component.adapts` usage with equivalent `zope.component.adapter` decorator.

- Replace deprecated `zope.interface.classProvides` usage with equivalent `zope.interface.provider` decorator.
- Replace deprecated `zope.interface.implements` usage with equivalent `zope.interface.implementer` decorator.
- Drop support for Python 2.4 and 2.5.
- Add test convenience helper `create_interaction` and `with interaction()`.

3.9.0 (2012-12-21)

- Pin `zope.proxy >= 4.1.0`
- Ship with an included `proxy.h` header which is compatible with the 4.1.x version ov `zope.proxy`.

3.8.5 (2012-12-21)

- Ship with an included `proxy.h` header which is compatible with the supported versions of `zope.proxy`.

3.8.4 (2012-12-20)

- Pin `zope.proxy >= 3.4.2, <4.1dev`

3.8.3 (2011-09-24)

- Fix a regression introduced in 3.8.1: `zope.location`'s LocationProxy did not get a security checker if `zope.security.decorator` was not imported manually. Now `zope.security.decorator` is imported in `zope.security.proxy` without re-introducing the circular import fixed in 3.8.1.

3.8.2 (2011-05-24)

- Fix a test that failed on Python 2.7.

3.8.1 (2011-05-03)

- Fix circular import beween `zope.security.decorator` and `zope.security.proxy` which led to an `ImportError` when only importing `zope.security.decorator`.

3.8.0 (2010-12-14)

- Add tests for our own `configure.zcml`.
- Add `zcml` extra dependencies; run related tests only if `zope.configuration` is available.
- Run tests related to the `untrustedpython` functionality only if `RestrictedPython` is available.

3.7.3 (2010-04-30)

- Prefer the standard library's `doctest` module to the one from `zope.testing`.
- Ensure `PermissionIdsVocabulary` directly provides `IVocabularyFactory`, even though it might be unnecessary because `IVocabularyFactory` is provided in ZCML.
- Remove the dependency on the zope.exceptions package: zope.security.checker now imports `DuplicationError` from zope.exceptions if available, otherwise it defines a package-specific `DuplicationError` class which inherits from Exception.

3.7.2 (2009-11-10)

- Add compatibility with Python 2.6 abstract base classes.

3.7.1 (2009-08-13)

- Fix for LP bug 181833 (from Gustavo Niemeyer). Before "visiting" a sub-object, a check should be made to ensure the object is still valid. Because garbage collection may involve loops, if you garbage collect an object, it is possible that the actions done on this object may modify the state of other objects. This may cause another round of garbage collection, eventually generating a segfault (see LP bug). The Py_VISIT macro does the necessary checks, so it is used instead of the previous code.

3.7.0 (2009-05-13)

- Make `pytz` a soft dependency: the checker for `pytz.UTC` is created / tested only if the package is already present. Run `bin/test_pytz` to run the tests with `pytz` on the path.

3.6.3 (2009-03-23)

- Ensure that simple zope.schema's `VocabularyRegistry` is used for `PermissionVocabulary` tests, because it's replaced implicitly in environments with `zope.app.schema` installed that makes that tests fail.
- Fix a bug in `DecoratedSecurityCheckerDescriptor` which made security-wrapping location proxied exception instances throw exceptions on Python 2.5. See https://bugs.launchpad.net/zope3/+bug/251848

3.6.2 (2009-03-14)

- Add `zope.i18nmessageid.Message` to non-proxied basic types. It's okay, because messages are immutable. Done previously by `zope.app.security`.
- Add `__name__` and `__parent__` attributes to list of available by default. Done previously by `zope.app.security`.
- Move `PermissionsVocabulary` and `PermissionIdsVocabulary` vocabularies to the `zope.security.permission` module from the `zope.app.security` package.
- Add zcml permission definitions for most common and useful permissions, like `zope.View` and `zope.ManageContent`, as well as for the special `zope.Public` permission. They are placed in a separate `permissions.zcml` file, so it can be easily excluded/redefined. They are selected part of permissions moved from `zope.app.security` and used by many `zope.*` packages.
- Add `addCheckerPublic` helper function in `zope.security.testing` module that registers the "zope.Public" permission as an IPermission utility.

- Add security declarations for the `zope.security.permisson.Permission` class.
- Improve test coverage.

3.6.1 (2009-03-10)

- Use `from` imports instead of `zope.deferred` to avoid circular import problems, thus drop dependency on `zope.deferredimport`.
- Raise `NoInteraction` when `zope.security.checkPermission` is called without interaction being active (LP #301565).
- Don't define security checkers for deprecated set types from the "sets" module on Python 2.6. It's discouraged to use them and `set` and `frozenset` built-in types should be used instead.
- Change package's mailng list address to zope-dev at zope.org as zope3-dev at zope.org is now retired.
- Remove old zpkg-related files.

3.6.0 (2009-01-31)

- Install decorated security checker support on `LocationProxy` from the outside.
- Add support to bootstrap on Jython.
- Move the `protectclass` module from `zope.app.security` to this package to reduce the number of dependencies on `zope.app.security`.
- Move the `<module>` directive implementation from `zope.app.security` to this package.
- Move the `<class>` directive implementation from `zope.app.component` to this package.

