aio-adb-shell Documentation

Release 0.0.1

Jeff Irion

May 07, 2020

CONTENTS

CHAPTER

ONE

AIO_ADB_SHELL

1.1 aio_adb_shell package

1.1.1 Subpackages

aio_adb_shell.auth package

Submodules

aio_adb_shell.auth.keygen module

This file implements encoding and decoding logic for Android's custom RSA public key binary format. Public keys are stored as a sequence of little-endian 32 bit words. Note that Android only supports little-endian processors, so we don't do any byte order conversions when parsing the binary struct.

Structure from: https://github.com/aosp-mirror/platform_system_core/blob/c55fab4a59cfa461857c6a61d8a0f1ae4591900c/libcrypto_utils/android_pubkey.c

```
typedef struct RSAPublicKey {
    // Modulus length. This must be ANDROID_PUBKEY_MODULUS_SIZE_WORDS
    uint32_t modulus_size_words;

    // Precomputed montgomery parameter: -1 / n[0] mod 2^32
    uint32_t n0inv;

    // RSA modulus as a little-endian array
    uint8_t modulus[ANDROID_PUBKEY_MODULUS_SIZE];

    // Montgomery parameter R^2 as a little-endian array of little-endian words
    uint8_t rr[ANDROID_PUBKEY_MODULUS_SIZE];

    // RSA modulus: 3 or 65537
    uint32_t exponent;
} RSAPublicKey;
```

Contents

- *encode_pubkey()*
- *get_user_info()*
- *keygen()*
- *write_public_keyfile()*

aio_adb_shell.auth.keygen.**ANDROID_PUBKEY_MODULUS_SIZE = 256**
Size of an RSA modulus such as an encrypted block or a signature.

aio_adb_shell.auth.keygen.**ANDROID_PUBKEY_MODULUS_SIZE_WORDS = 64**
Size of the RSA modulus in words.

aio_adb_shell.auth.keygen.**ANDROID_RSAPUBLICKEY_STRUCT = '<LL256s256sL'**
Python representation of "struct RSAPublicKey"

aio_adb_shell.auth.keygen.**_to_bytes**(*n*, *length*, *endianess='big'*)
Partial python2 compatibility with int.to_bytes

https://stackoverflow.com/a/20793663

Parameters

- **n** (*TODO*) – TODO
- **length** (*TODO*) – TODO
- **endianess** (*str, TODO*) – TODO

Returns TODO

Return type TODO

aio_adb_shell.auth.keygen.**decode_pubkey**(*public_key*)
Decode a public RSA key stored in Android's custom binary format.

Parameters **public_key** (*TODO*) – TODO

aio_adb_shell.auth.keygen.**decode_pubkey_file**(*public_key_path*)
TODO

Parameters **public_key_path** (*str*) – TODO

aio_adb_shell.auth.keygen.**encode_pubkey**(*private_key_path*)
Encodes a public RSA key into Android's custom binary format.

Parameters **private_key_path** (*str*) – TODO

Returns TODO

Return type TODO

aio_adb_shell.auth.keygen.**get_user_info**()
TODO

Returns `' <username>@<hostname>`

Return type str

aio_adb_shell.auth.keygen.**keygen**(*filepath*)
Generate an ADB public/private key pair.

- The private key is stored in `filepath`.
- The public key is stored in `filepath + '.pub'`

(Existing files will be overwritten.)

Parameters **filepath** (*str*) – File path to write the private/public keypair

aio_adb_shell.auth.keygen.**write_public_keyfile**(*private_key_path*, *public_key_path*)
Write a public keyfile to public_key_path in Android's custom RSA public key format given a path to a private keyfile.

Parameters

- **private_key_path** (*TODO*) – TODO
- **public_key_path** (*TODO*) – TODO

aio_adb_shell.auth.sign_cryptography module

ADB authentication using the cryptography package.

Contents

class aio_adb_shell.auth.sign_cryptography.**CryptographySigner**(*rsa_key_path*)
Bases: object

AuthSigner using cryptography.io.

Parameters **rsa_key_path** (*str*) – The path to the private key.

public_key
The contents of the public key file

Type str

rsa_key
The loaded private key

Type cryptography.hazmat.backends.openssl.rsa._RSAPrivateKey

GetPublicKey()
Returns the public key in PEM format without headers or newlines.

Returns **self.public_key** – The contents of the public key file

Return type str

Sign(*data*)
Signs given data using a private key.

Parameters **data** (*TODO*) – TODO

Returns The signed data

Return type TODO

aio_adb_shell.auth.sign_pycryptodome module

ADB authentication using pycryptodome.

Contents

class aio_adb_shell.auth.sign_pycryptodome.**PycryptodomeAuthSigner**(*rsa_key_path=None*)
Bases: object

AuthSigner using the pycryptodome package.

