inline-bitcoin

Link

> https://github.com/bitcoin/bitcoin/blob/e25af11225d9d94ecf7068bf7a9a359268786fbe/contrib/v
> binaries/verify.py#L267C1-L267C1

▼ Code

```
def verify_shasums_signature(
    signature_file_path: str, sums_file_path: str, args: argparse.Namespace
) -> t.Tuple[
   ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]
]:
    min_good_sigs = args.min_good_sigs
    gpg_allowed_codes = [0, 2]

    gpg_retval, gpg_output, good, unknown, bad = check_multisig(sums_file_path, signature_file

    if gpg_retval not in gpg_allowed_codes:
        if gpg_retval == 1:
            log.critical(f"Bad signature (code: {gpg_retval}).")
        else:
            log.critical(f"unexpected GPG exit code ({gpg_retval})")

        log.error(f"gpg output:\n{indent(gpg_output)}")
        return (ReturnCode.INTEGRITY_FAILURE, [], [], [], [])


    trusted_keys = set()
    if args.trusted_keys:
        trusted_keys |= set(args.trusted_keys.split(','))

    good_trusted = [sig for sig in good if sig.trusted or sig.key in trusted_keys]
    good_untrusted = [sig for sig in good if sig not in good_trusted]
    num_trusted = len(good_trusted) + len(good_untrusted)
    log.info(f"got {num_trusted} good signatures")

    if num_trusted < min_good_sigs:
        log.info("Maybe you need to import "
                  f"(`gpg --keyserver {args.keyserver} --recv-keys <key-id>`) "
                  "some of the following keys: ")
        log.info('')
        for sig in unknown:
            log.info(f"    {sig.key} ({sig.name})")
        log.info('')
        log.error(
            "not enough trusted sigs to meet threshold "
            f"({num_trusted} vs. {min_good_sigs})")

        return (ReturnCode.NOT_ENOUGH_GOOD_SIGS, [], [], [], [])
    for sig in good_trusted:
        log.info(f"GOOD SIGNATURE: {sig}")

    for sig in good_untrusted:
        log.info(f"GOOD SIGNATURE (untrusted): {sig}")

    for sig in [sig for sig in good if sig.status == 'expired']:
        log.warning(f"key {sig.key} for {sig.name} is expired")

    for sig in bad:
        log.warning(f"BAD SIGNATURE: {sig}")

    for sig in unknown:
        log.warning(f"UNKNOWN SIGNATURE: {sig}")

    return (ReturnCode.SUCCESS, good_trusted, good_untrusted, unknown, bad)
```

Documentation

```
def verify_shasums_signature(
    signature_file_path: str, sums_file_path: str, args: argparse.Namespace
) -> t.Tuple[
   ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]
]:
    min_good_sigs = args.min_good_sigs
    gpg_allowed_codes = [0, 2]  # 2 is returned when untrusted signatures are present.

    gpg_retval, gpg_output, good, unknown, bad = check_multisig(sums_file_path, signature_fil

    if gpg_retval not in gpg_allowed_codes:
        if gpg_retval == 1:
            log.critical(f"Bad signature (code: {gpg_retval}).")
        else:
            log.critical(f"unexpected GPG exit code ({gpg_retval})")

        log.error(f"gpg output:\n{indent(gpg_output)}")
        return (ReturnCode.INTEGRITY_FAILURE, [], [], [], [])

    # Decide which keys we trust, though not "trust" in the GPG sense, but rather
    # which pubkeys convince us that this sums file is legitimate. In other words,
    # which pubkeys within the Bitcoin community do we trust for the purposes of
    # binary verification?
    trusted_keys = set()
    if args.trusted_keys:
        trusted_keys |= set(args.trusted_keys.split(','))

    # Tally signatures and make sure we have enough goods to fulfill
    # our threshold.
    good_trusted = [sig for sig in good if sig.trusted or sig.key in trusted_keys]
    good_untrusted = [sig for sig in good if sig not in good_trusted]
    num_trusted = len(good_trusted) + len(good_untrusted)
    log.info(f"got {num_trusted} good signatures")
    if num_trusted < min_good_sigs:
        log.info("Maybe you need to import "
                  f"(`gpg --keyserver {args.keyserver} --recv-keys <key-id>`) "
                  "some of the following keys: ")
        log.info('')
        for sig in unknown:
            log.info(f"    {sig.key} ({sig.name})")
        log.info('')
        log.error(
            "not enough trusted sigs to meet threshold "
            f"({num_trusted} vs. {min_good_sigs})")

        return (ReturnCode.NOT_ENOUGH_GOOD_SIGS, [], [], [], [])

    for sig in good_trusted:
        log.info(f"GOOD SIGNATURE: {sig}")

    for sig in good_untrusted:
        log.info(f"GOOD SIGNATURE (untrusted): {sig}")

    for sig in [sig for sig in good if sig.status == 'expired']:
        log.warning(f"key {sig.key} for {sig.name} is expired")

    for sig in bad:
        log.warning(f"BAD SIGNATURE: {sig}")

    for sig in unknown:
        log.warning(f"UNKNOWN SIGNATURE: {sig}")

    return (ReturnCode.SUCCESS, good_trusted, good_untrusted, unknown, bad)
```

