paramiko / **paramiko** Public

Code | Issues 734 | Pull requests 219 | Actions | Wiki | Security | ...

363a28d94c

paramiko / paramiko / pkey.py / <> Jump to

bitprophet Add support for RSA SHA2 host and public keys ... ✓ History

13 contributors +1

746 lines (651 sloc) | 27.6 KB

```python
# Copyright (C) 2003-2007  Robey Pointer <robeypointer@gmail.com>
#
# This file is part of paramiko.
#
# Paramiko is free software; you can redistribute it and/or modify it under the
# terms of the GNU Lesser General Public License as published by the Free
# Software Foundation; either version 2.1 of the License, or (at your option)
# any later version.
#
# Paramiko is distributed in the hope that it will be useful, but WITHOUT ANY
# WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR
# A PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
# details.
#
# You should have received a copy of the GNU Lesser General Public License
# along with Paramiko; if not, write to the Free Software Foundation, Inc.,
# 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA.

"""
Common API for all public keys.
"""

import base64
from binascii import unhexlify
import os
from hashlib import md5
import re
import struct

import six
import bcrypt

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers import algorithms, modes, Cipher

from paramiko import util
from paramiko.common import o600
from paramiko.py3compat import u, b, encodebytes, decodebytes, string_types
from paramiko.ssh_exception import SSHException, PasswordRequiredException
from paramiko.message import Message


OPENSSH_AUTH_MAGIC = b"openssh-key-v1\x00"


def _unpad_openssh(data):
    # At the moment, this is only used for unpadding private keys on disk. This
    # really ought to be made constant time (possibly by upstreaming this logic
    # into pyca/cryptography).
    padding_length = six.indexbytes(data, -1)
    if 0x20 <= padding_length < 0x7f:
        return data  # no padding, last byte part comment (printable ascii)
    if padding_length > 15:
        raise SSHException("Invalid key")
    for i in range(padding_length):
        if six.indexbytes(data, i - padding_length) != i + 1:
            raise SSHException("Invalid key")
    return data[:-padding_length]


class PKey(object):
    """
    Base class for public keys.
    """

    # known encryption types for private key files:
    _CIPHER_TABLE = {
        "AES-128-CBC": {
            "cipher": algorithms.AES,
            "keysize": 16,
            "blocksize": 16,
            "mode": modes.CBC,
        },
        "AES-256-CBC": {
            "cipher": algorithms.AES,
            "keysize": 32,
            "blocksize": 16,
            "mode": modes.CBC,
        },
        "DES-EDE3-CBC": {
            "cipher": algorithms.TripleDES,
            "keysize": 24,
            "blocksize": 8,
            "mode": modes.CBC,
        },
    }
    _PRIVATE_KEY_FORMAT_ORIGINAL = 1
    _PRIVATE_KEY_FORMAT_OPENSSH = 2
    BEGIN_TAG = re.compile(
        r"^-{5}BEGIN (RSA|DSA|EC|OPENSSH) PRIVATE KEY-{5}\s*$"
    )
    END_TAG = re.compile(r"^-{5}END (RSA|DSA|EC|OPENSSH) PRIVATE KEY-{5}\s*$")

    def __init__(self, msg=None, data=None):
        """
        Create a new instance of this public key type.  If ``msg`` is given,
        the key's public part(s) will be filled in from the message.  If
        ``data`` is given, the key's public part(s) will be filled in from
        the string.

        :param .Message msg:
            an optional SSH `.Message` containing a public key of this type.
        :param str data: an optional string containing a public key
            of this type

        :raises: `.SSHException` --
            if a key cannot be created from the ``data`` or ``msg`` given, or
            no key was passed in.
        """
        pass

    def asbytes(self):
        """
        Return a string of an SSH `.Message` made up of the public part(s) of
        this key.  This string is suitable for passing to `__init__` to
        re-create the key object later.
