Django SQRL Documentation

Release 0.1

Miroslav Shubernetskiy

Sep 20, 2017

Contents

CHAPTER 1

Contents

Installation

At the command line:

```
$ easy_install django-sqrl
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv django-sqrl
$ pip install django-sqrl
```

Usage

TODO

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/miki725/django-sqrl/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Django SQRL could always use more documentation, whether as part of the official Django SQRL docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/miki725/django-sqrl/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *django-sqrl* for local development.

1. Fork the *django-sqrl* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-sqrl.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-sqrl
$ cd django-sqrl/
$ make install
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ make lint
$ make test-all
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, and for PyPy. Check https://travis-ci.org/miki725/django-sqrl/pull_requests and make sure that the tests pass for all supported Python versions.

Credits

Development Lead

- Miroslav Shubernetskiy - https://github.com/miki725

Contributors

None yet. Why not be the first?

History

0.1.0 (2015-05-20)

- First release on PyPI.

API Documentation

sqrl package

Subpackages

sqrl.management package

Subpackages

sqrl.management.commands package

Submodules

sqrl.management.commands.clearsqrlnuts module

sqrl.management.commands.clearsqrlnuts module

sqrl.templatetags package

Submodules

sqrl.templatetags.sqrl module

sqrl.templatetags.sqrl module

Submodules

sqrl.admin module

sqrl.backends module

sqrl.crypto module

class `sqrl.crypto.`**`Ed25519`**(*public_key*, *private_key*, *msg*)

Bases: `object`

Utility class for signing and verifying ed25519 signatures.

More information about ed25519 can be found at http://ed25519.cr.yp.to/.

Parameters

- **`public_key`** (*bytes*) – Key used for verifying signature.
- **`private_key`** (*bytes*) – Key used for signing data.
- **`msg`** (*bytes*) – Binary data for which to generate the signature.

`is_signature_valid`(*other_signature*)

Check if `other_signature` is a valid signature for the provided message.

Returns Boolean indicating whether validation has succeeded.

Return type bool

`sign_data`()

Generate ed25519 signature for the provided data.

Returns ed25519 signature

Return type bytes

class sqrl.crypto.**HMAC**(*nut*, *data*)
Bases: object

Utility class for generating and verifying HMAC signatures.

This class relies on Django's built in salted_hmac() to compute actual HMAC values by using SECRET_KEY as key.

Parameters

- **nut** (*SQRLNut*) – Nut from which necessary data is extracted to add a salt value to the HMAC input data. Currently only models.SQRLNut.session_key is used.
- **data** (*OrderedDict*) – Dict for which to either compute or validate HMAC signature.

is_signature_valid(*other_signature*)
Check if the other_signature is a valid signature for the provided data and the nut.

Returns Boolean indicating whether validation has succeeded.

Return type bool

sign_data()
Generate HMAC signature for the provided data.

Note: max key is ignored in the input data if that key is present.

Returns Binary signature of the data

Return type bytes

sqrl.crypto.**generate_randomness**(*bytes=32*)
Generate random sample of specified size bytes.

Parameters **bytes** (*int, optional*) – Number of bytes to generate random sample

Returns *Base64.encode()* encoded random sample

Return type str

sqrl.exceptions module

class sqrl.exceptions.**TIF**
Bases: int

SQRL TIF int subclass which can represent SQRL TIF flags.

Example

```
>>> tif = TIF(TIF.IP_MATCH | TIF.TRANSIENT_FAILURE | TIF.COMMAND_FAILED)
>>> tif.is_ip_match
True
>>> tif.is_id_match
False
>>> tif.is_transient_failure
True
>>> tif
100
```

```
>>> tif.as_hex_string()
'64'
>>> tif.breakdown() == {
...     'id_match': False,
...     'previous_id_match': False,
...     'ip_match': True,
...     'sqrl_disabled': False,
...     'not_supported': False,
...     'transient_failure': True,
...     'command_failed': True,
...     'client_failure': False,
... }
True
```

BAD_ID_ASSOCIATION = 256
SQRL Identity is already a ssociated with a different account

CLIENT_FAILURE = 128
SQRL command failed because SQRL client sent invalid data

COMMAND_FAILED = 64
SQRL command failed for any reason

ID_MATCH = 1
SQRL ID was found in DB

IP_MATCH = 4
SQRL client is used from same IP as where transaction started

NOT_SUPPORTED = 16
SQRL client requested SQRl operation which is not supported

PREVIOUS_ID_MATCH = 2
Previous SQRL ID was found in DB

SQRL_DISABLED = 8
SQRL auth is disabled for the found SQRL identity as per users request

TRANSIENT_FAILURE = 32
SQRL command failed transiently. Most likely restarting SQRL transaction should fix this

as_hex_string()
Return TIF value as hex string

breakdown()
Returns a full breakdown of the TIF value.