3.5.2 (2008-07-27)

- Make C code compatible with Python 2.5 on 64bit architectures.

3.5.1 (2008-06-04)

- Add `frozenset`, `set`, `reversed`, and `sorted` to the list of safe builtins.

3.5.0 (2008-03-05)

- Changed title for `zope.security.management.system_user` to be more presentable.

3.4.3 - (2009/11/26)

- Backport a fix made by Gary Poster to the 3.4 branch: Fix for LP bug 181833 (from Gustavo Niemeyer). Before "visiting" a sub-object, a check should be made to ensure the object is still valid. Because garbage collection may involve loops, if you garbage collect an object, it is possible that the actions done on this object may modify the state of other objects. This may cause another round of garbage collection, eventually generating a segfault (see LP bug). The `Py_VISIT` macro does the necessary checks, so it is used instead of the previous code.

3.4.2 - (2009/03/23)

- Add dependency on `zope.thread` to setup.py; without it, the tests were failing.
- Backport a fix made by Albertas Agejevas to the 3.4 branch. He fixed a bug in DecoratedSecurityCheckerDescriptor which made security-wrapping location proxied exception instances throw exceptions on Python 2.5. See https://bugs.launchpad.net/zope3/+bug/251848

3.4.1 - 2008/07/27

- Make C code compatible with Python 2.5 on 64bit architectures.

3.4.0 (2007-10-02)

- Update meta-data.

3.4.0b5 (2007-08-15)

- Fix a circular import in the C implementation.

3.4.0b4 (2007-08-14)

- Improve ugly/brittle ID of `zope.security.management.system_user`.

3.4.0b3 (2007-08-14)

- Add support for Python 2.5.
- Bug: `zope.security.management.system_user` wasn't a valid principal (didn't provide IPrincipal).
- Bug: Fix inclusion of doctest to use the doctest module from `zope.testing`. Now tests can be run multiple times without breaking. (#98250)

3.4.0b2 (2007-06-15)

- Bug: Remove stack extraction in `newInteraction`. When using eggs this is an extremly expensive function. The publisher is now more than 10 times faster when using eggs and about twice as fast with a zope trunk checkout.

3.4.0b1

- Temporarily fixed the hidden (and accidental) dependency on zope.testing to become optional.

Note: The releases between 3.2.0 and 3.4.0b1 where not tracked as an individual package and have been documented in the Zope 3 changelog.

3.2.0 (2006-01-05)

- Corresponds to the verison of the `zope.security` package shipped as part of the Zope 3.2.0 release.
- Remove deprecated helper functions, `proxy.trustedRemoveSecurityProxy` and `proxy.getProxiedObject`.
- Make handling of `management.{end,restore}Interaction` more careful w.r.t. edge cases.
- Make behavior of `canWrite` consistent with `canAccess`: if `canAccess` does not raise `ForbiddenAttribute`, then neither will `canWrite`. See: http://www.zope.org/Collectors/Zope3-dev/506
- Code style / documentation / test fixes.

3.1.0 (2005-10-03)

- Add support for use of the new Python 2.4 datatypes, `set` and `frozenset`, within checked code.
- Make the C security proxy depend on the `proxy.h` header from the `zope.proxy` package.
- XXX: the spelling of the `#include` is bizarre! It seems to be related to `zpkg`-based builds, and should likely be revisited. For the moment, I have linked in the `zope.proxy` package into our own `include` directory. See the subversion checkin: http://svn.zope.org/Zope3/?rev=37882&view=rev
- Update checker to avoid re-proxying objects which have and explicit `__Security_checker__` assigned.
- Corresponds to the verison of the `zope.security` package shipped as part of the Zope 3.1.0 release.
- Clarify contract of `IChecker` to indicate that its `check*` methods may raise only `Forbidden` or `Unauthorized` exceptions.
- Add interfaces, (`IPrincipal`, `IGroupAwarePrincipal`, `IGroup`, and `IPermission`) specifying contracts of components in the security framework.
- Code style / documentation / test fixes.

3.0.0 (2004-11-07)

- Corresponds to the version of the `zope.security` package shipped as part of the Zope X3.0.0 release.

CHAPTER 2

API Reference

2.1 zope.security.interfaces

2.2 zope.security.adapter

2.3 zope.security.checker

2.3.1 Module API Documentation

2.3.2 API Doctests

Protections for Modules

The `moduleChecker()` API can be used to determine whether a module has been protected: Initially, there's no checker defined for the module:

```
>>> from zope.security.checker import moduleChecker
>>> from zope.security.tests import test_zcml_functest
>>> moduleChecker(test_zcml_functest) is None
True
```

We can add a checker using `zope.security.metaconfigure.protectModule()` (although this is more commonly done using ZCML):

```
>>> from zope.component import provideUtility
>>> from zope.security.metaconfigure import protectModule
>>> from zope.security.permission import Permission
>>> from zope.security.interfaces import IPermission
>>> TEST_PERM = 'zope.security.metaconfigure.test'
>>> perm = Permission(TEST_PERM, '')
```

