```
**********************************************************
   29993 Mon Sep 22 23:59:11 2014
new/usr/src/tools/scripts/validate_pkg.py
5189 validate_pkg should support mediated links
Reviewed by: Richard Lowe <richlowe@richlowe.net>
Reviewed by: Josef 'Jeff' Sipek <josef.sipek@nexenta.com>
**********************************************************
```

```
#!/usr/bin/python2.6
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

#
# Compare the content generated by a build to a set of manifests
# describing how that content is to be delivered.
#


import getopt
import os
import stat
import sys

from pkg import actions
from pkg import manifest


#
# Dictionary used to map action names to output format.  Each entry is
# indexed by action name, and consists of a list of tuples that map
# FileInfo class members to output labels.
#
OUTPUTMAP = {
    "dir": [
        ("group", "group="),
        ("mode", "mode="),
        ("owner", "owner="),
        ("path", "path=")
    ],
    "file": [
        ("hash", ""),
        ("group", "group="),
        ("mode", "mode="),
        ("owner", "owner="),
```



```
        ("path", "path=")
    ],
    "link": [
        ("mediator", "mediator="),
#endif /* ! codereview */
        ("path", "path="),
        ("target", "target=")
    ],
    "hardlink": [
        ("path", "path="),
        ("hardkey", "target=")
    ],
}

# Mode checks used to validate safe file and directory permissions
ALLMODECHECKS = frozenset(("m", "w", "s", "o"))
DEFAULTMODECHECKS = frozenset(("m", "w", "o"))

class FileInfo(object):
    """Base class to represent a file.

    Subclassed according to whether the file represents an actual filesystem
    object (RealFileInfo) or an IPS manifest action (ActionInfo).
    """

    def __init__(self):
        self.path = None
        self.isdir = False
        self.target = None
        self.owner = None
        self.group = None
        self.mode = None
        self.hardkey = None
        self.hardpaths = set()
        self.editable = False

    def name(self):
        """Return the IPS action name of a FileInfo object.
        """
        if self.isdir:
            return "dir"

        if self.target:
            return "link"

        if self.hardkey:
            return "hardlink"

        return "file"

    def checkmodes(self, modechecks):
        """Check for and report on unsafe permissions.

        Returns a potentially empty list of warning strings.
        """
        w = []

        t = self.name()
        if t in ("link", "hardlink"):
            return w
        m = int(self.mode, 8)
        o = self.owner
        p = self.path

        if "s" in modechecks and t == "file":
            if m & (stat.S_ISUID | stat.S_ISGID):
```

```
126                 if m & (stat.S_IRGRP | stat.S_IROTH):
127                     w.extend(["%s: 0%o: setuid/setgid file should not be " \
128                         "readable by group or other" % (p, m)])

130         if "o" in modechecks and o != "root" and ((m & stat.S_ISUID) == 0):
131             mu = (m & stat.S_IRWXU) >> 6
132             mg = (m & stat.S_IRWXG) >> 3
133             mo = m & stat.S_IRWXO
134             e = self.editable

136             if (((mu & 02) == 0 and (mo & mg & 04) == 04) or
137                 (t == "file" and mo & 01 == 1) or
138                 (mg, mo) == (mu, mu) or
139                 ((t == "file" and not e or t == "dir" and o == "bin") and
140                 (mg & 05 == mo & 05)) or
141                 (t == "file" and o == "bin" and mu & 01 == 01) or
142                 (m & 0105 != 0 and p.startswith("etc/security/dev/"))):
143                 w.extend(["%s: owner \"%s\" may be safely " \
144                     "changed to \"root\"" % (p, o)])

146         if "w" in modechecks and t == "file" and o != "root":
147             uwx = stat.S_IWUSR | stat.S_IXUSR
148             if m & uwx == uwx:
149                 w.extend(["%s: non-root-owned executable should not " \
150                     "also be writable by owner." % p])

152         if ("m" in modechecks and
153             m & (stat.S_IWGRP | stat.S_IWOTH) != 0 and
154             m & stat.S_ISVTX == 0):
155             w.extend(["%s: 0%o: should not be writable by group or other" %
156                 (p, m)])

158         return w

160     def __ne__(self, other):
161         """Compare two FileInfo objects.