Returns Keys are the SQRL TIF property and values are booleans.

Return type dict

is_bad_id_association
Property which returns boolean whether `0x100` or `0b100000000` bit is present in the TIF value.

is_client_failure
Property which returns boolean whether `0x80` or `0b10000000` bit is present in the TIF value.

is_command_failed
Property which returns boolean whether `0x40` or `0b1000000` bit is present in the TIF value.

is_id_match
Property which returns boolean whether `0x1` or `0b1` bit is present in the TIF value.

is_ip_match
Property which returns boolean whether `0x4` or `0b100` bit is present in the TIF value.

is_not_supported
Property which returns boolean whether `0x10` or `0b10000` bit is present in the TIF value.

is_previous_id_match
Property which returns boolean whether `0x2` or `0b10` bit is present in the TIF value.

is_sqrl_disabled
Property which returns boolean whether `0x8` or `0b1000` bit is present in the TIF value.

is_transient_failure
Property which returns boolean whether `0x20` or `0b100000` bit is present in the TIF value.

update(*other*)
Return updated TIF which will contain both bits already set in the `self` value as well as the ‘‘other value.

Parameters **other** (*int*) – Other `TIF` value which be merged with `self` bits

Returns New *TIF* value which has merged bits.

Return type *TIF*

exception sqrl.exceptions.**TIFException**(*tif*)
Bases: `Exception`

Custom Exception which can be used in the views to raise specific *TIF* bits and immediately return appropriate response to the user.

sqrl.fields module

class sqrl.fields.**Base64CharField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=’‘*, **args*, ***kwargs*)
Bases: *sqrl.fields.Base64Field*

Similar to *Base64Field* however this field normalizes to `str` (`unicode`) data.

default_error_messages = {‘base64_ascii’: ‘Invalid value. Must be ascii base64url encoded string.’}

to_python(*value*)
Returns base64 decoded data as string.

Uses *Base64Field.to_python()* to decode base64 value which returns binary data and then this method further decodes ascii data to return `str` (`unicode`) data.

class sqrl.fields.**Base64ConditionalPairsField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=’‘*, **args*, ***kwargs*)
Bases: *sqrl.fields.Base64PairsField*

Similar to *Base64PairsField* but this field does not force the value to be keypairs.

always_pairs = False

class sqrl.fields.**Base64Field**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=’‘*, **args*, ***kwargs*)
Bases: `django.forms.fields.CharField`

Field which decodes base64 values using *utils.Base64.decode()*.

default_error_messages = {‘base64’: ‘Invalid value. Must be base64url encoded string.’}

to_python(*value*)
Decodes base64 value and returns binary data.

class sqrl.fields.**Base64PairsField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)
Bases: *sqrl.fields.Base64CharField*

Field which normalizes base64 encoded multistring key-value pairs to `OrderedDict`.

always_pairs
bool – Boolean which enforces that the value must always be keypairs. When `False` and the value is not a keypair, the value itself is returned.

always_pairs = True

default_error_messages = {'crlf': 'Invalid value. Must be multi-line string separated by CRLF.', 'pairs': 'Invalid v

to_python(*value*)
Normalizes multiline base64 keypairs string to `OrderedDict`.

class sqrl.fields.**ExtractedNextUrlField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)
Bases: *sqrl.fields.NextUrlField*

Similar to *NextUrlField* however this extracts next url from full encoded URL.

default_error_messages = {'missing_next': 'Missing next query parameter.'}

to_python(*value*)
Extract next url from full URL string and then use *NextUrlField* to validate that value is valid URL.

class sqrl.fields.**NextUrlField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)
Bases: `django.forms.fields.CharField`

Custom `CharField` which validates that a value is a valid next URL.