        """
        return bytes()

    def __str__(self):
        return self.asbytes()

    # noinspection PyUnresolvedReferences
    # TODO: The comparison functions should be removed as per:
    # https://docs.python.org/3.0/whatsnew/3.0.html#ordering-comparisons
    def __cmp__(self, other):
```

```python
        """
        Compare this key to another.  Returns 0 if this key is equivalent to
        the given key, or non-0 if they are different.  Only the public parts
        of the key are compared, so a public key will compare equal to its
        corresponding private key.

        :param .PKey other: key to compare to.
        """
        hs = hash(self)
        ho = hash(other)
        if hs != ho:
            return cmp(hs, ho)  # noqa
        return cmp(self.asbytes(), other.asbytes())  # noqa

    def __eq__(self, other):
        return self._fields == other._fields

    def __hash__(self):
        return hash(self._fields)

    @property
    def _fields(self):
        raise NotImplementedError

    def get_name(self):
        """
        Return the name of this private key implementation.

        :return:
            name of this private key type, in SSH terminology, as a `str` (for
            example, ``"ssh-rsa"``).
        """
        return ""

    def get_bits(self):
        """
        Return the number of significant bits in this key.  This is useful
        for judging the relative security of a key.

        :return: bits in the key (as an `int`)
        """
        return 0

    def can_sign(self):
        """
        Return ``True`` if this key has the private part necessary for signing
        data.
        """
        return False

    def get_fingerprint(self):
        """
        Return an MD5 fingerprint of the public part of this key.  Nothing
        secret is revealed.

        :return:
            a 16-byte `string <str>` (binary) of the MD5 fingerprint, in SSH
            format.
        """
        return md5(self.asbytes()).digest()

    def get_base64(self):
        """
        Return a base64 string containing the public part of this key.  Nothing
        secret is revealed.  This format is compatible with that used to store
        public key files or recognized host keys.

        :return: a base64 `string <str>` containing the public part of the key.
        """
        return u(encodebytes(self.asbytes())).replace("\n", "")

    def sign_ssh_data(self, data, algorithm=None):
        """
        Sign a blob of data with this private key, and return a `.Message`
        representing an SSH signature message.

        :param str data:
            the data to sign.
        :param str algorithm:
            the signature algorithm to use, if different from the key's
            internal name. Default: ``None``.
        :return: an SSH signature `message <.Message>`.

        .. versionchanged:: 2.9
            Added the ``algorithm`` kwarg.
        """
        return bytes()

    def verify_ssh_sig(self, data, msg):
        """
        Given a blob of data, and an SSH message representing a signature of
        that data, verify that it was signed with this key.

        :param str data: the data that was signed.
        :param .Message msg: an SSH signature message
        :return:
            ``True`` if the signature verifies correctly; ``False`` otherwise.
        """
```

```python
            return False

    @classmethod
    def from_private_key_file(cls, filename, password=None):
        """
        Create a key object by reading a private key file.  If the private
        key is encrypted and ``password`` is not ``None``, the given password
        will be used to decrypt the key (otherwise `.PasswordRequiredException`
        is thrown).  Through the magic of Python, this factory method will
        exist in all subclasses of PKey (such as `.RSAKey` or `.DSSKey`), but
        is useless on the abstract PKey class.

        :param str filename: name of the file to read
        :param str password:
            an optional password to use to decrypt the key file, if it's
            encrypted
        :return: a new `.PKey` based on the given private key

        :raises: ``IOError`` -- if there was an error reading the file
        :raises: `.PasswordRequiredException` -- if the private key file is
            encrypted, and ``password`` is ``None``
        :raises: `.SSHException` -- if the key file is invalid
        """
        key = cls(filename=filename, password=password)
        return key

    @classmethod
    def from_private_key(cls, file_obj, password=None):
        """
        Create a key object by reading a private key from a file (or file-like)
        object.  If the private key is encrypted and ``password`` is not
        ``None``, the given password will be used to decrypt the key (otherwise
        `.PasswordRequiredException` is thrown).