163         Note this is the "not equal" comparison, so a return value of False
164         indicates that the objects are functionally equivalent.
165         """
166         #
167         # Map the objects such that the lhs is always the ActionInfo,
168         # and the rhs is always the RealFileInfo.
169         #
170         # It's only really important that the rhs not be an
171         # ActionInfo; if we're comparing FileInfo the RealFileInfo, it
172         # won't actually matter what we choose.
173         #
174         if isinstance(self, ActionInfo):
175             lhs = self
176             rhs = other
177         else:
178             lhs = other
179             rhs = self

181         #
182         # Because the manifest may legitimately translate a relative
183         # path from the proto area into a different path on the installed
184         # system, we don't compare paths here.  We only expect this comparison
185         # to be invoked on items with identical relative paths in
186         # first place.
187         #

189         #
190         # All comparisons depend on type.  For symlink and directory, they
191         # must be the same.  For file and hardlink, see below.
```



```
192         #
193         typelhs = lhs.name()
194         typerhs = rhs.name()
195         if typelhs in ("link", "dir"):
196             if typelhs != typerhs:
197                 return True

199         #
200         # For symlinks, all that's left is the link target.
201         # For mediated symlinks targets can differ.
202 #endif /* ! codereview */
203         #
204         if typelhs == "link":
205             return (lhs.mediator is None) and (lhs.target != rhs.target)
 63             return lhs.target != rhs.target

207         #
208         # For a directory, it's important that both be directories,
209         # the modes be identical, and the paths are identical.  We already
210         # checked all but the modes above.
211         #
212         # If both objects are files, then we're in the same boat.
213         #
214         if typelhs == "dir" or (typelhs == "file" and typerhs == "file"):
215             return lhs.mode != rhs.mode

217         #
218         # For files or hardlinks:
219         #
220         # Since the key space is different (inodes for real files and
221         # actual link targets for hard links), and since the proto area will
222         # identify all N occurrences as hardlinks, but the manifests as one
223         # file and N-1 hardlinks, we have to compare files to hardlinks.
224         #

226         #
227         # If they're both hardlinks, we just make sure that
228         # the same target path appears in both sets of
229         # possible targets.
230         #
231         if typelhs == "hardlink" and typerhs == "hardlink":
232             return len(lhs.hardpaths.intersection(rhs.hardpaths)) == 0

234         #
235         # Otherwise, we have a mix of file and hardlink, so we
236         # need to make sure that the file path appears in the
237         # set of possible target paths for the hardlink.
238         #
239         # We already know that the ActionInfo, if present, is the lhs
240         # operator.  So it's the rhs operator that's guaranteed to
241         # have a set of hardpaths.
242         #
243         return lhs.path not in rhs.hardpaths

245     def __str__(self):
246         """Return an action-style representation of a FileInfo object.

248         We don't currently quote items with embedded spaces.  If we
249         ever decide to parse this output, we'll want to revisit that.
250         """
251         name = self.name()
252         out = name

254         for member, label in OUTPUTMAP[name]:
255             out += " " + label + str(getattr(self, member))
```

```
257         return out

259     def protostr(self):
260         """Return a protolist-style representation of a FileInfo object.
261         """
262         target = "-"
263         major = "-"
264         minor = "-"

266         mode = self.mode
267         owner = self.owner
268         group = self.group

270         name = self.name()
271         if name == "dir":
272             ftype = "d"
273         elif name in ("file", "hardlink"):
274             ftype = "f"
275         elif name == "link":
276             ftype = "s"
277             target = self.target
278             mode = "777"
279             owner = "root"
280             group = "other"

282         out = "%c %-30s %-20s %4s %-5s %-5s %6d %2ld  -  -" % \
283             (ftype, self.path, target, mode, owner, group, 0, 1)

285         return out


288 class ActionInfo(FileInfo):
289     """Object to track information about manifest actions.