It validates that by checking that the value can be resolved to a view hence guaranteeing that when redirected URL will not fail.

default_error_messages = {'invalid_url': 'Invalid next url.'}

to_python(*value*)
Validate that value is a valid URL for this project.

class sqrl.fields.**SQRLURLField**(**args*, ***kwargs*)
Bases: `django.forms.fields.URLField`

SQRL URL field which uses *SQRLURLValidator* for validation.

default_validators = [<sqrl.fields.SQRLURLValidator object>]

class sqrl.fields.**SQRLURLValidator**(*schemes=None*, ***kwargs*)
Bases: `django.core.validators.URLValidator`

Custom URL validator which validates that a URL is a valid SQRL url.

These are the differences with regular HTTP URLs:

- scheme is either sqrl (secure) and qrl (non-secure)
- `:` is a valid path separator which can be used to indicate which section of the SQRL should be used to generate public/provate keypair for the domain.

schemes = ['sqrl', 'qrl']

class sqrl.fields.**TildeMultipleValuesField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)

Bases: django.forms.fields.CharField

Field which returns tilde-separated list.

to_python(*value*)

Normalizes to a Python list by splitting string by tilde (~) delimiter.

class sqrl.fields.**TildeMultipleValuesFieldChoiceField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)

Bases: *sqrl.fields.TildeMultipleValuesField*, django.forms.fields.ChoiceField

Similar to *TildeMultipleValuesField* however this field also validates each value to be a valid choice.

validate(*value*)

sqrl.forms module

sqrl.managers module

class sqrl.managers.**SQRLNutManager**

Bases: django.db.models.manager.Manager

Customer models.SQRLNut model manager.

replace_or_create(*session_key*, ***kwargs*)

This method creates new models.SQRLNut with given parameters.

If nut already exists, it removes it before creating new nut.

Parameters

- **session_key** (*str*) – Key of the session. All nuts with matching session will be removed.
- ****kwargs** – Kwargs which will be used to create new models.SQRLNut

sqrl.models module

sqrl.response module

class sqrl.response.**SQRLHttpResponse**(*nut*, *data*, **args*, ***kwargs*)

Bases: django.http.response.HttpResponse

Custom HTTPResponse class used to return SQRL-formatted response.

The response is automatically signed, normalized and encoded as per SQRL specification.

This view also adds a couple of DEBUG logs for easier SQRL debugging and also returns all SQRL data back as X-SQRL-* headers.

Parameters

- **nut** (*SQRLNut*) – Nut which will be used to sign the response data.
- **data** (*OrderedDict*) – Data to be returned back to the user.

sign_response(*nut*, *data*)
When nut is present, this method signs the data by adding `mac` key.

For signing *crypto.HMAC.sign_data()* is used.

sqrl.sqrl module

sqrl.urls module

sqrl.utils module

class sqrl.utils.**Base64**
Bases: object

Helper class for base64 encoding/decoding

classmethod decode(*s*)
Decode unicode string from base64 where remaining "=" characters were stripped.

Parameters **s** (*str*) – Unicode string to be decoded from base64

classmethod encode(*s*)
Encode binary string as base64. Remaining "=" characters are removed.

Parameters **s** (*bytes*) – Bytes string to be encoded as base64

class sqrl.utils.**Encoder**
Bases: object

Helper class for encoding/decoding SQRL response data.

classmethod base64_dumps(*data*)
Dumps given data into a single Base64 string.

Practically this is the same as *dumps()* except *dumps()* can return multiline string for `dict`. This method normalizes that further by converting that multiline string to a single base64 encoded value.

Returns Base64 encoded binary data of input `data`

Return type binary

classmethod dumps(*data*)
Recursively dumps given data to SQRL response format.

Before data is dumped out, it is normalized by using *normalize()*.

This dumps each data type as follows:

Dict returns an `\r\n` multiline string. Each line is for a single key-pair of format `<key>=<dumped value>`.

List tilde (~) joined dumped list of values

Other no operation

classmethod normalize(*data*)
Recursively normalize data for encoding.

This encodes each data type as follows:

Dict returns an `OrderedDict` where all values are recursively normalized. Empty dict is normalized to empty string

List each value is recursively normalized

Binary Base64 encode data

Str no operation

Other data is casted to string using `__str__` (or `__unicode__`)

class `sqrl.utils.`**`QRGenerator`**(*url*)
Bases: `object`

Helper class for generating a QR image for the given SQRL url.