(continues on next page)

(continued from previous page)

```
>>> provideUtility(perm, IPermission, TEST_PERM)
>>> protectModule(test_zcml_functest, 'foo', TEST_PERM)
```

Now, the checker should exist and have an access dictionary with the name and permission:

```
>>> def pprint(ob, width=70):
...     from pprint import PrettyPrinter
...     PrettyPrinter(width=width).pprint(ob)
>>> checker = moduleChecker(test_zcml_functest)
>>> cdict = checker.get_permissions
>>> pprint(cdict)
{'foo': 'zope.security.metaconfigure.test'}

If we define additional names, they will be added to the dict:

>>> protectModule(test_zcml_functest, 'bar', TEST_PERM)
>>> protectModule(test_zcml_functest, 'baz', TEST_PERM)
>>> pprint(cdict)
{'bar': 'zope.security.metaconfigure.test',
 'baz': 'zope.security.metaconfigure.test',
 'foo': 'zope.security.metaconfigure.test'}
```

The allow directive creates actions for each name defined directly, or via interface:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.security.metaconfigure import allow
>>> class I1(Interface):
...     def x(): pass
...     y = Attribute("Y")
>>> class I2(I1):
...     def a(): pass
...     b = Attribute("B")
>>> class AContext(object):
...     def __init__(self):
...         self.actions = []
...
...     def action(self, discriminator, callable, args):
...         self.actions.append(
...             {'discriminator': discriminator,
...              'callable': int(callable is protectModule),
...              'args': args})
...     module='testmodule'

>>> context = AContext()
>>> allow(context, attributes=['foo', 'bar'], interface=[I1, I2])
>>> context.actions.sort(key=lambda a: a['discriminator'])
>>> pprint(context.actions)
[{'args': ('testmodule', 'a', 'zope.Public'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'a')},
 {'args': ('testmodule', 'b', 'zope.Public'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
```

(continues on next page)

(continued from previous page)

```
                    'b')},
 {'args': ('testmodule', 'bar', 'zope.Public'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'bar')},
 {'args': ('testmodule', 'foo', 'zope.Public'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'foo')},
 {'args': ('testmodule', 'x', 'zope.Public'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'x')},
 {'args': ('testmodule', 'y', 'zope.Public'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'y')}]
```

The provide directive creates actions for each name defined directly, or via interface:

```
>>> from zope.security.metaconfigure import require
>>> class RContext(object):
...     def __init__(self):
...         self.actions = []
...     def action(self, discriminator, callable, args):
...         self.actions.append(
...             {'discriminator': discriminator,
...              'callable': int(callable is protectModule),
...              'args': args})
...     module='testmodule'

>>> context = RContext()
>>> require(context, attributes=['foo', 'bar'],
...         interface=[I1, I2], permission='p')

>>> context.actions.sort(key=lambda a: a['discriminator'])
>>> pprint(context.actions)
[{'args': ('testmodule', 'a', 'p'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'a')},
 {'args': ('testmodule', 'b', 'p'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'b')},
 {'args': ('testmodule', 'bar', 'p'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'bar')},
 {'args': ('testmodule', 'foo', 'p'),
```

(continues on next page)

(continued from previous page)

```
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'foo')},
 {'args': ('testmodule', 'x', 'p'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'x')},
 {'args': ('testmodule', 'y', 'p'),
  'callable': 1,
  'discriminator': ('http://namespaces.zope.org/zope:module',
                    'testmodule',
                    'y')}]
```

Protections for standard objects

```
>>> from zope.security.checker import ProxyFactory
>>> from zope.security.interfaces import ForbiddenAttribute
>>> def check_forbidden_get(object, attr):
...     from zope.security.interfaces import ForbiddenAttribute
...     try:
...         return getattr(object, attr)
...     except ForbiddenAttribute as e:
...         return 'ForbiddenAttribute: %s' % e.args[0]
>>> def check_forbidden_setitem(object, item, value):
...     from zope.security.interfaces import ForbiddenAttribute
...     try:
...         object[item] = value
...     except ForbiddenAttribute as e:
...         return 'ForbiddenAttribute: %s' % e.args[0]
>>> def check_forbidden_delitem(object, item):
...     from zope.security.interfaces import ForbiddenAttribute
...     try:
...         del object[item]
...     except ForbiddenAttribute as e:
...         return 'ForbiddenAttribute: %s' % e.args[0]
>>> def check_forbidden_call(callable, *args): # **
...     from zope.security.interfaces import ForbiddenAttribute
...     try:
...         return callable(*args) # **
...     except ForbiddenAttribute as e:
...         return 'ForbiddenAttribute: %s' % e.args[0]
```

Rocks

Rocks are immutable, non-callable objects without interesting methods. They *don't* get proxied.