        :param file_obj: the file-like object to read from
        :param str password:
            an optional password to use to decrypt the key, if it's encrypted
        :return: a new `.PKey` based on the given private key

        :raises: ``IOError`` -- if there was an error reading the key
        :raises: `.PasswordRequiredException` --
            if the private key file is encrypted, and ``password`` is ``None``
        :raises: `.SSHException` -- if the key file is invalid
        """
        key = cls(file_obj=file_obj, password=password)
        return key

    def write_private_key_file(self, filename, password=None):
        """
        Write private key contents into a file.  If the password is not
        ``None``, the key is encrypted before writing.

        :param str filename: name of the file to write
        :param str password:
            an optional password to use to encrypt the key file

        :raises: ``IOError`` -- if there was an error writing the file
        :raises: `.SSHException` -- if the key is invalid
        """
        raise Exception("Not implemented in PKey")

    def write_private_key(self, file_obj, password=None):
        """
        Write private key contents into a file (or file-like) object.  If the
        password is not ``None``, the key is encrypted before writing.

        :param file_obj: the file-like object to write into
        :param str password: an optional password to use to encrypt the key

        :raises: ``IOError`` -- if there was an error writing to the file
        :raises: `.SSHException` -- if the key is invalid
        """
        raise Exception("Not implemented in PKey")

    def _read_private_key_file(self, tag, filename, password=None):
        """
        Read an SSH2-format private key file, looking for a string of the type
        ``"BEGIN xxx PRIVATE KEY"`` for some ``xxx``, base64-decode the text we
        find, and return it as a string.  If the private key is encrypted and
        ``password`` is not ``None``, the given password will be used to
        decrypt the key (otherwise `.PasswordRequiredException` is thrown).

        :param str tag: ``"RSA"`` or ``"DSA"``, the tag used to mark the
            data block.
        :param str filename: name of the file to read.
        :param str password:
            an optional password to use to decrypt the key file, if it's
            encrypted.
        :return: data blob (`str`) that makes up the private key.

        :raises: ``IOError`` -- if there was an error reading the file.
        :raises: `.PasswordRequiredException` -- if the private key file is
            encrypted, and ``password`` is ``None``.
        :raises: `.SSHException` -- if the key file is invalid.
        """
        with open(filename, "r") as f:
            data = self._read_private_key(tag, f, password)
        return data
```

```python

    def _read_private_key(self, tag, f, password=None):
        lines = f.readlines()

        # find the BEGIN tag
        start = 0
        m = self.BEGIN_TAG.match(lines[start])
        line_range = len(lines) - 1
        while start < line_range and not m:
            start += 1
            m = self.BEGIN_TAG.match(lines[start])
        start += 1
        keytype = m.group(1) if m else None
        if start >= len(lines) or keytype is None:
            raise SSHException("not a valid {} private key file".format(tag))

        # find the END tag
        end = start
        m = self.END_TAG.match(lines[end])
        while end < line_range and not m:
            end += 1
            m = self.END_TAG.match(lines[end])

        if keytype == tag:
            data = self._read_private_key_pem(lines, end, password)
            pkformat = self._PRIVATE_KEY_FORMAT_ORIGINAL
        elif keytype == "OPENSSH":
            data = self._read_private_key_openssh(lines[start:end], password)
            pkformat = self._PRIVATE_KEY_FORMAT_OPENSSH
        else:
            raise SSHException(
                "encountered {} key, expected {} key".format(keytype, tag)
            )

        return pkformat, data

    def _got_bad_key_format_id(self, id_):
        err = "{}._read_private_key() spat out an unknown key format id '{}'"
        raise SSHException(err.format(self.__class__.__name__, id_))

    def _read_private_key_pem(self, lines, end, password):
        start = 0
        # parse any headers first
        headers = {}
        start += 1
        while start < len(lines):
            line = lines[start].split(": ")
            if len(line) == 1:
                break
            headers[line[0].lower()] = line[1].strip()