Parameters **`url`** (*`str`*) – URL for which to generate QR image

`generate_image`()
Generate QR image and get its binary data.

Returns Binary data of the png image file which can directly be returned to the user

Return type bytes

`sqrl.utils.`**`get_user_ip`**(*request*)
Utility function for getting user's IP from request address.

This either returns the IP address from the `request.REMOTE_ADDR` or `request.META'HTTP_X_REAL_IP']` when request might of been reverse proxied.

sqrl.views module

sqrl.admin module

sqrl.backends module

sqrl.crypto module

class `sqrl.crypto.`**`Ed25519`**(*public_key*, *private_key*, *msg*)
Bases: `object`

Utility class for signing and verifying ed25519 signatures.

More information about ed25519 can be found at http://ed25519.cr.yp.to/.

Parameters

- **`public_key`** (*`bytes`*) – Key used for verifying signature.
- **`private_key`** (*`bytes`*) – Key used for signing data.
- **`msg`** (*`bytes`*) – Binary data for which to generate the signature.

`is_signature_valid`(*other_signature*)
Check if `other_signature` is a valid signature for the provided message.

Returns Boolean indicating whether validation has succeeded.

Return type bool

`sign_data`()
Generate ed25519 signature for the provided data.

Returns ed25519 signature

Return type bytes

class sqrl.crypto.**HMAC**(*nut*, *data*)
Bases: object

Utility class for generating and verifying HMAC signatures.

This class relies on Django's built in salted_hmac() to compute actual HMAC values by using SECRET_KEY as key.

Parameters

- **nut** (*SQRLNut*) – Nut from which necessary data is extracted to add a salt value to the HMAC input data. Currently only models.SQRLNut.session_key is used.
- **data** (*OrderedDict*) – Dict for which to either compute or validate HMAC signature.

is_signature_valid(*other_signature*)
Check if the other_signature is a valid signature for the provided data and the nut.

Returns Boolean indicating whether validation has succeeded.

Return type bool

sign_data()
Generate HMAC signature for the provided data.

Note: max key is ignored in the input data if that key is present.

Returns Binary signature of the data

Return type bytes

sqrl.crypto.**generate_randomness**(*bytes=32*)
Generate random sample of specified size bytes.

Parameters **bytes** (*int, optional*) – Number of bytes to generate random sample

Returns *Base64.encode()* encoded random sample

Return type str

sqrl.exceptions module

class sqrl.exceptions.**TIF**
Bases: int

SQRL TIF int subclass which can represent SQRL TIF flags.

Example

```
>>> tif = TIF(TIF.IP_MATCH | TIF.TRANSIENT_FAILURE | TIF.COMMAND_FAILED)
>>> tif.is_ip_match
True
>>> tif.is_id_match
False
>>> tif.is_transient_failure
True
>>> tif
100
```

```
>>> tif.as_hex_string()
'64'
>>> tif.breakdown() == {
...     'id_match': False,
...     'previous_id_match': False,
...     'ip_match': True,
...     'sqrl_disabled': False,
...     'not_supported': False,
...     'transient_failure': True,
...     'command_failed': True,
...     'client_failure': False,
... }
True
```

BAD_ID_ASSOCIATION = 256
SQRL Identity is already a ssociated with a different account

CLIENT_FAILURE = 128
SQRL command failed because SQRL client sent invalid data

COMMAND_FAILED = 64
SQRL command failed for any reason

ID_MATCH = 1
SQRL ID was found in DB

IP_MATCH = 4
SQRL client is used from same IP as where transaction started

NOT_SUPPORTED = 16
SQRL client requested SQRl operation which is not supported

PREVIOUS_ID_MATCH = 2
Previous SQRL ID was found in DB

SQRL_DISABLED = 8
SQRL auth is disabled for the found SQRL identity as per users request

TRANSIENT_FAILURE = 32
SQRL command failed transiently. Most likely restarting SQRL transaction should fix this

as_hex_string()
Return TIF value as hex string

breakdown()
Returns a full breakdown of the TIF value.

Returns Keys are the SQRL TIF property and values are booleans.