Parameters **rsa_key_path** (*str, None*) – The path to the private key

public_key
The contents of the public key file

Type str

rsa_key
The contents of theprivate key

Type Crypto.PublicKey.RSA.RsaKey

GetPublicKey()
Returns the public key in PEM format without headers or newlines.

Returns **self.public_key** – The contents of the public key file

Return type str

Sign(*data*)
Signs given data using a private key.

Parameters **data** (*bytes, bytearray*) – The data to be signed

Returns The signed data

Return type bytes

aio_adb_shell.auth.sign_pythonrsa module

ADB authentication using the rsa package.

Contents

class aio_adb_shell.auth.sign_pythonrsa.**PythonRSASigner**(*pub=None*, *priv=None*)
Bases: object

Implements adb_protocol.AuthSigner using http://stuvel.eu/rsa.

Parameters

- **pub** (*str, None*) – The contents of the public key file
- **priv** (*str, None*) – The path to the private key

priv_key
The loaded private key

Type rsa.key.PrivateKey

pub_key
The contents of the public key file

Type str, None

classmethod FromRSAKeyPath(*rsa_key_path*)
Create a *PythonRSASigner* instance using the provided private key.

Parameters **rsa_key_path** (*str*) – The path to the private key; the public key must be rsa_key_path + '.pub'.

Returns A *PythonRSASigner* with private key rsa_key_path and public key rsa_key_path + '.pub'

Return type *PythonRSASigner*

GetPublicKey()
Returns the public key in PEM format without headers or newlines.

Returns **self.pub_key** – The contents of the public key file, or None if a public key was not provided.

Return type str, None

Sign(*data*)
Signs given data using a private key.

Parameters **data** (*bytes*) – The data to be signed

Returns The signed data

Return type bytes

class aio_adb_shell.auth.sign_pythonrsa.**_Accum**
Bases: object

A fake hashing algorithm.

The Python rsa lib hashes all messages it signs. ADB does it already, we just need to slap a signature on top of already hashed message. Introduce a "fake" hashing algo for this.

_buf
A buffer for storing data before it is signed

Type bytes

digest()
Return the digest value as a string of binary data.

Returns **self._buf** – self._buf

Return type bytes

update(*msg*)
Update this hash object's state with the provided `msg`.

Parameters **msg** (*bytes*) – The message to be appended to `self._buf`

`aio_adb_shell.auth.sign_pythonrsa.`**_load_rsa_private_key**(*pem*)
PEM encoded PKCS#8 private key -> `rsa.PrivateKey`.

ADB uses private RSA keys in pkcs#8 format. The `rsa` library doesn't support them natively. Do some ASN unwrapping to extract naked RSA key (in der-encoded form).

See:

- https://www.ietf.org/rfc/rfc2313.txt
- http://superuser.com/a/606266

Parameters **pem** (*str*) – The private key to be loaded

Returns The loaded private key

Return type rsa.key.PrivateKey

Module contents

aio_adb_shell.handle package

Submodules

aio_adb_shell.handle.base_handle module

A base class for handles used to communicate with a device.

class aio_adb_shell.handle.base_handle.**BaseHandle**
Bases: `abc.ABC`

A base handle class.

_abc_impl = <_abc_data object>

abstract async bulk_read(*numbytes*, *timeout_s=None*)
Read data from the device.

Parameters

- **numbytes** (*int*) – The maximum amount of data to be received
- **timeout_s** (*float, None*) – A timeout for the read operation

Returns The received data

Return type bytes

abstract async bulk_write(*data*, *timeout_s=None*)
Send data to the device.

Parameters

- **data** (*bytes*) – The data to be sent
- **timeout_s** (*float, None*) – A timeout for the write operation

Returns The number of bytes sent

Return type int

abstract async close()
Close the connection.

abstract async connect(*timeout_s=None*)
Create a connection to the device.

Parameters **timeout_s** (*float, None*) – A connection timeout

aio_adb_shell.handle.tcp_handle module

A class for creating a socket connection with the device and sending and receiving data.

class aio_adb_shell.handle.tcp_handle.**TcpHandle**(*host*, *port=5555*, *default_timeout_s=None*)
Bases: *aio_adb_shell.handle.base_handle.BaseHandle*

TCP connection object.

Parameters

- **host** (*str*) – The address of the device; may be an IP address or a host name
- **port** (*int*) – The device port to which we are connecting (default is 5555)
- **default_timeout_s** (*float, None*) – Default timeout in seconds for TCP packets, or None

_default_timeout_s
Default timeout in seconds for TCP packets, or None

Type float, None

_host
The address of the device; may be an IP address or a host name

Type str

_port
The device port to which we are connecting (default is 5555)

Type int

_reader
TODO

Type StreamReader, None

_writer
TODO

Type StreamWriter, None

_abc_impl = <_abc_data object>

async bulk_read(*numbytes*, *timeout_s=None*)
Receive data from the socket.