```
>>> type(ProxyFactory(object())) is object
True
>>> type(ProxyFactory(1)) is int
True
>>> type(ProxyFactory(1.0)) is float
```

(continues on next page)

(continued from previous page)

```
True
>>> type(ProxyFactory(1j)) is complex
True
>>> type(ProxyFactory(None)) is type(None)
True
>>> type(ProxyFactory('xxx')) is str
True
>>> type(ProxyFactory(True)) is type(True)
True
```

Datetime-related instances are rocks, too:

```
>>> from datetime import timedelta, datetime, date, time, tzinfo
>>> type(ProxyFactory(  timedelta(1)  )) is timedelta
True
>>> type(ProxyFactory(  datetime(2000, 1, 1)  )) is datetime
True
>>> type(ProxyFactory(  date(2000, 1, 1)  )) is date
True
>>> type(ProxyFactory(  time()  )) is time
True
>>> type(ProxyFactory(  tzinfo() )) is tzinfo
True
>>> try:
...     from pytz import UTC
... except ImportError:  # pytz checker only if pytz is present.
...     True
... else:
...      type(ProxyFactory(  UTC )) is type(UTC)
True
```

dicts

We can do everything we expect to be able to do with proxied dicts.

```
>>> d = ProxyFactory({'a': 1, 'b': 2})
>>> check_forbidden_get(d, 'clear') # Verify that we are protected
'ForbiddenAttribute: clear'
>>> check_forbidden_setitem(d, 3, 4) # Verify that we are protected
'ForbiddenAttribute: __setitem__'
>>> d['a']
1
>>> len(d)
2
>>> sorted(list(d))
['a', 'b']
>>> d.get('a')
1
>>> 'a' in d
True
>>> c = d.copy()
>>> check_forbidden_get(c, 'clear')
'ForbiddenAttribute: clear'
>>> str(c) in ("{'a': 1, 'b': 2}", "{'b': 2, 'a': 1}")
True
```

(continues on next page)

(continued from previous page)

```
>>> repr(c) in ("{'a': 1, 'b': 2}", "{'b': 2, 'a': 1}")
True
>>> def sorted(x):
...     x = list(x)
...     x.sort()
...     return x
>>> sorted(d.keys())
['a', 'b']
>>> sorted(d.values())
[1, 2]
>>> sorted(d.items())
[('a', 1), ('b', 2)]
```

Always available (note, that dicts in python-3.x are not orderable, so we are not checking that under python > 2):

```
>>> d != d
False
>>> bool(d)
True
>>> d.__class__ == dict
True
```

lists

We can do everything we expect to be able to do with proxied lists.

```
>>> l = ProxyFactory([1, 2])
>>> check_forbidden_delitem(l, 0)
'ForbiddenAttribute: __delitem__'
>>> check_forbidden_setitem(l, 0, 3)
'ForbiddenAttribute: __setitem__'
>>> l[0]
1
>>> l[0:1]
[1]
>>> check_forbidden_setitem(l[:1], 0, 2)
'ForbiddenAttribute: __setitem__'
>>> len(l)
2
>>> tuple(l)
(1, 2)
>>> 1 in l
True
>>> l.index(2)
1
>>> l.count(2)
1
>>> str(l)
'[1, 2]'
>>> repr(l)
'[1, 2]'
>>> l + l
[1, 2, 1, 2]
```

Always available:

```
>>> l < l
False
>>> l > l
False
>>> l <= l
True
>>> l >= l
True
>>> l == l
True
>>> l != l
False
>>> bool(l)
True
>>> l.__class__ == list
True
```

tuples

We can do everything we expect to be able to do with proxied tuples.

```
>>> from zope.security.checker import ProxyFactory
>>> l = ProxyFactory((1, 2))
>>> l[0]
1
>>> l[0:1]
(1,)
>>> len(l)
2
>>> list(l)
[1, 2]
>>> 1 in l
True
>>> str(l)
'(1, 2)'
>>> repr(l)
'(1, 2)'
>>> l + l
(1, 2, 1, 2)
```

Always available:

```
>>> l < l
False
>>> l > l
False
>>> l <= l
True
>>> l >= l
True
>>> l == l
True
>>> l != l
False
>>> bool(l)
```

(continues on next page)

(continued from previous page)

```
True
>>> l.__class__ == tuple
True
```

sets

we can do everything we expect to be able to do with proxied sets.

```
>>> us = set((1, 2))
>>> s = ProxyFactory(us)

>>> check_forbidden_get(s, 'add') # Verify that we are protected
'ForbiddenAttribute: add'
>>> check_forbidden_get(s, 'remove') # Verify that we are protected
'ForbiddenAttribute: remove'
>>> check_forbidden_get(s, 'discard') # Verify that we are protected
'ForbiddenAttribute: discard'
>>> check_forbidden_get(s, 'pop') # Verify that we are protected
'ForbiddenAttribute: pop'
>>> check_forbidden_get(s, 'clear') # Verify that we are protected
'ForbiddenAttribute: clear'

>>> len(s)
2

>>> 1 in s
True

>>> 1 not in s
False

>>> s.issubset(set((1,2,3)))
True

>>> s.issuperset(set((1,2,3)))
False

>>> c = s.union(set((2, 3)))
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s | set((2, 3))
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s | ProxyFactory(set((2, 3)))
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'
```

(continues on next page)

(continued from previous page)