Return type dict

is_bad_id_association
Property which returns boolean whether `0x100` or `0b100000000` bit is present in the TIF value.

is_client_failure
Property which returns boolean whether `0x80` or `0b10000000` bit is present in the TIF value.

is_command_failed
Property which returns boolean whether `0x40` or `0b1000000` bit is present in the TIF value.

is_id_match
Property which returns boolean whether `0x1` or `0b1` bit is present in the TIF value.

is_ip_match
Property which returns boolean whether `0x4` or `0b100` bit is present in the TIF value.

is_not_supported
Property which returns boolean whether `0x10` or `0b10000` bit is present in the TIF value.

is_previous_id_match
Property which returns boolean whether `0x2` or `0b10` bit is present in the TIF value.

is_sqrl_disabled
Property which returns boolean whether `0x8` or `0b1000` bit is present in the TIF value.

is_transient_failure
Property which returns boolean whether `0x20` or `0b100000` bit is present in the TIF value.

update(*other*)
Return updated TIF which will contain both bits already set in the `self` value as well as the ‘‘other value.

Parameters **other** (*int*) – Other `TIF` value which be merged with `self` bits

Returns New *TIF* value which has merged bits.

Return type *TIF*

exception `sqrl.exceptions.`**TIFException**(*tif*)
Bases: `Exception`

Custom Exception which can be used in the views to raise specific *TIF* bits and immediately return appropriate response to the user.

`sqrl.exceptions.`**_make_tif_property**(*val*)
Helper function for generating property methods for *TIF* which will boolean whether a particular SQRL `TIF` bit is `True` in the `TIF` value.

Parameters **val** (*int*) – Value with particular `True` bit which will be tested within the generated property.

Returns Function which can be made into a property

Return type function

sqrl.fields module

class `sqrl.fields.`**Base64CharField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=’‘*, **args*, ***kwargs*)
Bases: *sqrl.fields.Base64Field*

Similar to *Base64Field* however this field normalizes to `str` (`unicode`) data.

default_error_messages = {‘base64_ascii’: ‘Invalid value. Must be ascii base64url encoded string.’}

to_python(*value*)
Returns base64 decoded data as string.

Uses *Base64Field.to_python()* to decode base64 value which returns binary data and then this method further decodes ascii data to return `str` (`unicode`) data.

class `sqrl.fields.`**Base64ConditionalPairsField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=’‘*, **args*, ***kwargs*)
Bases: *sqrl.fields.Base64PairsField*

Similar to *Base64PairsField* but this field does not force the value to be keypairs.

always_pairs = False

class sqrl.fields.**Base64Field**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)
Bases: django.forms.fields.CharField

Field which decodes base64 values using *utils.Base64.decode()*.

default_error_messages = {'base64': 'Invalid value. Must be base64url encoded string.'}

to_python(*value*)
Decodes base64 value and returns binary data.

class sqrl.fields.**Base64PairsField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)
Bases: *sqrl.fields.Base64CharField*

Field which normalizes base64 encoded multistring key-value pairs to OrderedDict.

always_pairs
bool – Boolean which enforces that the value must always be keypairs. When False and the value is not a keypair, the value itself is returned.

always_pairs = True

default_error_messages = {'crlf': 'Invalid value. Must be multi-line string separated by CRLF.', 'pairs': 'Invalid v

to_python(*value*)
Normalizes multiline base64 keypairs string to OrderedDict.

class sqrl.fields.**ExtractedNextUrlField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)
Bases: *sqrl.fields.NextUrlField*

Similar to *NextUrlField* however this extracts next url from full encoded URL.

default_error_messages = {'missing_next': 'Missing next query parameter.'}

to_python(*value*)
Extract next url from full URL string and then use *NextUrlField* to validate that value is valid URL.

class sqrl.fields.**NextUrlField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)
Bases: django.forms.fields.CharField

Custom CharField which validates that a value is a valid next URL.

It validates that by checking that the value can be resolved to a view hence guaranteeing that when redirected URL will not fail.

default_error_messages = {'invalid_url': 'Invalid next url.'}

to_python(*value*)
Validate that value is a valid URL for this project.

class sqrl.fields.**SQRLURLField**(**args*, ***kwargs*)
Bases: django.forms.fields.URLField

SQRL URL field which uses *SQRLURLValidator* for validation.

default_validators = [<sqrl.fields.SQRLURLValidator object>]

class sqrl.fields.**SQRLURLValidator**(*schemes=None*, ***kwargs*)
Bases: django.core.validators.URLValidator

Custom URL validator which validates that a URL is a valid SQRL url.