Parameters

- **numbytes** (*int*) – The maximum amount of data to be received
- **timeout_s** (*float, None*) – When the timeout argument is omitted, `select.select` blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

Returns The received data

Return type bytes

Raises ***TcpTimeoutException*** – Reading timed out.

async bulk_write(*data*, *timeout_s=None*)
Send data to the socket.

Parameters

- **data** (*bytes*) – The data to be sent
- **timeout_s** (*float, None*) – When the timeout argument is omitted, `select.select` blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

Returns The number of bytes sent

Return type int

Raises ***TcpTimeoutException*** – Sending data timed out. No data was sent.

async close()
Close the socket connection.

async connect(*timeout_s=None*)
Create a socket connection to the device.

Parameters **timeout_s** (*float, None*) – Set the timeout on the socket instance

Module contents

1.1.2 Submodules

aio_adb_shell.adb_device module

Implement the *AdbDevice* class, which can connect to a device and run ADB shell commands.

Contents

class aio_adb_shell.adb_device.**AdbDevice**(*handle*, *banner=None*)
Bases: object

A class with methods for connecting to a device and executing ADB commands.

Parameters

- **handle** (BaseHandle) – A user-provided handle for communicating with the device; must be an instance of a subclass of BaseHandle
- **banner** (*str, bytes, None*) – The hostname of the machine where the Python interpreter is currently running; if it is not provided, it will be determined via socket.gethostname()

Raises **adb_shell.exceptions.InvalidHandleError** – The passed handle is not an instance of a subclass of BaseHandle

_available
Whether an ADB connection to the device has been established

Type bool

_banner
The hostname of the machine where the Python interpreter is currently running

Type bytearray, bytes

_handle
The handle that is used to connect to the device; must be a subclass of BaseHandle

Type *BaseHandle*

async _close(*adb_info*)
Send a b'CLSE' message.

> **Warning:** This is not to be confused with the *AdbDevice.close()* method!

Parameters **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

async _filesync_flush(*adb_info*, *filesync_info*)
Write the data in the buffer up to filesync_info.send_idx, then set filesync_info.send_idx to 0.

Parameters

- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

async _filesync_read(*expected_ids*, *adb_info*, *filesync_info*, *read_data=True*)
Read ADB messages and return FileSync packets.

Parameters

- **expected_ids** (*tuple[bytes]*) – If the received header ID is not in expected_ids, an exception will be raised
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction
- **read_data** (*bool*) – Whether to read the received data

Returns

- **command_id** (*bytes*) – The received header ID
- *tuple* – The contents of the header
- **data** (*bytearray, None*) – The received data, or `None` if `read_data` is False

Raises

- **adb_shell.exceptions.AdbCommandFailureException** – Command failed
- **adb_shell.exceptions.InvalidResponseError** – Received response was not in `expected_ids`

async _filesync_read_buffered(*size*, *adb_info*, *filesync_info*)

Read `size` bytes of data from `self.recv_buffer`.

Parameters

- **size** (*int*) – The amount of data to read
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Returns **result** – The read data

Return type bytearray

_filesync_read_until(*expected_ids*, *finish_ids*, *adb_info*, *filesync_info*)

Useful wrapper around *AdbDevice._filesync_read()*.

Parameters

- **expected_ids** (*tuple[bytes]*) – If the received header ID is not in `expected_ids`, an exception will be raised
- **finish_ids** (*tuple[bytes]*) – We will read until we find a header ID that is in `finish_ids`
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Yields

- **cmd_id** (*bytes*) – The received header ID
- **header** (*tuple*) – TODO
- **data** (*bytearray*) – The received data

async _filesync_send(*command_id*, *adb_info*, *filesync_info*, *data=b''*, *size=0*)

Send/buffer FileSync packets.

Packets are buffered and only flushed when this connection is read from. All messages have a response from the device, so this will always get flushed.

Parameters

- **command_id** (*bytes*) – Command to send.
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction
- **data** (*str, bytes*) – Optional data to send, must set data or size.
- **size** (*int*) – Optionally override size from len(data).

static _handle_progress(*progress_callback*)
Calls the callback with the current progress and total bytes written/received.

Parameters **progress_callback** (*function*) – Callback method that accepts filename, bytes_written, and total_bytes; total_bytes will be -1 for file-like objects.

async _okay(*adb_info*)
Send an b'OKAY' mesage.

Parameters **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

async _open(*destination*, *adb_info*)
Opens a new connection to the device via an b'OPEN' message.