```
>>> c = set((2, 3)) | s
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.intersection(set((2, 3)))
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s & set((2, 3))
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s & ProxyFactory(set((2, 3)))
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = set((2, 3)) & s
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.difference(set((2, 3)))
>>> sorted(c)
[1]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s - ProxyFactory(set((2, 3)))
>>> sorted(c)
[1]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s - set((2, 3))
>>> sorted(c)
[1]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = set((2, 3)) - s
>>> sorted(c)
[3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.symmetric_difference(set((2, 3)))
>>> sorted(c)
[1, 3]
```

(continues on next page)

(continued from previous page)

```
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s ^ set((2, 3))
>>> sorted(c)
[1, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s ^ ProxyFactory(set((2, 3)))
>>> sorted(c)
[1, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = set((2, 3)) ^ s
>>> sorted(c)
[1, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.copy()
>>> sorted(c)
[1, 2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> str(s) == str(us)
True

>>> repr(s) == repr(us)
True

Always available:

>>> s < us
False
>>> s > us
False
>>> s <= us
True
>>> s >= us
True
>>> s == us
True
>>> s != us
False
```

Note that you can't compare proxied sets with other proxied sets due to a limitation in the set comparison functions which won't work with any kind of proxy.

```
>>> bool(s)
True
>>> s.__class__ == set
True
```

frozensets

we can do everything we expect to be able to do with proxied frozensets.

```
>>> def check_forbidden_get(object, attr):
...     from zope.security.interfaces import ForbiddenAttribute
...     try:
...         return getattr(object, attr)
...     except ForbiddenAttribute as e:
...         return 'ForbiddenAttribute: %s' % e.args[0]
>>> from zope.security.checker import ProxyFactory
>>> from zope.security.interfaces import ForbiddenAttribute
>>> us = frozenset((1, 2))
>>> s = ProxyFactory(us)

>>> check_forbidden_get(s, 'add') # Verify that we are protected
'ForbiddenAttribute: add'
>>> check_forbidden_get(s, 'remove') # Verify that we are protected
'ForbiddenAttribute: remove'
>>> check_forbidden_get(s, 'discard') # Verify that we are protected
'ForbiddenAttribute: discard'
>>> check_forbidden_get(s, 'pop') # Verify that we are protected
'ForbiddenAttribute: pop'
>>> check_forbidden_get(s, 'clear') # Verify that we are protected
'ForbiddenAttribute: clear'

>>> len(s)
2

>>> 1 in s
True

>>> 1 not in s
False

>>> s.issubset(frozenset((1,2,3)))
True

>>> s.issuperset(frozenset((1,2,3)))
False

>>> c = s.union(frozenset((2, 3)))
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s | frozenset((2, 3))
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s | ProxyFactory(frozenset((2, 3)))
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
```

(continues on next page)

(continued from previous page)

```
'ForbiddenAttribute: add'

>>> c = frozenset((2, 3)) | s
>>> sorted(c)
[1, 2, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.intersection(frozenset((2, 3)))
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s & frozenset((2, 3))
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s & ProxyFactory(frozenset((2, 3)))
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = frozenset((2, 3)) & s
>>> sorted(c)
[2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.difference(frozenset((2, 3)))
>>> sorted(c)
[1]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s - ProxyFactory(frozenset((2, 3)))
>>> sorted(c)
[1]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s - frozenset((2, 3))
>>> sorted(c)
[1]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = frozenset((2, 3)) - s
>>> sorted(c)
[3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.symmetric_difference(frozenset((2, 3)))
```

(continues on next page)

(continued from previous page)

```
>>> sorted(c)
[1, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s ^ frozenset((2, 3))
>>> sorted(c)
[1, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s ^ ProxyFactory(frozenset((2, 3)))
>>> sorted(c)
[1, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = frozenset((2, 3)) ^ s
>>> sorted(c)
[1, 3]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> c = s.copy()
>>> sorted(c)
[1, 2]
>>> check_forbidden_get(c, 'add')
'ForbiddenAttribute: add'

>>> str(s) == str(us)
True

>>> repr(s) == repr(us)
True

Always available:

>>> s < us
False
>>> s > us
False
>>> s <= us
True
>>> s >= us
True
>>> s == us
True
>>> s != us
False
```

Note that you can't compare proxied sets with other proxied sets due to a limitation in the frozenset comparison functions which won't work with any kind of proxy.