291     This currently understands file, link, dir, and hardlink actions.
292     """

294     def __init__(self, action):
295         FileInfo.__init__(self)
296         #
297         # Currently, all actions that we support have a "path"
298         # attribute.  If that changes, then we'll need to
299         # catch a KeyError from this assignment.
300         #
301         self.path = action.attrs["path"]

303         if action.name == "file":
304             self.owner = action.attrs["owner"]
305             self.group = action.attrs["group"]
306             self.mode = action.attrs["mode"]
307             self.hash = action.hash
308             if "preserve" in action.attrs:
309                 self.editable = True
310         elif action.name == "link":
311             target = action.attrs["target"]
312             self.target = os.path.normpath(target)
313             self.mediator = action.attrs.get("mediator")
314 #endif /* ! codereview */
315         elif action.name == "dir":
316             self.owner = action.attrs["owner"]
317             self.group = action.attrs["group"]
318             self.mode = action.attrs["mode"]
319             self.isdir = True
320         elif action.name == "hardlink":
321             target = os.path.normpath(action.get_target_path())
322             self.hardkey = target
```



```
323             self.hardpaths.add(target)

325     @staticmethod
326     def supported(action):
327         """Indicates whether the specified IPS action time is
328         correctly handled by the ActionInfo constructor.
329         """
330         return action in frozenset(("file", "dir", "link", "hardlink"))


333 class UnsupportedFileFormatError(Exception):
334     """This means that the stat.S_IFMT returned something we don't
335     support, ie a pipe or socket.  If it's appropriate for such an
336     object to be in the proto area, then the RealFileInfo constructor
337     will need to evolve to support it, or it will need to be in the
338     exception list.
339     """
340     def __init__(self, path, mode):
341         Exception.__init__(self)
342         self.path = path
343         self.mode = mode

345     def __str__(self):
346         return '%s: unsupported S_IFMT %07o' % (self.path, self.mode)


349 class RealFileInfo(FileInfo):
350     """Object to track important-to-packaging file information.

352     This currently handles regular files, directories, and symbolic links.

354     For multiple RealFileInfo objects with identical hardkeys, there
355     is no way to determine which of the hard links should be
356     delivered as a file, and which as hardlinks.
357     """

359     def __init__(self, root=None, path=None):
360         FileInfo.__init__(self)
361         self.path = path
362         path = os.path.join(root, path)
363         lstat = os.lstat(path)
364         mode = lstat.st_mode

366         #
367         # Per stat.py, these cases are mutually exclusive.
368         #
369         if stat.S_ISREG(mode):
370             self.hash = self.path
371         elif stat.S_ISDIR(mode):
372             self.isdir = True
373         elif stat.S_ISLNK(mode):
374             self.target = os.path.normpath(os.readlink(path))
375             self.mediator = None
376 #endif /* ! codereview */
377         else:
378             raise UnsupportedFileFormatError(path, mode)

380         if not stat.S_ISLNK(mode):
381             self.mode = "%04o" % stat.S_IMODE(mode)
382             #
383             # Instead of reading the group and owner from the proto area after
384             # a non-root build, just drop in dummy values.  Since we don't
385             # compare them anywhere, this should allow at least marginally
386             # useful comparisons of protolist-style output.
387             #
388             self.owner = "owner"
```

```
389             self.group = "group"

391         #
392         # refcount > 1 indicates a hard link
393         #
394         if lstat.st_nlink > 1:
395             #
396             # This could get ugly if multiple proto areas reside
397             # on different filesystems.
398             #
399             self.hardkey = lstat.st_ino


402 class DirectoryTree(dict):
403     """Meant to be subclassed according to population method.
404     """
405     def __init__(self, name):
406         dict.__init__(self)
407         self.name = name

409     def compare(self, other):
410         """Compare two different sets of FileInfo objects.