1. *_send()* an b'OPEN' command to the device that specifies the local_id
2. *_read()* a response from the device that includes a command, another local ID (their_local_id), and remote_id
   - If local_id and their_local_id do not match, raise an exception.
   - If the received command is b'CLSE', *_read()* another response from the device
   - If the received command is not b'OKAY', raise an exception
   - Set the adb_info.local_id and adb_info.remote_id attributes

Parameters

- **destination** (*bytes*) – b'SERVICE:COMMAND'
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

Raises **adb_shell.exceptions.InvalidResponseError** – Wrong local_id sent to us.

async _pull(*filename*, *dest*, *progress_callback*, *adb_info*, *filesync_info*)
Pull a file from the device into the file-like dest_file.

Parameters

- **filename** (*str*) – The file to be pulled
- **dest** (*_io.BytesIO*) – File-like object for writing to
- **progress_callback** (*function, None*) – Callback method that accepts filename, bytes_written, and total_bytes
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

async _push(*datafile*, *filename*, *st_mode*, *mtime*, *progress_callback*, *adb_info*, *filesync_info*)
Push a file-like object to the device.

Parameters

- **datafile** (*_io.BytesIO*) – File-like object for reading from
- **filename** (*str*) – Filename to push to
- **st_mode** (*int*) – Stat mode for filename
- **mtime** (*int*) – Modification time
- **progress_callback** (*function, None*) – Callback method that accepts `filename`, `bytes_written`, and `total_bytes`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Raises ***PushFailedError*** – Raised on push failure.

async _read(*expected_cmds*, *adb_info*)
Receive a response from the device.

1. Read a message from the device and unpack the `cmd`, `arg0`, `arg1`, `data_length`, and `data_checksum` fields
2. If `cmd` is not a recognized command in `adb_shell.constants.WIRE_TO_ID`, raise an exception
3. If the time has exceeded `total_timeout_s`, raise an exception
4. Read `data_length` bytes from the device
5. If the checksum of the read data does not match `data_checksum`, raise an exception
6. Return `command`, `arg0`, `arg1`, and `bytes(data)`

Parameters

- **expected_cmds** (*list[bytes]*) – We will read packets until we encounter one whose "command" field is in `expected_cmds`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Returns

- **command** (*bytes*) – The received command, which is in `adb_shell.constants.WIRE_TO_ID` and must be in `expected_cmds`
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- *bytes* – The data that was read

Raises

- **adb_shell.exceptions.InvalidCommandError** – Unknown command *or* never got one of the expected responses.
- **adb_shell.exceptions.InvalidChecksumError** – Received checksum does not match the expected checksum.

async _read_until(*expected_cmds*, *adb_info*)
Read a packet, acknowledging any write packets.

1. Read data via *AdbDevice._read()*

2. If a `b'WRTE'` packet is received, send an `b'OKAY'` packet via *AdbDevice._okay()*
3. Return the `cmd` and `data` that were read by *AdbDevice._read()*

Parameters

- **expected_cmds** (*list[bytes]*) – *AdbDevice._read()* with look for a packet whose command is in `expected_cmds`
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

Returns

- **cmd** (*bytes*) – The command that was received by *AdbDevice._read()*, which is in `adb_shell.constants.WIRE_TO_ID` and must be in `expected_cmds`
- **data** (*bytes*) – The data that was received by *AdbDevice._read()*

Raises

- **adb_shell.exceptions.InterleavedDataError** – We don't support multiple streams...
- **adb_shell.exceptions.InvalidResponseError** – Incorrect remote id.
- **adb_shell.exceptions.InvalidCommandError** – Never got one of the expected responses.

_read_until_close(*adb_info*)

Yield packets until a `b'CLSE'` packet is received.

1. Read the `cmd` and `data` fields from a `b'CLSE'` or `b'WRTE'` packet via *AdbDevice._read_until()*
2. If `cmd` is `b'CLSE'`, then send a `b'CLSE'` message and stop
3. If `cmd` is not `b'WRTE'`, raise an exception
   - If `cmd` is `b'FAIL'`, raise `AdbCommandFailureException`
   - Otherwise, raise `InvalidCommandError`
4. Yield `data` and repeat

Parameters **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

Yields **data** (*bytes*) – The data that was read by *AdbDevice._read_until()*

async _send(*msg*, *adb_info*)

Send a message to the device.

1. Send the message header (`adb_shell.adb_message.AdbMessage.pack`)
2. Send the message data

Parameters

- **msg** (AdbMessage) – The data that will be sent
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

async _service(*service*, *command*, *timeout_s=None*, *total_timeout_s=10.0*, *decode=True*)

Send an ADB command to the device.

Parameters

- **service** (*bytes*) – The ADB service to talk to (e.g., `b'shell'`)
- **command** (*bytes*) – The command that will be sent
- **timeout_s** (*float, None*) – Timeout in seconds for sending and receiving packets, or `None`; see `BaseHandle.bulk_read()` and `BaseHandle.bulk_write()`
- **total_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *AdbDevice._read()*
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns The output of the ADB command as a string if `decode` is True, otherwise as bytes.