            start += 1
        # if we trudged to the end of the file, just try to cope.
        try:
            data = decodebytes(b("".join(lines[start:end])))
        except base64.binascii.Error as e:
            raise SSHException("base64 decoding error: {}".format(e))
        if "proc-type" not in headers:
            # unencryped: done
            return data
        # encrypted keyfile: will need a password
        proc_type = headers["proc-type"]
        if proc_type != "4,ENCRYPTED":
            raise SSHException(
                'Unknown private key structure "{}"'.format(proc_type)
            )
        try:
            encryption_type, saltstr = headers["dek-info"].split(",")
        except:
            raise SSHException("Can't parse DEK-info in private key file")
        if encryption_type not in self._CIPHER_TABLE:
            raise SSHException(
                'Unknown private key cipher "{}"'.format(encryption_type)
            )
        # if no password was passed in,
        # raise an exception pointing out that we need one
        if password is None:
            raise PasswordRequiredException("Private key file is encrypted")
        cipher = self._CIPHER_TABLE[encryption_type]["cipher"]
        keysize = self._CIPHER_TABLE[encryption_type]["keysize"]
        mode = self._CIPHER_TABLE[encryption_type]["mode"]
        salt = unhexlify(b(saltstr))
        key = util.generate_key_bytes(md5, salt, password, keysize)
        decryptor = Cipher(
            cipher(key), mode(salt), backend=default_backend()
        ).decryptor()
        return decryptor.update(data) + decryptor.finalize()

    def _read_private_key_openssh(self, lines, password):
        """
        Read the new OpenSSH SSH2 private key format available
        since OpenSSH version 6.5
        Reference:
        https://github.com/openssh/openssh-portable/blob/master/PROTOCOL.key
        """
        try:
            data = decodebytes(b("".join(lines)))
        except base64.binascii.Error as e:
            raise SSHException("base64 decoding error: {}".format(e))
```

```python

        # read data struct
        auth_magic = data[:15]
        if auth_magic != OPENSSH_AUTH_MAGIC:
            raise SSHException("unexpected OpenSSH key header encountered")

        cstruct = self._uint32_cstruct_unpack(data[15:], "sssur")
        cipher, kdfname, kdf_options, num_pubkeys, remainder = cstruct
        # For now, just support 1 key.
        if num_pubkeys > 1:
            raise SSHException(
                "unsupported: private keyfile has multiple keys"
            )
        pubkey, privkey_blob = self._uint32_cstruct_unpack(remainder, "ss")

        if kdfname == b("bcrypt"):
            if cipher == b("aes256-cbc"):
                mode = modes.CBC
            elif cipher == b("aes256-ctr"):
                mode = modes.CTR
            else:
                raise SSHException(
                    "unknown cipher `{}` used in private key file".format(
                        cipher.decode("utf-8")
                    )
                )
            # Encrypted private key.
            # If no password was passed in, raise an exception pointing
            # out that we need one
            if password is None:
                raise PasswordRequiredException(
                    "private key file is encrypted"
                )

            # Unpack salt and rounds from kdfoptions
            salt, rounds = self._uint32_cstruct_unpack(kdf_options, "su")

            # run bcrypt kdf to derive key and iv/nonce (32 + 16 bytes)
            key_iv = bcrypt.kdf(
                b(password),
                b(salt),
                48,
                rounds,
                # We can't control how many rounds are on disk, so no sense
                # warning about it.
                ignore_few_rounds=True,
            )
            key = key_iv[:32]
            iv = key_iv[32:]

            # decrypt private key blob
            decryptor = Cipher(
                algorithms.AES(key), mode(iv), default_backend()
            ).decryptor()
            decrypted_privkey = decryptor.update(privkey_blob)
            decrypted_privkey += decryptor.finalize()
        elif cipher == b("none") and kdfname == b("none"):
            # Unencrypted private key
            decrypted_privkey = privkey_blob
        else:
            raise SSHException(
                "unknown cipher or kdf used in private key file"
            )

        # Unpack private key and verify checkints
        cstruct = self._uint32_cstruct_unpack(decrypted_privkey, "uusr")
        checkint1, checkint2, keytype, keydata = cstruct

        if checkint1 != checkint2:
            raise SSHException(
                "OpenSSH private key file checkints do not match"
            )

        return _unpad_openssh(keydata)

    def _uint32_cstruct_unpack(self, data, strformat):
        """
        Used to read new OpenSSH private key format.
