Package 'vaultr'

May 16, 2019

Title Vault Client for Secrets and Sensitive Data

Version 1.0.2

Description Provides an interface to a 'HashiCorp' vault server over its http API (typically these are self-hosted; see <https://www.vaultproject.io>). This allows for secure storage and retrieval of secrets over a network, such as tokens, passwords and certificates. Authentication with vault is supported through several backends including user name/password and authentication via 'GitHub'.

License MIT + file LICENSE

URL https://github.com/vimc/vaultr

BugReports https://github.com/vimc/vaultr/issues

SystemRequirements vault

Imports R6, getPass, httr, jsonlite

Suggests knitr, mockery, processx, rmarkdown, testthat, withr

RoxygenNote 6.1.1

Encoding UTF-8

VignetteBuilder knitr

Language en-GB

NeedsCompilation no

Author Rich FitzJohn [aut, cre],
Robert Ashton [aut],
Wes Hinsley [aut],
Imperial College of Science, Technology and Medicine [cph]

Maintainer Rich FitzJohn <rich.fitzjohn@gmail.com>

Repository CRAN

Date/Publication 2019-05-16 13:30:03 UTC

R topics documented:

`vaultr` *Vault Client for Secrets and Sensitive Data*

Description

Vault client for secrets and sensitive data; this package provides wrappers for HashiCorp's vault server. The package wraps most of the high-level API, and includes support for authentication via a number of backends (tokens, username and password, github, and "AppRole"), as well as a number of secrets engines (two key-value stores, vault's cubbyhole and the transit backend for encryption-as-a-service).

Details

To get started, you might want to start with the "vaultr" vignette, available from the package with `vignette("vaultr")`.

The basic design of the package is that it has very few entrypoints - for most uses one will interact almost entirely with the `vault_client` function. That function returns an R6 object with several methods (functions) but also several objects that themselves contain more methods and objects, creating a nested tree of functionality.

From any object, online help is available via the help method, for example

```
client <- vaultr::vault_client()
client$secrets$transit$help()
```

For testing packages that rely on vault, there is support for creating temporary vault servers; see vault_test_server and the "packages" vignette.

---

vault_api_client *Vault Low-Level Client*

Description

Low-level API client. This can be used to directly communicate with the vault server. This object will primarily be useful for debugging, testing or developing new vault methods, but is nonetheless described here.

Methods

`is_authenticated` Test if the vault client currently holds a vault token. This method does not verify the token - only test that is present.
Usage:

```
is_authenticated()
```

`set_token` Set a token within the client
Usage:

```
set_token(token, verify = FALSE, quiet = FALSE)
```

Arguments:

- `token`: String, with the new vault client token
- `verify`: Logical, indicating if we should test that the token is valid. If `TRUE`, then we use `$verify_token()` to test the token before setting it and if it is not valid an error will be thrown and the token not set.
- `quiet`: Logical, if `TRUE`, then informational messages will be suppressed.

`verify_token` Test that a token is valid with the vault. This will call vault's `/sys/capabilities-self` endpoint with the token provided and check the `/sys` path.
Usage:

```
verify_token(token, quiet = TRUE)
```

Arguments:

- `token`: String, with the vault client token to test
- `quiet`: Logical, if `TRUE`, then informational messages will be suppressed.

`server_version` Retrieve the vault server version. This is by default cached within the client for a session. Will return an R `numeric_version` object.
Usage:

```
server_version(refresh = FALSE)
```

Arguments:

- `refresh`: Logical, indicating if the server version information should be refreshed even if known.

`GET` Send a `GET` request to the vault server

Usage:

```
GET(path, ...)
```

Arguments:

- `path`: The server path to use. This is the "interesting" part of the path only, with the server base url and api version information added.
- `...`: Additional `httr`-compatible options. These will be named parameters or `httr` "request" objects.