Return type bytes, str

_streaming_command(*service*, *command*, *adb_info*)

One complete set of USB packets for a single command.

1. *_open()* a new connection to the device, where the `destination` parameter is `service:command`
2. Read the response data via *AdbDevice._read_until_close()*

Note: All the data is held in memory, and thus large responses will be slow and can fill up memory.

Parameters

- **service** (*bytes*) – The ADB service (e.g., `b'shell'`, as used by *AdbDevice.shell()*)
- **command** (*bytes*) – The service command
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Yields *bytes* – The responses from the service.

_streaming_service(*service*, *command*, *timeout_s=None*, *total_timeout_s=10.0*, *decode=True*)

Send an ADB command to the device, yielding each line of output.

Parameters

- **service** (*bytes*) – The ADB service to talk to (e.g., `b'shell'`)
- **command** (*bytes*) – The command that will be sent
- **timeout_s** (*float, None*) – Timeout in seconds for sending and receiving packets, or `None`; see `BaseHandle.bulk_read()` and `BaseHandle.bulk_write()`
- **total_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *AdbDevice._read()*
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Yields *bytes, str* – The line-by-line output of the ADB command as a string if `decode` is True, otherwise as bytes.

async _write(*data*, *adb_info*)

Write a packet and expect an Ack.

Parameters

- **data** (*bytes*) – The data that will be sent
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

property available
Whether or not an ADB connection to the device has been established.

Returns self._available

Return type bool

async close()
Close the connection via the provided handle's close() method.

async connect(*rsa_keys=None*, *timeout_s=None*, *auth_timeout_s=10.0*, *total_timeout_s=10.0*, *auth_callback=None*)
Establish an ADB connection to the device.

1. Use the handle to establish a connection
2. Send a b'CNXN' message
3. Unpack the cmd, arg0, arg1, and banner fields from the response
4. If cmd is not b'AUTH', then authentication is not necesary and so we are done
5. If no rsa_keys are provided, raise an exception
6. Loop through our keys, signing the last banner that we received
   1. If the last arg0 was not adb_shell.constants.AUTH_TOKEN, raise an exception
   2. Sign the last banner and send it in an b'AUTH' message
   3. Unpack the cmd, arg0, and banner fields from the response via adb_shell.adb_message.unpack()
   4. If cmd is b'CNXN', return banner
7. None of the keys worked, so send rsa_keys[0]'s public key; if the response does not time out, we must have connected successfully

Parameters

- **rsa_keys** (*list, None*) – A list of signers of type CryptographySigner, PycryptodomeAuthSigner, or PythonRSASigner
- **timeout_s** (*float, None*) – Timeout in seconds for sending and receiving packets, or None; see BaseHandle.bulk_read() and BaseHandle.bulk_write()
- **auth_timeout_s** (*float, None*) – The time in seconds to wait for a b'CNXN' authentication response
- **total_timeout_s** (*float*) – The total time in seconds to wait for expected commands in *AdbDevice._read()*
- **auth_callback** (*function, None*) – Function callback invoked when the connection needs to be accepted on the device

Returns Whether the connection was established (*AdbDevice.available*)

Return type bool

Raises

- **adb_shell.exceptions.DeviceAuthError** – Device authentication required, no keys available

- **adb_shell.exceptions.InvalidResponseError** – Invalid auth response from the device

async list(*device_path*, *timeout_s=None*, *total_timeout_s=10.0*)
Return a directory listing of the given path.

Parameters

- **device_path** (*str*) – Directory to list.
- **timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **total_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *AdbDevice._read()*

Returns files – Filename, mode, size, and mtime info for the files in the directory

Return type list[*DeviceFile*]

async pull(*device_filename*, *dest_file=None*, *progress_callback=None*, *timeout_s=None*, *total_timeout_s=10.0*)
Pull a file from the device.

Parameters

- **device_filename** (*str*) – Filename on the device to pull.
- **dest_file** (*str, file, io.IOBase, None*) – If set, a filename or writable file-like object.
- **progress_callback** (*function, None*) – Callback method that accepts filename, bytes_written and total_bytes, total_bytes will be -1 for file-like objects
- **timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **total_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *AdbDevice._read()*

Returns The file data if `dest_file` is not set. Otherwise, `True` if the destination file exists

Return type bytes, bool

Raises ValueError – If `dest_file` is of unknown type.

async push(*source_file*, *device_filename*, *st_mode=33272*, *mtime=0*, *progress_callback=None*, *timeout_s=None*, *total_timeout_s=10.0*)
Push a file or directory to the device.