411         """
412         keys1 = frozenset(self.keys())
413         keys2 = frozenset(other.keys())

415         common = keys1.intersection(keys2)
416         onlykeys1 = keys1.difference(common)
417         onlykeys2 = keys2.difference(common)

419         if onlykeys1:
420             print "Entries present in %s but not %s:" % \
421                 (self.name, other.name)
422             for path in sorted(onlykeys1):
423                 print("\t%s" % str(self[path]))
424             print ""

426         if onlykeys2:
427             print "Entries present in %s but not %s:" % \
428                 (other.name, self.name)
429             for path in sorted(onlykeys2):
430                 print("\t%s" % str(other[path]))
431             print ""

433         nodifferences = True
434         for path in sorted(common):
435             if self[path] != other[path]:
436                 if nodifferences:
437                     nodifferences = False
438                     print "Entries that differ between %s and %s:" \
439                         % (self.name, other.name)
440                 print("%14s %s" % (self.name, self[path]))
441                 print("%14s %s" % (other.name, other[path]))
442         if not nodifferences:
443             print ""


446 class BadProtolistFormat(Exception):
447     """This means that the user supplied a file via -l, but at least
448     one line from that file doesn't have the right number of fields to
449     parse as protolist output.
450     """
451     def __str__(self):
452         return 'bad proto list entry: "%s"' % Exception.__str__(self)
```



```
455 class ProtoTree(DirectoryTree):
456     """Describes one or more proto directories as a dictionary of
457     RealFileInfo objects, indexed by relative path.
458     """

460     def adddir(self, proto, exceptions):
461         """Extends the ProtoTree dictionary with RealFileInfo
462         objects describing the proto dir, indexed by relative
463         path.
464         """
465         newentries = {}

467         pdir = os.path.normpath(proto)
468         strippdir = lambda r, n: os.path.join(r, n)[len(pdir)+1:]
469         for root, dirs, files in os.walk(pdir):
470             for name in dirs + files:
471                 path = strippdir(root, name)
472                 if path not in exceptions:
473                     try:
474                         newentries[path] = RealFileInfo(pdir, path)
475                     except OSError, e:
476                         sys.stderr.write("Warning: unable to stat %s: %s\n" %
477                             (path, e))
478                         continue
479                 else:
480                     exceptions.remove(path)
481                     if name in dirs:
482                         dirs.remove(name)

484         #
485         # Find the sets of paths in this proto dir that are hardlinks
486         # to the same inode.
487         #
488         # It seems wasteful to store this in each FileInfo, but we
489         # otherwise need a linking mechanism.  With this information
490         # here, FileInfo object comparison can be self contained.
491         #
492         # We limit this aggregation to a single proto dir, as
493         # represented by newentries.  That means we don't need to care
494         # about proto dirs on separate filesystems, or about hardlinks
495         # that cross proto dir boundaries.
496         #
497         hk2path = {}
498         for path, fileinfo in newentries.iteritems():
499             if fileinfo.hardkey:
500                 hk2path.setdefault(fileinfo.hardkey, set()).add(path)
501         for fileinfo in newentries.itervalues():
502             if fileinfo.hardkey:
503                 fileinfo.hardpaths.update(hk2path[fileinfo.hardkey])
504         self.update(newentries)

506     def addprotolist(self, protolist, exceptions):
507         """Read in the specified file, assumed to be the
508         output of protolist.

510         This has been tested minimally, and is potentially useful for
511         comparing across the transition period, but should ultimately
512         go away.
513         """

515         try:
516             plist = open(protolist)
517         except IOError, exc:
518             raise IOError("cannot open proto list: %s" % str(exc))

520         newentries = {}
```

```
522         for pline in plist:
523             pline = pline.split()
524             #
525             # Use a FileInfo() object instead of a RealFileInfo()
526             # object because we want to avoid the RealFileInfo
527             # constructor, because there's nothing to actually stat().