Prompt

> As a code documentation assistant, you are assigned to document at the in-line level of the given code snippet. When in-line comments are needed, you insert comments within the code itself. The output should be the entire code, along with the documentation you've added. Code is as follows:
>
> ```
> def verify_shasums_signature(
>     signature_file_path: str, sums_file_path: str, args: argparse.Namespace
> ) -> t.Tuple[
>    ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]
> ]:
>     min_good_sigs = args.min_good_sigs
>     gpg_allowed_codes = [0, 2]

```
    gpg_retval, gpg_output, good, unknown, bad = check_multisig(sums_file_path,
signature_file_path, args)

    if gpg_retval not in gpg_allowed_codes:
        if gpg_retval == 1:
            log.critical(f"Bad signature (code: {gpg_retval}).")
        else:
            log.critical(f"unexpected GPG exit code ({gpg_retval})")

        log.error(f"gpg output:\n{indent(gpg_output)}")
        return (ReturnCode.INTEGRITY_FAILURE, [], [], [], [])


    trusted_keys = set()
    if args.trusted_keys:
        trusted_keys |= set(args.trusted_keys.split(','))

    good_trusted = [sig for sig in good if sig.trusted or sig.key in trusted_keys]
    good_untrusted = [sig for sig in good if sig not in good_trusted]
    num_trusted = len(good_trusted) + len(good_untrusted)
    log.info(f"got {num_trusted} good signatures")

    if num_trusted < min_good_sigs:
        log.info("Maybe you need to import "
                 f"(`gpg --keyserver {args.keyserver} --recv-keys <key-id>`) "
                 "some of the following keys: ")
        log.info('')
        for sig in unknown:
            log.info(f"    {sig.key} ({sig.name})")
        log.info('')
        log.error(
            "not enough trusted sigs to meet threshold "
            f"({num_trusted} vs. {min_good_sigs})")

        return (ReturnCode.NOT_ENOUGH_GOOD_SIGS, [], [], [], [])

    for sig in good_trusted:
        log.info(f"GOOD SIGNATURE: {sig}")

    for sig in good_untrusted:
        log.info(f"GOOD SIGNATURE (untrusted): {sig}")
```

> ```
>     for sig in [sig for sig in good if sig.status == 'expired']:
>         log.warning(f"key {sig.key} for {sig.name} is expired")
>
>     for sig in bad:
>         log.warning(f"BAD SIGNATURE: {sig}")
>
>     for sig in unknown:
>         log.warning(f"UNKNOWN SIGNATURE: {sig}")
>
>     return (ReturnCode.SUCCESS, good_trusted, good_untrusted, unknown, bad)
> ```