Parameters

- **source_file** (*str*) – Either a filename, a directory or file-like object to push to the device.
- **device_filename** (*str*) – Destination on the device to write to.
- **st_mode** (*int*) – Stat mode for filename
- **mtime** (*int*) – Modification time to set on the file.
- **progress_callback** (*function, None*) – Callback method that accepts filename, bytes_written and total_bytes, total_bytes will be -1 for file-like objects
- **timeout_s** (*float, None*) – Expected timeout for any part of the push.
- **total_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *AdbDevice._read()*

async shell(*command*, *timeout_s=None*, *total_timeout_s=10.0*, *decode=True*)
Send an ADB shell command to the device.

Parameters

- **command** (*str*) – The shell command that will be sent
- **timeout_s** (*float, None*) – Timeout in seconds for sending and receiving packets, or None; see BaseHandle.bulk_read() and BaseHandle.bulk_write()
- **total_timeout_s** (*float*) – The total time in seconds to wait for a b'CLSE' or b'OKAY' command in *AdbDevice._read()*
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns The output of the ADB shell command as a string if decode is True, otherwise as bytes.

Return type bytes, str

async stat(*device_filename*, *timeout_s=None*, *total_timeout_s=10.0*)
Get a file's stat() information.

Parameters

- **device_filename** (*str*) – The file on the device for which we will get information.
- **timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **total_timeout_s** (*float*) – The total time in seconds to wait for a b'CLSE' or b'OKAY' command in *AdbDevice._read()*

Returns

- **mode** (*int*) – The octal permissions for the file
- **size** (*int*) – The size of the file
- **mtime** (*int*) – The last modified time for the file

streaming_shell(*command*, *timeout_s=None*, *total_timeout_s=10.0*, *decode=True*)
Send an ADB shell command to the device, yielding each line of output.

Parameters

- **command** (*str*) – The shell command that will be sent
- **timeout_s** (*float, None*) – Timeout in seconds for sending and receiving packets, or None; see BaseHandle.bulk_read() and BaseHandle.bulk_write()
- **total_timeout_s** (*float*) – The total time in seconds to wait for a b'CLSE' or b'OKAY' command in *AdbDevice._read()*
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Yields *bytes, str* – The line-by-line output of the ADB shell command as a string if decode is True, otherwise as bytes.

class aio_adb_shell.adb_device.**AdbDeviceTcp**(*host*, *port=5555*, *default_timeout_s=None*, *banner=None*)
Bases: *aio_adb_shell.adb_device.AdbDevice*

A class with methods for connecting to a device via TCP and executing ADB commands.

Parameters

- **host** (*str*) – The address of the device; may be an IP address or a host name

- **port** (*int*) – The device port to which we are connecting (default is 5555)
- **default_timeout_s** (*float, None*) – Default timeout in seconds for TCP packets, or None
- **banner** (*str, bytes, None*) – The hostname of the machine where the Python interpreter is currently running; if it is not provided, it will be determined via `socket.gethostname()`

_available
Whether an ADB connection to the device has been established

Type bool

_banner
The hostname of the machine where the Python interpreter is currently running

Type bytearray, bytes

_handle
The handle that is used to connect to the device

Type *TcpHandle*

class aio_adb_shell.adb_device.**DeviceFile**(*filename*, *mode*, *size*, *mtime*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('filename', 'mode', 'size', 'mtime')

_fields_defaults = {}

classmethod _make(*iterable*)
Make a new DeviceFile object from a sequence or iterable

_replace(***kwds*)
Return a new DeviceFile object replacing specified fields with new values

property filename
Alias for field number 0

property mode
Alias for field number 1

property mtime
Alias for field number 3

property size
Alias for field number 2

class aio_adb_shell.adb_device.**_AdbTransactionInfo**(*local_id*, *remote_id*, *timeout_s=None*, *total_timeout_s=10.0*)
Bases: object

A class for storing info and settings used during a single ADB "transaction."

Parameters

- **local_id** (*int*) – The ID for the sender (i.e., the device running this code)
- **remote_id** (*int*) – The ID for the recipient

- **timeout_s** (*float, None*) – Timeout in seconds for sending and receiving packets, or `None`; see `BaseHandle.bulk_read()` and `BaseHandle.bulk_write()`
- **total_timeout_s** (*float*) – The total time in seconds to wait for a command in `expected_cmds` in *AdbDevice._read()*

local_id
The ID for the sender (i.e., the device running this code)

Type int

remote_id
The ID for the recipient

Type int

timeout_s
Timeout in seconds for sending and receiving packets, or `None`; see `BaseHandle.bulk_read()` and `BaseHandle.bulk_write()`

Type float, None

total_timeout_s
The total time in seconds to wait for a command in `expected_cmds` in *AdbDevice._read()*

Type float

class aio_adb_shell.adb_device.**_FileSyncTransactionInfo**(*recv_message_format*)
Bases: `object`

A class for storing info used during a single FileSync "transaction."