528             #
529             fileinfo = FileInfo()
530             try:
531                 if pline[1] in exceptions:
532                     exceptions.remove(pline[1])
533                     continue
534                 if pline[0] == "d":
535                     fileinfo.isdir = True
536                 fileinfo.path = pline[1]
537                 if pline[2] != "-":
538                     fileinfo.target = os.path.normpath(pline[2])
539                 fileinfo.mode = int("0%s" % pline[3])
540                 fileinfo.owner = pline[4]
541                 fileinfo.group = pline[5]
542                 if pline[6] != "0":
543                     fileinfo.hardkey = pline[6]
544                 newentries[pline[1]] = fileinfo
545             except IndexError:
546                 raise BadProtolistFormat(pline)

548         plist.close()
549         hk2path = {}
550         for path, fileinfo in newentries.iteritems():
551             if fileinfo.hardkey:
552                 hk2path.setdefault(fileinfo.hardkey, set()).add(path)
553         for fileinfo in newentries.itervalues():
554             if fileinfo.hardkey:
555                 fileinfo.hardpaths.update(hk2path[fileinfo.hardkey])
556         self.update(newentries)


559 class ManifestParsingError(Exception):
560     """This means that the Manifest.set_content() raised an
561     ActionError.  We raise this, instead, to tell us which manifest
562     could not be parsed, rather than what action error we hit.
563     """
564     def __init__(self, mfile, error):
565         Exception.__init__(self)
566         self.mfile = mfile
567         self.error = error

569     def __str__(self):
570         return "unable to parse manifest %s: %s" % (self.mfile, self.error)


573 class ManifestTree(DirectoryTree):
574     """Describes one or more directories containing arbitrarily
575     many manifests as a dictionary of ActionInfo objects, indexed
576     by the relative path of the data source within the proto area.
577     That path may or may not be the same as the path attribute of the
578     given action.
579     """

581     def addmanifest(self, root, mfile, arch, modechecks, exceptions):
582         """Treats the specified input file as a pkg(5) package
583         manifest, and extends the ManifestTree dictionary with entries
584         for the actions therein.
585         """
586         mfest = manifest.Manifest()
```



```
587         try:
588             mfest.set_content(open(os.path.join(root, mfile)).read())
589         except IOError, exc:
590             raise IOError("cannot read manifest: %s" % str(exc))
591         except actions.ActionError, exc:
592             raise ManifestParsingError(mfile, str(exc))

594         #
595         # Make sure the manifest is applicable to the user-specified
596         # architecture.  Assumption: if variant.arch is not an
597         # attribute of the manifest, then the package should be
598         # installed on all architectures.
599         #
600         if arch not in mfest.attributes.get("variant.arch", (arch,)):
601             return

603         modewarnings = set()
604         for action in mfest.gen_actions():
605             if "path" not in action.attrs or \
606                 not ActionInfo.supported(action.name):
607                 continue

609             #
610             # The dir action is currently fully specified, in that it
611             # lists owner, group, and mode attributes.  If that
612             # changes in pkg(5) code, we'll need to revisit either this
613             # code or the ActionInfo() constructor.  It's possible
614             # that the pkg(5) system could be extended to provide a
615             # mechanism for specifying directory permissions outside
616             # of the individual manifests that deliver files into
617             # those directories.  Doing so at time of manifest
618             # processing would mean that validate_pkg continues to work,
619             # but doing so at time of publication would require updates.
620             #

622             #
623             # See pkgsend(1) for the use of NOHASH for objects with
624             # datastreams.  Currently, that means "files," but this
625             # should work for any other such actions.
626             #
627             if getattr(action, "hash", "NOHASH") != "NOHASH":
628                 path = action.hash
629             else:
630                 path = action.attrs["path"]

632             #
633             # This is the wrong tool in which to enforce consistency
634             # on a set of manifests.  So instead of comparing the
635             # different actions with the same "path" attribute, we
636             # use the first one.
637             #
638             if path in self:
639                 continue

641             #
642             # As with the manifest itself, if an action has specified
643             # variant.arch, we look for the target architecture
644             # therein.
645             #
646             var = None

648             #
649             # The name of this method changed in pkg(5) build 150, we need to
650             # work with both sets.