Responses

ChatGPT

```
import argparse
import typing as t

def verify_shasums_signature(
    signature_file_path: str, sums_file_path: str, args: argparse.Namespace
) -> t.Tuple[
   ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]
]:
    # Get the minimum number of good signatures required and allowed GPG exit codes
    min_good_sigs = args.min_good_sigs
    gpg_allowed_codes = [0, 2]

    # Call the check_multisig function to verify signatures
    gpg_retval, gpg_output, good, unknown, bad = check_multisig(sums_file_path, signature_file_p

    # Handle different GPG exit codes
    if gpg_retval not in gpg_allowed_codes:
        if gpg_retval == 1:
            log.critical(f"Bad signature (code: {gpg_retval}).")
        else:
            log.critical(f"unexpected GPG exit code ({gpg_retval})")

        log.error(f"gpg output:\n{indent(gpg_output)}")
        return (ReturnCode.INTEGRITY_FAILURE, [], [], [], [])

    # Create a set of trusted keys if provided in command-line arguments
    trusted_keys = set()
    if args.trusted_keys:
        trusted_keys |= set(args.trusted_keys.split(','))

    # Separate good signatures into trusted and untrusted based on keys
    good_trusted = [sig for sig in good if sig.trusted or sig.key in trusted_keys]
    good_untrusted = [sig for sig in good if sig not in good_trusted]
```

```
    num_trusted = len(good_trusted) + len(good_untrusted)
    log.info(f"got {num_trusted} good signatures")

    # Check if the number of trusted signatures meets the threshold
    if num_trusted < min_good_sigs:
        log.info("Maybe you need to import "
                 f"(`gpg --keyserver {args.keyserver} --recv-keys <key-id>`) "
                 "some of the following keys: ")
        log.info('')
        for sig in unknown:
            log.info(f"    {sig.key} ({sig.name})")
        log.info('')
        log.error(
            "not enough trusted sigs to meet threshold "
            f"({num_trusted} vs. {min_good_sigs})")

        return (ReturnCode.NOT_ENOUGH_GOOD_SIGS, [], [], [], [])

    # Log information about good, untrusted, expired, bad, and unknown signatures
    for sig in good_trusted:
        log.info(f"GOOD SIGNATURE: {sig}")

    for sig in good_untrusted:
        log.info(f"GOOD SIGNATURE (untrusted): {sig}")

    for sig in [sig for sig in good if sig.status == 'expired']:
        log.warning(f"key {sig.key} for {sig.name} is expired")

    for sig in bad:
        log.warning(f"BAD SIGNATURE: {sig}")

    for sig in unknown:
        log.warning(f"UNKNOWN SIGNATURE: {sig}")

    return (ReturnCode.SUCCESS, good_trusted, good_untrusted, unknown, bad)
```