```
>>> bool(s)
True
>>> s.__class__ == frozenset
True
```

iterators

```
>>> [a for a in ProxyFactory(iter([1, 2]))]
[1, 2]
>>> list(ProxyFactory(iter([1, 2])))
[1, 2]
>>> list(ProxyFactory(iter((1, 2))))
[1, 2]
>>> list(ProxyFactory(iter({1:1, 2:2})))
[1, 2]
>>> def f():
...     for i in 1, 2:
...             yield i
...
>>> list(ProxyFactory(f()))
[1, 2]
>>> list(ProxyFactory(f)())
[1, 2]
```

We can iterate over custom sequences, too:

```
>>> class X(object):
...   d = 1, 2, 3
...   def __getitem__(self, i):
...      return self.d[i]
...
>>> x = X()
```

We can iterate over sequences

```
>>> list(x)
[1, 2, 3]

>>> from zope.security.checker import NamesChecker
>>> from zope.security.checker import ProxyFactory
>>> c = NamesChecker(['__getitem__', '__len__'])
>>> p = ProxyFactory(x, c)
```

Even if they are proxied

```
>>> list(p)
[1, 2, 3]
```

But if the class has an iter:

```
>>> X.__iter__ = lambda self: iter(self.d)
>>> list(x)
[1, 2, 3]
```

We shouldn't be able to iterate if we don't have an assertion:

```
>>> check_forbidden_call(list, p)
'ForbiddenAttribute: __iter__'
```

New-style classes

```
>>> from zope.security.checker import NamesChecker
>>> class C(object):
...     x = 1
...     y = 2
>>> C = ProxyFactory(C)
>>> check_forbidden_call(C)
'ForbiddenAttribute: __call__'
>>> check_forbidden_get(C, '__dict__')
'ForbiddenAttribute: __dict__'
>>> s = str(C)
>>> s = repr(C)
>>> C.__module__ == __name__
True
>>> len(C.__bases__)
1
>>> len(C.__mro__)
2
```

Always available:

```
>>> C == C
True
>>> C != C
False
>>> bool(C)
True
>>> C.__class__ == type
True
```

New-style Instances

```
>>> class C(object):
...     x = 1
...     y = 2
>>> c = ProxyFactory(C(), NamesChecker(['x']))
>>> check_forbidden_get(c, 'y')
'ForbiddenAttribute: y'
>>> check_forbidden_get(c, 'z')
'ForbiddenAttribute: z'
>>> c.x
1
>>> c.__class__ == C
True
```

Always available:

```
>>> c == c
True
>>> c != c
False
>>> bool(c)
True
```

(continues on next page)

(continued from previous page)

```
>>> c.__class__ == C
True
```

Classic Classes

```
>>> class C:
...     x = 1
>>> C = ProxyFactory(C)
>>> check_forbidden_call(C)
'ForbiddenAttribute: __call__'
>>> check_forbidden_get(C, '__dict__')
'ForbiddenAttribute: __dict__'
>>> s = str(C)
>>> s = repr(C)
>>> C.__module__ == __name__
True
```

Note that these are really only classic on Python 2:

```
>>> import sys
>>> len(C.__bases__) == (0 if sys.version_info[0] == 2 else 1)
True
```

Always available:

```
>>> C == C
True
>>> C != C
False
>>> bool(C)
True
```

Classic Instances

```
>>> class C(object):
...     x, y = 1, 2
>>> c = ProxyFactory(C(), NamesChecker(['x']))
>>> check_forbidden_get(c, 'y')
'ForbiddenAttribute: y'
>>> check_forbidden_get(c, 'z')
'ForbiddenAttribute: z'
>>> c.x
1
>>> c.__class__ == C
True
```

Always available:

```
>>> c == c
True
>>> c != c
False
```

(continues on next page)

(continued from previous page)

```
>>> bool(c)
True
>>> c.__class__ == C
True
```

Interfaces and declarations

We can still use interfaces though proxies:

```
>>> from zope.interface import directlyProvides
>>> from zope.interface import implementer
>>> from zope.interface import provider
>>> class I(Interface):
...     pass
>>> class IN(Interface):
...     pass
>>> class II(Interface):
...     pass
>>> @implementer(I)
... @provider(IN)
... class N(object):
...     pass
>>> n = N()
>>> directlyProvides(n, II)
>>> N = ProxyFactory(N)
>>> n = ProxyFactory(n)
>>> I.implementedBy(N)
True
>>> IN.providedBy(N)
True
>>> I.providedBy(n)
True
>>> II.providedBy(n)
True
```

Abstract Base Classes

We work with the ABCMeta meta class:

```
>>> import abc
>>> MyABC = abc.ABCMeta('MyABC', (object,), {})
>>> class Foo(MyABC): pass
>>> class Bar(Foo): pass
>>> PBar = ProxyFactory(Bar)
>>> [c.__name__ for c in PBar.__mro__]
['Bar', 'Foo', 'MyABC', 'object']
>>> check_forbidden_call(PBar)
'ForbiddenAttribute: __call__'
>>> check_forbidden_get(PBar, '__dict__')
'ForbiddenAttribute: __dict__'
>>> s = str(PBar)
>>> s = repr(PBar)
>>> PBar.__module__ == __name__
```

(continues on next page)

(continued from previous page)

```
True
>>> len(PBar.__bases__)
1
```

Always available:

```
>>> PBar == PBar
True
>>> PBar != PBar
False
>>> bool(PBar)
True
>>> PBar.__class__ == type
False
```

2.4 zope.security.decorator

2.4.1 API Examples

To illustrate, we'll create a class that will be proxied:

```
>>> class Foo(object):
...     a = 'a'
```

and a class to proxy it that uses a decorated security checker:

```
>>> from zope.security.decorator import DecoratedSecurityCheckerDescriptor
>>> from zope.proxy import ProxyBase
>>> class Wrapper(ProxyBase):
...     b = 'b'
...     __Security_checker__ = DecoratedSecurityCheckerDescriptor()
```

Next we'll create and register a checker for `Foo`:

```
>>> from zope.security.checker import NamesChecker, defineChecker
>>> fooChecker = NamesChecker(['a'])
>>> defineChecker(Foo, fooChecker)
```

along with a checker for `Wrapper`:

```
>>> wrapperChecker = NamesChecker(['b'])
>>> defineChecker(Wrapper, wrapperChecker)
```

Using `zope.security.checker.selectChecker()`, we can confirm that a `Foo` object uses `fooChecker`:

```
>>> from zope.security.checker import selectChecker
>>> from zope.security.interfaces import ForbiddenAttribute
>>> foo = Foo()
>>> selectChecker(foo) is fooChecker
True
>>> fooChecker.check(foo, 'a')
>>> try:
```

(continues on next page)

(continued from previous page)

```
...     fooChecker.check(foo, 'b')
... except ForbiddenAttribute as e:
...     e
ForbiddenAttribute('b', <...Foo object ...>)
```

and that a `Wrapper` object uses `wrappeChecker`:

```
>>> wrapper = Wrapper(foo)
>>> selectChecker(wrapper) is wrapperChecker
True
>>> wrapperChecker.check(wrapper, 'b')
>>> try:
...     wrapperChecker.check(wrapper, 'a')
... except ForbiddenAttribute as e:
...     e
ForbiddenAttribute('a', <...Foo object ...>)
```

(Note that the object description says *Foo* because the object is a proxy and generally looks and acts like the object it's proxying.)

When we access wrapper's `__Security_checker__` attribute, we invoke the decorated security checker descriptor. The decorator's job is to make sure checkers from both objects are used when available. In this case, because both objects have checkers, we get a combined checker:

```
>>> from zope.security.checker import CombinedChecker
>>> checker = wrapper.__Security_checker__
>>> type(checker)
<class 'zope.security.checker.CombinedChecker'>
>>> checker.check(wrapper, 'a')
>>> checker.check(wrapper, 'b')
```

The decorator checker will work even with security proxied objects. To illustrate, we'll proxify `foo`:

```
>>> from zope.security.proxy import ProxyFactory
>>> secure_foo = ProxyFactory(foo)
>>> secure_foo.a
'a'
>>> try:
...     secure_foo.b
... except ForbiddenAttribute as e:
...     e
ForbiddenAttribute('b', <...Foo object ...>)
```

when we wrap the secured `foo`:

```
>>> wrapper = Wrapper(secure_foo)
```

we still get a combined checker:

```
>>> checker = wrapper.__Security_checker__
>>> type(checker)
<class 'zope.security.checker.CombinedChecker'>
>>> checker.check(wrapper, 'a')
>>> checker.check(wrapper, 'b')
```

The decorator checker has three other scenarios:

- the wrapper has a checker but the proxied object doesn't

- the proxied object has a checker but the wrapper doesn't
- neither the wrapper nor the proxied object have checkers

When the wrapper has a checker but the proxied object doesn't:

```
>>> from zope.security.checker import NoProxy, _checkers
>>> del _checkers[Foo]
>>> defineChecker(Foo, NoProxy)
>>> selectChecker(foo) is None
True
>>> selectChecker(wrapper) is wrapperChecker
True
```

the decorator uses only the wrapper checker:

```
>>> wrapper = Wrapper(foo)
>>> wrapper.__Security_checker__ is wrapperChecker
True
```

When the proxied object has a checker but the wrapper doesn't:

```
>>> del _checkers[Wrapper]
>>> defineChecker(Wrapper, NoProxy)
>>> selectChecker(wrapper) is None
True
>>> del _checkers[Foo]
>>> defineChecker(Foo, fooChecker)
>>> selectChecker(foo) is fooChecker
True
```

the decorator uses only the proxied object checker:

```
>>> wrapper.__Security_checker__ is fooChecker
True
```

Finally, if neither the wrapper not the proxied have checkers:

```
>>> del _checkers[Foo]
>>> defineChecker(Foo, NoProxy)
>>> selectChecker(foo) is None
True
>>> selectChecker(wrapper) is None
True
```

the decorator doesn't have a checker:

```
>>> wrapper.__Security_checker__
Traceback (most recent call last):
  ...
AttributeError: 'Foo' has no attribute '__Security_checker__'
```

`__Security_checker__` cannot be None, otherwise Checker.proxy blows up:

```
>>> checker.proxy(wrapper) is wrapper
True
```

2.5 zope.security.management

2.6 zope.security.permission

```
>>> from zope.security.permission import checkPermission
>>> from zope.component import provideUtility
>>> from zope.security.interfaces import IPermission
>>> from zope.security.permission import Permission
>>> x = Permission('x')
>>> provideUtility(x, IPermission, 'x')