651             #
652             if hasattr(action, 'get_variants'):
```

```
653                 var = action.get_variants()
654             else:
655                 var = action.get_variant_template()
656             if "variant.arch" in var and arch not in var["variant.arch"]:
657                 return

659             self[path] = ActionInfo(action)
660             if modechecks is not None and path not in exceptions:
661                 modewarnings.update(self[path].checkmodes(modechecks))

663         if len(modewarnings) > 0:
664             print "warning: unsafe permissions in %s" % mfile
665             for w in sorted(modewarnings):
666                 print w
667             print ""

669     def adddir(self, mdir, arch, modechecks, exceptions):
670         """Walks the specified directory looking for pkg(5) manifests.
671         """
672         for mfile in os.listdir(mdir):
673             if (mfile.endswith(".mog") and
674                 stat.S_ISREG(os.lstat(os.path.join(mdir, mfile)).st_mode)):
675                 try:
676                     self.addmanifest(mdir, mfile, arch, modechecks, exceptions)
677                 except IOError, exc:
678                     sys.stderr.write("warning: %s\n" % str(exc))

680     def resolvehardlinks(self):
681         """Populates mode, group, and owner for resolved (ie link target
682         is present in the manifest tree) hard links.
683         """
684         for info in self.values():
685             if info.name() == "hardlink":
686                 tgt = info.hardkey
687                 if tgt in self:
688                     tgtinfo = self[tgt]
689                     info.owner = tgtinfo.owner
690                     info.group = tgtinfo.group
691                     info.mode = tgtinfo.mode

693 class ExceptionList(set):
694     """Keep track of an exception list as a set of paths to be excluded
695     from any other lists we build.
696     """

698     def __init__(self, files, arch):
699         set.__init__(self)
700         for fname in files:
701             try:
702                 self.readexceptionfile(fname, arch)
703             except IOError, exc:
704                 sys.stderr.write("warning: cannot read exception file: %s\n" %
705                     str(exc))

707     def readexceptionfile(self, efile, arch):
708         """Build a list of all pathnames from the specified file that
709         either apply to all architectures (ie which have no trailing
710         architecture tokens), or to the specified architecture (ie
711         which have the value of the arch arg as a trailing
712         architecture token.)
713         """

715         excfile = open(efile)

717         for exc in excfile:
718             exc = exc.split()
```



```
719             if len(exc) and exc[0][0] != "#":
720                 if arch in (exc[1:] or arch):
721                     self.add(os.path.normpath(exc[0]))

723         excfile.close()


726 USAGE = """%s [-v] -a arch [-e exceptionfile]... [-L|-M [-X check]...] input_1 [

728 where input_1 and input_2 may specify proto lists, proto areas,
729 or manifest directories.  For proto lists, use one or more

731     -l file

733 arguments.  For proto areas, use one or more

735     -p dir

737 arguments.  For manifest directories, use one or more

739     -m dir

741 arguments.

743 If -L or -M is specified, then only one input source is allowed, and
744 it should be one or more manifest directories.  These two options are
745 mutually exclusive.

747 The -L option is used to generate a proto list to stdout.

749 The -M option is used to check for safe file and directory modes.
750 By default, this causes all mode checks to be performed.  Individual
751 mode checks may be turned off using "-X check," where "check" comes
752 from the following set of checks:

754     m   check for group or other write permissions
755     w   check for user write permissions on files and directories
756         not owned by root
757     s   check for group/other read permission on executable files
758         that have setuid/setgid bit(s)
759     o   check for files that could be safely owned by root
760 """ % sys.argv[0]


763 def usage(msg=None):
764     """Try to give the user useful information when they don't get the
765     command syntax right.
766     """
767     if msg:
768         sys.stderr.write("%s: %s\n" % (sys.argv[0], msg))
769     sys.stderr.write(USAGE)
770     sys.exit(2)


773 def main(argv):
774     """Compares two out of three possible data sources: a proto list, a
775     set of proto areas, and a set of manifests.