GPT4

```
def verify_shasums_signature(
    signature_file_path: str, sums_file_path: str, args: argparse.Namespace
) -> t.Tuple[
   ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]
]:
    # Minimum number of good signatures required, taken from command line arguments
    min_good_sigs = args.min_good_sigs

    # GPG return codes that are considered acceptable
    gpg_allowed_codes = [0, 2]

    # Check the multisignature of the sums file against the signature file
    gpg_retval, gpg_output, good, unknown, bad = check_multisig(sums_file_path, signature_file_p

    # Check if the GPG return value is not within the allowed codes
    if gpg_retval not in gpg_allowed_codes:
        # Log critical error messages based on the GPG return value
        if gpg_retval == 1:
            log.critical(f"Bad signature (code: {gpg_retval}).")
        else:
            log.critical(f"unexpected GPG exit code ({gpg_retval})")

        # Log the output of the GPG process for debugging
        log.error(f"gpg output:\n{indent(gpg_output)}")

        # Return an integrity failure code with empty signature lists
        return (ReturnCode.INTEGRITY_FAILURE, [], [], [], [])

    # Initialize an empty set for trusted keys
    trusted_keys = set()
    # If trusted keys are provided as arguments, add them to the set
    if args.trusted_keys:
        trusted_keys |= set(args.trusted_keys.split(','))

    # Filter good signatures into trusted and untrusted categories
    good_trusted = [sig for sig in good if sig.trusted or sig.key in trusted_keys]
    good_untrusted = [sig for sig in good if sig not in good_trusted]

    # Count the total number of trusted signatures
    num_trusted = len(good_trusted) + len(good_untrusted)

    # Log the number of good signatures found
    log.info(f"got {num_trusted} good signatures")

    # Check if the number of trusted signatures meets the required minimum
    if num_trusted < min_good_sigs:
        # Suggest importing keys for unknown signatures
        log.info("Maybe you need to import "
                 f"(`gpg --keyserver {args.keyserver} --recv-keys <key-id>`) "
                 "some of the following keys: ")
        for sig in unknown:
            log.info(f"    {sig.key} ({sig.name})")
        log.error(
            "not enough trusted sigs to meet threshold "
            f"({num_trusted} vs. {min_good_sigs})")

        # Return a failure code if the threshold is not met
        return (ReturnCode.NOT_ENOUGH_GOOD_SIGS, [], [], [], [])

    # Log the details of good and trusted signatures
    for sig in good_trusted:
        log.info(f"GOOD SIGNATURE: {sig}")

    # Log the details of good but untrusted signatures
    for sig in good_untrusted:
        log.info(f"GOOD SIGNATURE (untrusted): {sig}")

    # Warn about any expired signatures
    for sig in [sig for sig in good if sig.status == 'expired']:
            log.warning(f"key {sig.key} for {sig.name} is expired")

    # Warn about bad signatures
    for sig in bad:
        log.warning(f"BAD SIGNATURE: {sig}")

    # Warn about unknown signatures
    for sig in unknown:
        log.warning(f"UNKNOWN SIGNATURE: {sig}")

    # Return success code along with the lists of good, unknown, and bad signatures
    return (ReturnCode.SUCCESS, good_trusted, good_untrusted, unknown, bad)
```

Bard

```
def verify_shasums_signature(
    signature_file_path: str,
    sums_file_path: str,
    args: argparse.Namespace,
) -> t.Tuple[ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]]:
    """
    Verifies the signatures in the given signature file against the given checksums file.

    Args:
        signature_file_path (str): The path to the signature file.
        sums_file_path (str): The path to the checksums file.
        args (argparse.Namespace): The command-line arguments.

    Returns:
        t.Tuple[ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]]:
            A tuple containing the return code, the list of good signatures, the list of untrust
            the list of unknown signatures, and the list of bad signatures.
    """
    # Set the minimum number of good signatures required
    min_good_sigs = args.min_good_sigs

    # Set the allowed GPG exit codes
    gpg_allowed_codes = [0, 2]

    # Check the multisig signatures
    gpg_retval, gpg_output, good, unknown, bad = check_multisig(sums_file_path, signature_file_p

    # Handle unexpected GPG exit codes
    if gpg_retval not in gpg_allowed_codes:
        if gpg_retval == 1:
            log.critical(f"Bad signature (code: {gpg_retval}).")
        else:
            log.critical(f"unexpected GPG exit code ({gpg_retval})")
        log.error(f"gpg output:\n{indent(gpg_output)}")
        return (ReturnCode.INTEGRITY_FAILURE, [], [], [], [])

    # Process trusted keys
    trusted_keys = set()
    if args.trusted_keys:
        trusted_keys |= set(args.trusted_keys.split(','))

    # Divide good signatures into trusted and untrusted
    good_trusted = [sig for sig in good if sig.trusted or sig.key in trusted_keys]
    good_untrusted = [sig for sig in good if sig not in good_trusted]

    # Count the number of trusted signatures
    num_trusted = len(good_trusted) + len(good_untrusted)

    # Log information about the number of good signatures
    log.info(f"got {num_trusted} good signatures")

    # Check if there are enough trusted signatures
    if num_trusted < min_good_sigs:
        log.info("Maybe you need to import" f"(`gpg --keyserver {args.keyserver} --recv-keys <ke
        log.info('')

        # Log the unknown keys
        for sig in unknown:
            log.info(f" {sig.key} ({sig.name})")
        log.info('')

        # Log an error message about the insufficient number of trusted signatures
        log.error(
            f"not enough trusted sigs to meet threshold " f"({num_trusted} vs. {min_good_sigs})"
        )

        # Return an error code
        return (ReturnCode.NOT_ENOUGH_GOOD_SIGS, [], [], [], [])

    # Log information about the trusted signatures
    for sig in good_trusted:
        log.info(f"GOOD SIGNATURE: {sig}")

    # Log information about the untrusted signatures
    for sig in good_untrusted:
        log.info(f"GOOD SIGNATURE (untrusted): {sig}")

    # Log warnings about expired signatures
    for sig in [sig for sig in good if sig.status == 'expired']:
        log.warning(f"key {sig.key} for {sig.name} is expired")

    # Log warnings about bad signatures
    for sig in bad:
        log.warning(f"BAD SIGNATURE: {sig}")

    # Log warnings about unknown signatures
    for sig in unknown:
        log.warning(f"UNKNOWN SIGNATURE: {sig}")

    # Return a success code and the lists of signatures
    return (ReturnCode.SUCCESS, good_trusted, good_untrusted, unknown, bad)
```