Parameters **recv_message_format** (*bytes*) – The FileSync message format

recv_buffer
A buffer for storing received data

Type bytearray

recv_message_format
The FileSync message format

Type bytes

recv_message_size
The FileSync message size

Type int

send_buffer
A buffer for storing data to be sent

Type bytearray

send_idx
The index in `recv_buffer` that will be the start of the next data packet sent

Type int

can_add_to_send_buffer(*data_len*)
Determine whether `data_len` bytes of data can be added to the send buffer without exceeding `constants.MAX_ADB_DATA`.

Parameters **data_len** (*int*) – The length of the data to be potentially added to the send buffer (not including the length of its header)

Returns Whether `data_len` bytes of data can be added to the send buffer without exceeding `constants.MAX_ADB_DATA`

Return type bool

`aio_adb_shell.adb_device._open`(*name*, *mode='r'*)

Handle opening and closing of files and IO streams.

Parameters

- **name** (*str, io.IOBase*) – The name of the file *or* an IO stream
- **mode** (*str*) – The mode for opening the file

Yields *io.IOBase* – The opened file *or* the IO stream

aio_adb_shell.adb_message module

Functions and an *AdbMessage* class for packing and unpacking ADB messages.

Contents

class `aio_adb_shell.adb_message.`**AdbMessage**(*command*, *arg0*, *arg1*, *data=b''*)

Bases: `object`

A helper class for packing ADB messages.

Parameters

- **command** (*bytes*) – A command; examples used in this package include `adb_shell.constants.AUTH`, `adb_shell.constants.CNXN`, `adb_shell.constants.CLSE`, `adb_shell.constants.OPEN`, and `adb_shell.constants.OKAY`
- **arg0** (*int*) – Usually the local ID, but `connect()` provides `adb_shell.constants.VERSION`, `adb_shell.constants.AUTH_SIGNATURE`, and `adb_shell.constants.AUTH_RSAPUBLICKEY`
- **arg1** (*int*) – Usually the remote ID, but `connect()` provides `adb_shell.constants.MAX_ADB_DATA`
- **data** (*bytes*) – The data that will be sent

arg0

Usually the local ID, but `connect()` provides `adb_shell.constants.VERSION`, `adb_shell.constants.AUTH_SIGNATURE`, and `adb_shell.constants.AUTH_RSAPUBLICKEY`

Type int

arg1

Usually the remote ID, but `connect()` provides `adb_shell.constants.MAX_ADB_DATA`

Type int

command
The input parameter `command` converted to an integer via `adb_shell.constants.ID_TO_WIRE`

Type int

data
The data that will be sent

Type bytes

magic
`self.command` with its bits flipped; in other words, `self.command + self.magic == 2**32 - 1`

Type int

property checksum
Return `checksum(self.data)`

Returns The checksum of `self.data`

Return type int

pack()
Returns this message in an over-the-wire format.

Returns The message packed into the format required by ADB

Return type bytes

aio_adb_shell.adb_message.**checksum**(*data*)
Calculate the checksum of the provided data.

Parameters **data** (*bytearray, bytes, str*) – The data

Returns The checksum

Return type int

aio_adb_shell.adb_message.**unpack**(*message*)
Unpack a received ADB message.

Parameters **message** (*bytes*) – The received message

Returns

- **cmd** (*int*) – The ADB command
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- **data_length** (*int*) – The length of the data sent by the device (used by `adb_shell.adb_device._read()`)
- **data_checksum** (*int*) – The checksum of the data sent by the device

Raises **ValueError** – Unable to unpack the ADB command.

aio_adb_shell.constants module

Constants used throughout the code.

aio_adb_shell.constants.**AUTH_RSAPUBLICKEY = 3**
AUTH constant for `arg0`

aio_adb_shell.constants.**AUTH_SIGNATURE = 2**
AUTH constant for `arg0`

aio_adb_shell.constants.**AUTH_TOKEN = 1**
AUTH constant for `arg0`

aio_adb_shell.constants.**CLASS = 255**
From adb.h

aio_adb_shell.constants.**DEFAULT_AUTH_TIMEOUT_S = 10.0**
Default authentication timeout (in s) for `adb_shell.tcp_handle.TcpHandle.connect()`

aio_adb_shell.constants.**DEFAULT_PUSH_MODE = 33272**
Default mode for pushed files.