These are the differences with regular HTTP URLs:

- scheme is either sqrl (secure) and qrl (non-secure)
- `:` is a valid path separator which can be used to indicate which section of the SQRL should be used to generate public/provate keypair for the domain.

schemes = ['sqrl', 'qrl']

class sqrl.fields.**TildeMultipleValuesField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)

Bases: django.forms.fields.CharField

Field which returns tilde-separated list.

to_python(*value*)

Normalizes to a Python list by splitting string by tilde (~) delimiter.

class sqrl.fields.**TildeMultipleValuesFieldChoiceField**(*max_length=None*, *min_length=None*, *strip=True*, *empty_value=''*, **args*, ***kwargs*)

Bases: *sqrl.fields.TildeMultipleValuesField*, django.forms.fields.ChoiceField

Similar to *TildeMultipleValuesField* however this field also validates each value to be a valid choice.

validate(*value*)

sqrl.forms module

sqrl.managers module

class sqrl.managers.**SQRLNutManager**

Bases: django.db.models.manager.Manager

Customer models.SQRLNut model manager.

replace_or_create(*session_key*, ***kwargs*)

This method creates new models.SQRLNut with given parameters.

If nut already exists, it removes it before creating new nut.

Parameters

- **session_key** (*str*) – Key of the session. All nuts with matching session will be removed.
- ****kwargs** – Kwargs which will be used to create new models.SQRLNut

sqrl.models module

sqrl.response module

class sqrl.response.**SQRLHttpResponse**(*nut*, *data*, **args*, ***kwargs*)

Bases: django.http.response.HttpResponse

Custom HTTPResponse class used to return SQRL-formatted response.

The response is automatically signed, normalized and encoded as per SQRL specification.

This view also adds a couple of DEBUG logs for easier SQRL debugging and also returns all SQRL data back as X-SQRL-* headers.

Parameters

- **nut** (*SQRLNut*) – Nut which will be used to sign the response data.
- **data** (*OrderedDict*) – Data to be returned back to the user.

sign_response(*nut*, *data*)
When nut is present, this method signs the data by adding `mac` key.

For signing *crypto.HMAC.sign_data()* is used.

sqrl.sqrl module

sqrl.urls module

sqrl.utils module

class sqrl.utils.**Base64**
Bases: object

Helper class for base64 encoding/decoding

classmethod decode(*s*)
Decode unicode string from base64 where remaining "=" characters were stripped.

Parameters **s** (*str*) – Unicode string to be decoded from base64

classmethod encode(*s*)
Encode binary string as base64. Remaining "=" characters are removed.

Parameters **s** (*bytes*) – Bytes string to be encoded as base64

class sqrl.utils.**Encoder**
Bases: object

Helper class for encoding/decoding SQRL response data.

classmethod base64_dumps(*data*)
Dumps given data into a single Base64 string.

Practically this is the same as *dumps()* except *dumps()* can return multiline string for `dict`. This method normalizes that further by converting that multiline string to a single base64 encoded value.

Returns Base64 encoded binary data of input `data`

Return type binary

classmethod dumps(*data*)
Recursively dumps given data to SQRL response format.

Before data is dumped out, it is normalized by using *normalize()*.

This dumps each data type as follows:

Dict returns an `\r\n` multiline string. Each line is for a single key-pair of format `<key>=<dumped value>`.

List tilde (~) joined dumped list of values

Other no operation

classmethod normalize(*data*)
Recursively normalize data for encoding.

This encodes each data type as follows:

Dict returns an `OrderedDict` where all values are recursively normalized. Empty dict is normalized to empty string

List each value is recursively normalized

Binary Base64 encode data

Str no operation

Other data is casted to string using `__str__` (or `__unicode__`)

class `sqrl.utils.`**`QRGenerator`**(*url*)
Bases: `object`

Helper class for generating a QR image for the given SQRL url.

Parameters **`url`** (*`str`*) – URL for which to generate QR image

`generate_image`()
Generate QR image and get its binary data.

Returns Binary data of the png image file which can directly be returned to the user

Return type bytes

`sqrl.utils.`**`get_user_ip`**(*request*)
Utility function for getting user's IP from request address.