LLama2

```
def verify_shasums_signature(
    signature_file_path: str, sums_file_path: str, args: argparse.Namespace
) -> t.Tuple[
   ReturnCode, t.List[SigData], t.List[SigData], t.List[SigData], t.List[SigData]
]:
    # Minimum number of good signatures required
    min_good_sigs = args.min_good_sigs

    # Allowable GPG exit codes
    gpg_allowed_codes = [0, 2]

    # Run GPG verification
    gpg_retval, gpg_output, good, unknown, bad = check_multisig(sums_file_path, signature_file_p

    # Check GPG exit code
    if gpg_retval not in gpg_allowed_codes:
        if gpg_retval == 1:
            log.critical(f"Bad signature (code: {gpg_retval}).")
        else:
            log.critical(f"Unexpected GPG exit code ({gpg_retval})")

        log.error(f"GPG output: \n{indent(gpg_output)}")
        return (ReturnCode.INTEGRITY_FAILURE, [], [], [], [])

    # Extract trusted keys from command line argument
    trusted_keys = set()
    if args.trusted_keys:
        trusted_keys |= set(args.trusted_keys.split(','))

    # Filter good signatures based on trusted keys
    good_trusted = [sig for sig in good if sig.trusted or sig.key in trusted_keys]
    good_untrusted = [sig for sig in good if sig not in good_trusted]
    num_trusted = len(good_trusted) + len(good_untrusted)
    log.info(f"got {num_trusted} good signatures")

    # Check if minimum number of good signatures met
    if num_trusted < min_good_sigs:
        log.info("Maybe you need to import some of the following keys: ")
        log.info('')
        for sig in unknown:
            log.info(f"    {sig.key} ({sig.name})")
        log.info('')
        log.error(
            "Not enough trusted sigs to meet threshold "
            f"({num_trusted} vs. {min_good_sigs})")

        return (ReturnCode.NOT_ENOUGH_GOOD_SIGS, [], [], [], [])

    # Print GOOD SIGNATURE messages
    for sig in good_trusted:
        log.info(f"GOOD SIGNATURE: {sig}")
    for sig in good_untrusted:
        log.info(f"GOOD SIGNATURE (untrusted): {sig}")

    # Print WARNING messages for expired keys
    for sig in [sig for sig in good if sig.status == 'expired']:
        log.warning(f"Key {sig.key} for {sig.name} is expired")

    # Print BAD SIGNATURE messages
    for sig in bad:
        log.warning(f"BAD SIGNATURE: {sig}")

    # Print UNKNOWN SIGNATURE messages
    for sig in unknown:
        log.warning(f"UNKNOWN SIGNATURE: {sig}")

    return (ReturnCode.SUCCESS, good_trusted, good_untrusted, unknown, bad)
```

StarChat

```
-
```