776     """
777     try:
778         opts, args = getopt.getopt(argv, 'a:e:Ll:Mm:p:vX:')
779     except getopt.GetoptError, exc:
780         usage(str(exc))

782     if args:
783         usage()
```

```
785     arch = None
786     exceptionlists = []
787     listonly = False
788     manifestdirs = []
789     manifesttree = ManifestTree("manifests")
790     protodirs = []
791     prototree = ProtoTree("proto area")
792     protolists = []
793     protolist = ProtoTree("proto list")
794     modechecks = set()
795     togglemodechecks = set()
796     trees = []
797     comparing = set()
798     verbose = False

800     for opt, arg in opts:
801         if opt == "-a":
802             if arch:
803                 usage("may only specify one architecture")
804             else:
805                 arch = arg
806         elif opt == "-e":
807             exceptionlists.append(arg)
808         elif opt == "-L":
809             listonly = True
810         elif opt == "-l":
811             comparing.add("protolist")
812             protolists.append(os.path.normpath(arg))
813         elif opt == "-M":
814             modechecks.update(DEFAULTMODECHECKS)
815         elif opt == "-m":
816             comparing.add("manifests")
817             manifestdirs.append(os.path.normpath(arg))
818         elif opt == "-p":
819             comparing.add("proto area")
820             protodirs.append(os.path.normpath(arg))
821         elif opt == "-v":
822             verbose = True
823         elif opt == "-X":
824             togglemodechecks.add(arg)

826     if listonly or len(modechecks) > 0:
827         if len(comparing) != 1 or "manifests" not in comparing:
828             usage("-L and -M require one or more -m args, and no -l or -p")
829         if listonly and len(modechecks) > 0:
830             usage("-L and -M are mutually exclusive")
831     elif len(comparing) != 2:
832         usage("must specify exactly two of -l, -m, and -p")

834     if len(togglemodechecks) > 0 and len(modechecks) == 0:
835         usage("-X requires -M")

837     for s in togglemodechecks:
838         if s not in ALLMODECHECKS:
839             usage("unknown mode check %s" % s)
840         modechecks.symmetric_difference_update((s))

842     if len(modechecks) == 0:
843         modechecks = None

845     if not arch:
846         usage("must specify architecture")

848     exceptions = ExceptionList(exceptionlists, arch)
849     originalexceptions = exceptions.copy()
```



```
851     if len(manifestdirs) > 0:
852         for mdir in manifestdirs:
853             manifesttree.adddir(mdir, arch, modechecks, exceptions)
854         if listonly:
855             manifesttree.resolvehardlinks()
856             for info in manifesttree.values():
857                 print "%s" % info.protostr()
858             sys.exit(0)
859         if modechecks is not None:
860             sys.exit(0)
861         trees.append(manifesttree)

863     if len(protodirs) > 0:
864         for pdir in protodirs:
865             prototree.adddir(pdir, exceptions)
866         trees.append(prototree)

868     if len(protolists) > 0:
869         for plist in protolists:
870             try:
871                 protolist.addprotolist(plist, exceptions)
872             except IOError, exc:
873                 sys.stderr.write("warning: %s\n" % str(exc))
874         trees.append(protolist)

876     if verbose and exceptions:
877         print "Entries present in exception list but missing from proto area:"
878         for exc in sorted(exceptions):
879             print "\t%s" % exc
880         print ""

882     usedexceptions = originalexceptions.difference(exceptions)
883     harmfulexceptions = usedexceptions.intersection(manifesttree)
884     if harmfulexceptions:
885         print "Entries present in exception list but also in manifests:"
886         for exc in sorted(harmfulexceptions):
887             print "\t%s" % exc
888             del manifesttree[exc]
889         print ""

891     trees[0].compare(trees[1])

893 if __name__ == '__main__':
894     try:
895         main(sys.argv[1:])
896     except KeyboardInterrupt:
897         sys.exit(1)
898     except IOError:
899         sys.exit(1)
```