>>> checkPermission(None, 'x')
>>> checkPermission(None, 'y')
Traceback (most recent call last):
...
ValueError: ('Undefined permission ID', 'y')
```

The `zope.security.checker.CheckerPublic` permission always exists:

```
>>> from zope.security.checker import CheckerPublic
>>> checkPermission(None, CheckerPublic)
```

```
>>> from zope.security.permission import allPermissions
>>> from zope.component import provideUtility
>>> y = Permission('y')
>>> provideUtility(y, IPermission, 'y')

>>> ids = sorted(allPermissions(None))
>>> for perm in sorted(allPermissions(None)):
...     print(perm)
x
y
```

To illustrate, we need to register the permissions vocabulary:

```
>>> from zope.security.permission import PermissionsVocabulary
>>> from zope.schema.vocabulary import _clear
>>> _clear()

>>> from zope.schema.vocabulary import getVocabularyRegistry
>>> registry = getVocabularyRegistry()
>>> registry.register('Permissions', PermissionsVocabulary)
```

We can now lookup the permissions we created earlier using the vocabulary:

```
>>> vocab = registry.get(None, 'Permissions')
>>> vocab.getTermByToken('x').value is x
True
>>> vocab.getTermByToken('y').value is y
True
```

To illustrate, we need to register the permission IDs vocabulary:

```
>>> from zope.security.permission import PermissionIdsVocabulary
>>> registry.register('Permission Ids', PermissionIdsVocabulary)
```

(continues on next page)

(continued from previous page)

```
We also need to register the special 'zope.Public' permission:

>>> provideUtility(Permission('zope.Public'), IPermission, 'zope.Public')
```

We can now lookup these permissions using the vocabulary:

```
>>> vocab = registry.get(None, 'Permission Ids')
```

The non-public permissions 'x' and 'y' are string values:

```
>>> print(vocab.getTermByToken('x').value)
x
>>> print(vocab.getTermByToken('y').value)
y
```

However, the public permission value is `CheckerPublic`:

```
>>> vocab.getTermByToken('zope.Public').value is CheckerPublic
True
```

and its title is shortened:

```
>>> print(vocab.getTermByToken('zope.Public').title)
Public
```

The terms are sorted by title except for the public permission, which is listed first:

```
>>> for term in vocab:
...     print(term.title)
Public
x
y
```

2.7 zope.security.protectclass

2.8 zope.security.proxy

An introduction to proxies and their uses can be found in *Untrusted Interpreters and Security Proxies*.

See also:

Known Issues With Proxies

```
>>> from zope.security.proxy import isinstance
>>> class C1(object):
...     pass

>>> c = C1()
>>> isinstance(c, C1)
True

>>> from zope.security.checker import ProxyFactory
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(ProxyFactory(c), C1)
True

>>> class C2(C1):
...     pass

>>> c = C2()
>>> isinstance(c, C1)
True

>>> from zope.security.checker import ProxyFactory
>>> isinstance(ProxyFactory(c), C1)
True
```

2.9 zope.security.simplepolicies

2.10 zope.security.testing

2.11 zope.security.metaconfigure

2.12 zope.security.zcml

Most users will not directly need to access the contents of this module; they will probably just *configure via ZCML*.

2.12.1 API Reference

Let's look at an example:

```
>>> from zope.security.zcml import Permission
>>> class FauxContext(object):
...     permission_mapping = {'zope.ManageCode':'zope.private'}
...     _actions = []
...     def action(self, **kws):
...         self._actions.append(kws)
>>> context = FauxContext()
>>> field = Permission().bind(context)
```

Let's test the fromUnicode method:

```
>>> field.fromUnicode(u'zope.foo')
'zope.foo'
>>> field.fromUnicode(u'zope.ManageCode')
'zope.private'
```

Now let's see whether validation works alright

```
>>> field._validate('zope.ManageCode')
>>> context._actions[0]['args']
(None, 'zope.foo')
```

(continues on next page)

(continued from previous page)

```
>>> from zope.schema.interfaces import InvalidId
>>> try:
...     field._validate('3 foo')
... except InvalidId as e:
...     e
InvalidId('3 foo')

zope.Public is always valid
>>> field._validate('zope.Public')
```

2.12.2 Configuring security via ZCML

`zope.security` provides a ZCML file that configures some utilities and a couple of standard permissions:

```
>>> from zope.component import getGlobalSiteManager
>>> from zope.configuration.xmlconfig import XMLConfig
>>> from zope.component.testing import setUp
>>> import zope.security
>>> setUp()  # clear global component registry
>>> XMLConfig('permissions.zcml', zope.security)()

>>> len(list(getGlobalSiteManager().registeredUtilities()))
7
```

Clear the current state:

```
>>> from zope.component.testing import setUp, tearDown
>>> tearDown()
>>> setUp()

>>> XMLConfig('configure.zcml', zope.security)()

>>> len(list(getGlobalSiteManager().registeredUtilities()))
10
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search