This either returns the IP address from the `request.REMOTE_ADDR` or `request.META['HTTP_X_REAL_IP']` when request might of been reverse proxied.

sqrl.views module

CHAPTER 2

Django SQRL

SQRL authentication support for Django

- Free software: MIT license
- GitHub: https://github.com/miki725/django-sqrl
- Documentation: https://django-sqrl.readthedocs.org.

Installing

SQRL Package

First step is to install `django-sqrl` which is easies to do using pip:

```
$ pip install django-sqrl
```

Django settings

Once installed there are a few required changes in Django settings:

1. Add `sqrl` to `INSTALLED_APPS`:

   ```
   INSTALLED_APPS = [
       ...
       'sqrl',
   ]
   ```

2. Make sure that some required Django apps are used:

   ```
   INSTALLED_APPS = [
       ...,
       'sqrl',
   ```

```
    'django.contrib.auth',
    'django.contrib.sessions',
    'django.contrib.staticfiles',
]
```

3. Make sure that some required Django middleware are used:

```
MIDDLEWARE_CLASSES = [
    ...
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
]
```

4. Change `AUTHENTICATION_BACKENDS` to use SQRL backend vs Django's `ModelBackend` (default):

```
AUTHENTICATION_BACKENDS = [
    'sqrl.backends.SQRLModelBackend',
]
```

5. If you are using Django admin, following are required:

   (a) Make sure that `sqrl` is listed before `admin` in the `INSTALLED_APPS`. This allows Django to prioritize `sqrl` templates since `django-sqrl` overwrites some of them.

```
INSTALLED_APPS = [
    ...,
    'sqrl',
    'django.contrib.admin',
    ...
]
```

   (b) Make sure to add a custom template directory in settings. `django-sqrl` extends Django admin's `base.html` which by default causes infinite recursion. To solve that, simply add a custom template directory which allows `django-sqrl` to explicitly extend from `django.contrib.admin` `base.html` template:

```
import os
import django
TEMPLATE_DIRS = [
    os.path.dirname(django.__file__),
]
```

URLs

All of SQRL functionality is enabled by adding its urls to the root url config:

```
url(r'^sqrl/', include(sqrl_urlpatterns, namespace='sqrl')),
```

If you use Django admin, then you should also want to add some SQRL urls to admin urls so that SQRL identity can be managed within Django admin:

```
from sqrl.views import AdminSiteSQRLIdentityManagementView
url(r'^admin/sqrl_manage/$', AdminSiteSQRLIdentityManagementView.as_view(), name=
↪'admin-sqrl_manage'),
url(r'^admin/', include(admin.site.urls)),
```

Templates

Now that SQRL is installed in a Django project you can use it in any login page with a simple template code:

```
{% sqrl as sqrl %}
<a href="{{ sqrl.sqrl_url }}">
    <img src="{% sqrl_qr_image_url sqrl %}">
</a>
{% sqrl_status_url_script_tag sqrl %}
<script src="{% static 'sqrl/sqrl.js' %}"></script>
```

The above template will add a QR image as a link which when used with SQRL client, will allow users to authenticate using SQRL.

If you would like to also add explicit button to trigger SQRL client on desktop applications, you can also use HTML form:

```
{% sqrl as sqrl %}
<form method="get" action="{{ sqrl.sqrl_url }}">
    {% sqrl_status_url_script_tag sqrl %}
    <a href="{{ sqrl.sqrl_url }}">
        <img src="{% sqrl_qr_image_url sqrl %}">
    </a>
    <input type="hidden" name="nut" value="{{ sqrl.nut.nonce }}">
    <input type="submit" value="Log in using SQRL">
</form>
{% sqrl_status_url_script_tag sqrl %}
<script src="{% static 'sqrl/sqrl.js' %}"></script>
```

Management Command

SQRL uses server state to keep track of open SQRL transactions in order to mitigate replay attacks. Since this state will constantly grow if not cleared, `django-sqrl` provides a helper management command to clear expired state:

```
$ python manage.py clearsqrlnuts
```

It is recommended to run this command as repeating task. Here is recommended cron config:

```
*/5 * * * * python manage.py clearsqrlnuts >/dev/null 2>&1
```

Testing

To run the tests you need to install testing requirements first:

```
$ make install
```

Then to run tests, you can use use Makefile command:

```
$ make test
```

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

D

E

G

H

I

N