aio_adb_shell.constants.**DEFAULT_TOTAL_TIMEOUT_S = 10.0**
Default total timeout (in s) for `adb_shell.adb_device.AdbDevice._read()`

aio_adb_shell.constants.**FILESYNC_IDS = (b'DATA', b'DENT', b'DONE', b'FAIL', b'LIST', b'OKA**
Commands that are recognized by `adb_shell.adb_device.AdbDevice._filesync_read()`

aio_adb_shell.constants.**FILESYNC_ID_TO_WIRE = {b'DATA': 1096040772, b'DENT': 1414415684, b**
A dictionary where the keys are the commands in *FILESYNC_IDS* and the values are the keys converted to integers

aio_adb_shell.constants.**FILESYNC_LIST_FORMAT = b'<5I'**
The format for FileSync "list" messages

aio_adb_shell.constants.**FILESYNC_PULL_FORMAT = b'<2I'**
The format for FileSync "pull" messages

aio_adb_shell.constants.**FILESYNC_PUSH_FORMAT = b'<2I'**
The format for FileSync "push" messages

aio_adb_shell.constants.**FILESYNC_STAT_FORMAT = b'<4I'**
The format for FileSync "stat" messages

aio_adb_shell.constants.**FILESYNC_WIRE_TO_ID = {1096040772: b'DATA', 1145980243: b'SEND',**
A dictionary where the keys are integers and the values are their corresponding commands (type = bytes) from *FILESYNC_IDS*

aio_adb_shell.constants.**IDS = (b'AUTH', b'CLSE', b'CNXN', b'OKAY', b'OPEN', b'SYNC', b'WRT**
Commands that are recognized by `adb_shell.adb_device.AdbDevice._read()`

aio_adb_shell.constants.**ID_TO_WIRE = {b'AUTH': 1213486401, b'CLSE': 1163086915, b'CNXN': 1**
A dictionary where the keys are the commands in *IDS* and the values are the keys converted to integers

aio_adb_shell.constants.**MAX_ADB_DATA = 4096**
Maximum amount of data in an ADB packet.

aio_adb_shell.constants.**MAX_PUSH_DATA = 2048**
Maximum size of a filesync DATA packet.

aio_adb_shell.constants.**MESSAGE_FORMAT = b'<6I'**
An ADB message is 6 words in little-endian.

aio_adb_shell.constants.**MESSAGE_SIZE = 24**
The size of an ADB message

aio_adb_shell.constants.**PROTOCOL = 1**
From adb.h

aio_adb_shell.constants.**SUBCLASS = 66**
From adb.h

aio_adb_shell.constants.**VERSION = 16777216**
ADB protocol version.

aio_adb_shell.constants.**WIRE_TO_ID = {1129208147: b'SYNC', 1163086915: b'CLSE', 11631540**
A dictionary where the keys are integers and the values are their corresponding commands (type = bytes) from *IDS*

aio_adb_shell.exceptions module

ADB-related exceptions.

exception aio_adb_shell.exceptions.**AdbCommandFailureException**
Bases: Exception

A `b'FAIL'` packet was received.

exception aio_adb_shell.exceptions.**AdbConnectionError**
Bases: Exception

ADB command not sent because a connection to the device has not been established.

exception aio_adb_shell.exceptions.**DeviceAuthError**(*message*, **args*)
Bases: Exception

Device authentication failed.

exception aio_adb_shell.exceptions.**InterleavedDataError**
Bases: Exception

We only support command sent serially.

exception aio_adb_shell.exceptions.**InvalidChecksumError**
Bases: Exception

Checksum of data didn't match expected checksum.

exception aio_adb_shell.exceptions.**InvalidCommandError**(*message*, *response_header*, *response_data*)
Bases: Exception

Got an invalid command.

exception aio_adb_shell.exceptions.**InvalidHandleError**
Bases: Exception

The provided handle does not implement the necessary methods: `close`, `connect`, `bulk_read`, and `bulk_write`.

exception aio_adb_shell.exceptions.**InvalidResponseError**
Bases: Exception

Got an invalid response to our command.

exception aio_adb_shell.exceptions.**PushFailedError**
Bases: Exception

Pushing a file failed for some reason.

exception aio_adb_shell.exceptions.**TcpTimeoutException**
Bases: Exception

TCP connection timed read/write operation exceeded the allowed time.

1.1.3 Module contents

ADB shell functionality.

This Python package implements ADB shell and FileSync functionality. It originated from python-adb.

CHAPTER

TWO

INSTALLATION

```
pip install aio-adb-shell
```

CHAPTER

THREE

EXAMPLE USAGE

(Based on androidtv/adb_manager.py)

```
from aio_adb_shell.adb_device import AdbDeviceTcp
from aio_adb_shell.auth.sign_pythonrsa import PythonRSASigner

# Connect (no authentication necessary)
device1 = AdbDeviceTcp('192.168.0.111', 5555, default_timeout_s=9.)
await device1.connect(auth_timeout_s=0.1)

# Connect (authentication required)
with open('path/to/adbkey') as f:
    priv = f.read()
signer = PythonRSASigner('', priv)
device2 = AdbDeviceTcp('192.168.0.222', 5555, default_timeout_s=9.)
await device2.connect(rsa_keys=[signer], auth_timeout_s=0.1)

# Send a shell command
response1 = await device1.shell('echo TEST1')
response2 = await device2.shell('echo TEST2')
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

S

T

U

V

W