        Unpacks a c data structure containing a mix of 32-bit uints and
        variable length strings prefixed by 32-bit uint size field,
        according to the specified format. Returns the unpacked vars
        in a tuple.
        Format strings:
          s - denotes a string
          i - denotes a long integer, encoded as a byte string
          u - denotes a 32-bit unsigned integer
          r - the remainder of the input string, returned as a string
        """
        arr = []
        idx = 0
        try:
            for f in strformat:
                if f == "s":
                    # string
                    s_size = struct.unpack(">L", data[idx : idx + 4])[0]
                    idx += 4
                    s = data[idx : idx + s_size]
                    idx += s_size
                    arr.append(s)
                if f == "i":
                    # long integer
                    s_size = struct.unpack(">L", data[idx : idx + 4])[0]
                    idx += 4
                    s = data[idx : idx + s_size]
                    idx += s_size
                    i = util.inflate_long(s, True)
                    arr.append(i)
                elif f == "u":
                    # 32-bit unsigned int
                    u = struct.unpack(">L", data[idx : idx + 4])[0]
                    idx += 4
                    arr.append(u)
                elif f == "r":
                    # remainder as string
                    s = data[idx:]
                    arr.append(s)
                    break
        except Exception as e:
            # PKey-consuming code frequently wants to save-and-skip-over issues
            # with loading keys, and uses SSHException as the (really friggin
            # awful) signal for this. So for now...we do this.
            raise SSHException(str(e))
        return tuple(arr)

    def _write_private_key_file(self, filename, key, format, password=None):
        """
        Write an SSH2-format private key file in a form that can be read by
        paramiko or openssh.  If no password is given, the key is written in
        a trivially-encoded format (base64) which is completely insecure.  If
        a password is given, DES-EDE3-CBC is used.

        :param str tag:
            ``"RSA"`` or ``"DSA"``, the tag used to mark the data block.
        :param filename: name of the file to write.
        :param str data: data blob that makes up the private key.
        :param str password: an optional password to use to encrypt the file.

        :raises: ``IOError`` -- if there was an error writing the file.
        """
        with open(filename, "w") as f:
            os.chmod(filename, o600)
            self._write_private_key(f, key, format, password=password)

    def _write_private_key(self, f, key, format, password=None):
        if password is None:
            encryption = serialization.NoEncryption()
        else:
            encryption = serialization.BestAvailableEncryption(b(password))

        f.write(
            key.private_bytes(
                serialization.Encoding.PEM, format, encryption
            ).decode()
        )

    def _check_type_and_load_cert(self, msg, key_type, cert_type):
        """
        Perform message type-checking & optional certificate loading.

        This includes fast-forwarding cert ``msg`` objects past the nonce, so
        that the subsequent fields are the key numbers; thus the caller may
        expect to treat the message as key material afterwards either way.

        The obtained key type is returned for classes which need to know what
        it was (e.g. ECDSA.)
        """
        # Normalization; most classes have a single key type and give a string,
        # but eg ECDSA is a 1:N mapping.
        key_types = key_type
        cert_types = cert_type
        if isinstance(key_type, string_types):
            key_types = [key_types]
        if isinstance(cert_types, string_types):
            cert_types = [cert_types]
        # Can't do much with no message, that should've been handled elsewhere
        if msg is None:
            raise SSHException("Key object may not be empty")
        # First field is always key type, in either kind of object. (make sure
        # we rewind before grabbing it - sometimes caller had to do their own
        # introspection first!)
        msg.rewind()
        type_ = msg.get_text()
        # Regular public key - nothing special to do besides the implicit
        # type check.
        if type_ in key_types:
            pass
        # OpenSSH-compatible certificate - store full copy as .public_blob
        # (so signing works correctly) and then fast-forward past the
        # nonce.
        elif type_ in cert_types:
            # This seems the cleanest way to 'clone' an already-being-read
            # message; they're *IO objects at heart and their .getvalue()
            # always returns the full value regardless of pointer position.
            self.load_certificate(Message(msg.asbytes()))
            # Read out nonce as it comes before the public numbers.
            # TODO: usefully interpret it & other non-public-number fields
            # (requires going back into per-type subclasses.)
            msg.get_string()
        else:
            err = "Invalid key (class: {}, data type: {}"
            raise SSHException(err.format(self.__class__.__name__, type_))

    def load_certificate(self, value):
        """
        Supplement the private key contents with data loaded from an OpenSSH
        public key (``.pub``) or certificate (``-cert.pub``) file, a string
        containing such a file, or a `.Message` object.

        The .pub contents adds no real value, since the private key
        file includes sufficient information to derive the public
        key info. For certificates, however, this can be used on
        the client side to offer authentication requests to the server
        based on certificate instead of raw public key.

        See:
        https://github.com/openssh/openssh-portable/blob/master/PROTOCOL.certkeys

        Note: very little effort is made to validate the certificate contents,
        that is for the server to decide if it is good enough to authenticate
        successfully.
        """
        if isinstance(value, Message):
            constructor = "from_message"
        elif os.path.isfile(value):
            constructor = "from_file"
        else:
            constructor = "from_string"
        blob = getattr(PublicBlob, constructor)(value)
        if not blob.key_type.startswith(self.get_name()):
            err = "PublicBlob type {} incompatible with key type {}"
            raise ValueError(err.format(blob.key_type, self.get_name()))
        self.public_blob = blob


# General construct for an OpenSSH style Public Key blob
# readable from a one-line file of the format:
#     <key-name> <base64-blob> [<comment>]
# Of little value in the case of standard public keys
# {ssh-rsa, ssh-dss, ssh-ecdsa, ssh-ed25519}, but should
# provide rudimentary support for {*-cert.v01}
class PublicBlob(object):
    """
    OpenSSH plain public key or OpenSSH signed public key (certificate).

    Tries to be as dumb as possible and barely cares about specific
    per-key-type data.

    .. note::

        Most of the time you'll want to call `from_file`, `from_string` or
        `from_message` for useful instantiation, the main constructor is
        basically "I should be using ``attrs`` for this."
    """

    def __init__(self, type_, blob, comment=None):
        """
        Create a new public blob of given type and contents.

        :param str type_: Type indicator, eg ``ssh-rsa``.
        :param blob: The blob bytes themselves.
        :param str comment: A comment, if one was given (e.g. file-based.)
        """
        self.key_type = type_
        self.key_blob = blob
        self.comment = comment

    @classmethod
    def from_file(cls, filename):
        """
        Create a public blob from a ``-cert.pub``-style file on disk.
        """
        with open(filename) as f:
            string = f.read()
        return cls.from_string(string)

    @classmethod
    def from_string(cls, string):
        """
        Create a public blob from a ``-cert.pub``-style string.
        """
        fields = string.split(None, 2)
        if len(fields) < 2:
            msg = "Not enough fields for public blob: {}"
            raise ValueError(msg.format(fields))
        key_type = fields[0]
        key_blob = decodebytes(b(fields[1]))
        try:
            comment = fields[2].strip()
        except IndexError:
            comment = None
        # Verify that the blob message first (string) field matches the
        # key_type
        m = Message(key_blob)
        blob_type = m.get_text()
```

```python
        if blob_type != key_type:
            deets = "key type={!r}, but blob type={!r}".format(
                key_type, blob_type
            )
            raise ValueError("Invalid PublicBlob contents: {}".format(deets))
        # All good? All good.
        return cls(type_=key_type, blob=key_blob, comment=comment)

    @classmethod
    def from_message(cls, message):
        """
        Create a public blob from a network `.Message`.

        Specifically, a cert-bearing pubkey auth packet, because by definition
        OpenSSH-style certificates 'are' their own network representation."
        """
        type_ = message.get_text()
        return cls(type_=type_, blob=message.asbytes())

    def __str__(self):
        ret = "{} public key/certificate".format(self.key_type)
        if self.comment:
            ret += "- {}".format(self.comment)
        return ret

    def __eq__(self, other):
        # Just piggyback on Message/BytesIO, since both of these should be one.
        return self and other and self.key_blob == other.key_blob

    def __ne__(self, other):
        return not self == other
```