`LIST` Send a `LIST` request to the vault server

Usage:

```
LIST(path, ...)
```

Arguments:

- `path`: The server path to use. This is the "interesting" part of the path only, with the server base url and api version information added.
- `...`: Additional `httr`-compatible options. These will be named parameters or `httr` "request" objects.

`POST` Send a `POST` request to the vault server

Usage:

```
POST(path, ...)
```

Arguments:

- `path`: The server path to use. This is the "interesting" part of the path only, with the server base url and api version information added.
- `...`: Additional `httr`-compatible options. These will be named parameters or `httr` "request" objects.

`PUT` Send a `PUT` request to the vault server

Usage:

```
PUT(path, ...)
```

Arguments:

- `path`: The server path to use. This is the "interesting" part of the path only, with the server base url and api version information added.
- `...`: Additional `httr`-compatible options. These will be named parameters or `httr` "request" objects.

`DELETE` Send a `DELETE` request to the vault server

Usage:

```
DELETE(path, ...)
```

Arguments:

- `path`: The server path to use. This is the "interesting" part of the path only, with the server base url and api version information added.
- `...`: Additional `httr`-compatible options. These will be named parameters or `httr` "request" objects.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  # Ordinarily, we would use the "vault_client" object for
  # high-level access to the vault server
  client <- server$client()
  client$status()

  # The api() method returns the "api client" object:
  api <- client$api()
  api

  # This allows running arbitrary HTTP requests against the server:
  api$GET("/sys/seal-status")

  # this is how vaultr is internally implemented so anything can
  # be done here, for example following vault's API documentation
  # https://www.vaultproject.io/api/secret/kv/kv-v1.html#sample-request-2
  api$POST("/secret/mysecret", body = list(key = "value"))
  api$GET("/secret/mysecret")
  api$DELETE("/secret/mysecret")

  # cleanup
  server$kill()
}
```

---

vault_client *Make a vault client*

Description

Make a vault client. This must be done before accessing the vault. The default values for arguments are controlled by environment variables (see Details) and values provided as arguments override these defaults.

Usage

```
vault_client(login = FALSE, ..., addr = NULL, tls_config = NULL)
```

Arguments

login	Login method. Specify a string to be passed along as the `method` argument to `$login`. The default `FALSE` means not to login. `TRUE` means to login using a default method specified by the environment variable `VAULTR_AUTH_METHOD` - if that variable is not set, an error is thrown. The value of `NULL` is the same as `TRUE` but does not throw an error if `VAULTR_AUTH_METHOD` is not set. Supported methods are `token`, `github` and `userpass`.

`...` Additional arguments passed along to the authentication method indicated by `login`, if used.

`addr` The value address *including protocol and port*, e.g., `https://vault.example.com:8200`. If not given, the default is the environment variable `VAULT_ADDR`, which is the same as used by vault's command line client.

`tls_config` TLS (https) configuration. For most uses this can be left blank. However, if your vault server uses a self-signed certificate you will need to provide this. Defaults to the environment variable `VAULT_CAPATH`, which is the same as vault's command line client.

Environment variables

The creation of a client is affected by a number of environment variables, following the main vault command line client.

`VAULT_ADDR` The url of the vault server. Must include a protocol (most likely `https://` but in testing `http://` might be used)

`VAULT_CAPATH` The path to CA certificates

`VAULT_TOKEN` A vault token to use in authentication. Only used for token-based authentication

`VAULT_AUTH_GITHUB_TOKEN` As for the command line client, a github token for authentication using the github authentication backend

`VAULTR_AUTH_METHOD` The method to use for authentication

Methods

`api` Returns an api client object that can be used to directly interact with the vault server.
Usage:

```
api()
```

`read` Read a value from the vault. This can be used to read any value that you have permission to read, and can also be used as an interface to a version 1 key-value store (see `vault_client_kv1`. Similar to the vault CLI command `vault read`.
Usage:

```
read(path, field = NULL, metadata = FALSE)
```

Arguments:

- `path`: Path for the secret to read, such as `/secret/mysecret`
- `field`: Optional field to read from the secret. Each secret is stored as a key/value set (represented in R as a named list) and this is equivalent to using `[[field]]` on the return value. The default, `NULL`, returns the full set of values.
- `metadata`: Logical, indicating if we should return metadata for this secret (lease information etc) as an attribute along with the values itself. Ignored if `field` is specified.

`write` Write data into the vault. This can be used to write any value that you have permission to write, and can also be used as an interface to a version 1 key-value store (see `vault_client_kv1`. Similar to the vault CLI command `vault write`.
Usage:

```
write(path, data)
```

Arguments:

- `path`: Path for the secret to write, such as `/secret/mysecret`
- `data`: A named list of values to write into the vault at this path. This *replaces* any existing values.

`list` List data in the vault at a given path. This can be used to list keys, etc (e.g., at `/secret`).

Usage:

```
list(path, full_names = FALSE)
```

Arguments:

- `path`: The path to list
- `full_names`: Logical, indicating if full paths (relative to the vault root) should be returned.

Value: A character vector (of zero length if no keys are found). Paths that are "directories" (i.e., that contain keys and could themselves be listed) will be returned with a trailing forward slash, e.g. `path/`

`delete` Delete a value from the vault

Usage:

```
delete(path)
```

Arguments:

- `path`: The path to delete

`login` Login to the vault. This method is more complicated than most.

Usage:

```
login(..., method = "token", mount = NULL, renew = FALSE,
      quiet = FALSE, token_only = FALSE, use_cache = TRUE)
```

Arguments:

- `...`: Additional named parameters passed through to the underlying method
- `method`: Authentication method to use, as a string. Supported values include `token` (the default), `github`, `approle` and `userpass`.
- `mount`: The mount path for the authentication backend, *if it has been mounted in a non-standard location*. If not given, then it is assumed that the backend was mounted at a path corresponding to the method name.
- `renew`: Login, even if we appear to hold a valid token. If `FALSE` and we have a token then `login` does nothing.
- `quiet`: Suppress some informational messages
- `token_only`: Logical, indicating that we do not want to actually log in, but instead just generate a token and return that. IF given then `renew` is ignored and we always generate a new token.
- `use_cache`: Logical, indicating if we should look in the session cache for a token for this client. If this is `TRUE` then when we log in we save a copy of the token for this session and any subsequent calls to `login` at this vault address that use `use_cache = TRUE` will be able to use this token. Using cached tokens will make using some authentication backends that require authentication with external resources (e.g., `github`) much faster.

`status` Return the status of the vault server, including whether it is sealed or not, and the vault server version.
Usage:

```
status()
```

`unwrap` Returns the original response inside the given wrapping token. The vault endpoints used by this method perform validation checks on the token, returns the original value on the wire rather than a JSON string representation of it, and ensures that the response is properly audit-logged.
Usage:

```
unwrap(token)
```

Arguments:

- `token`: Specifies the wrapping token ID

`wrap_lookup` Look up properties of a wrapping token.
Usage:

```
wrap_lookup(token)
```

Arguments:

- `token`: Specifies the wrapping token ID to lookup

Author(s)

Rich FitzJohn

Examples

```
# We work with a test vault server here (see ?vault_test_server) for
# details.  To use it, you must have a vault binary installed on your
# system.  These examples will not affect any real running vault
# instance that you can connect to.
server <- vaultr::vault_test_server(if_disabled = message)

if (!is.null(server)) {
  # Create a vault_client object by providing the address of the vault
  # server.
  client <- vaultr::vault_client(addr = server$addr)

  # The client has many methods, grouped into a structure:
  client

  # For example, token related commands:
  client$token

  # The client is not authenticated by default:
  try(client$list("/secret"))

  # A few methods are unauthenticated and can still be run
```

```
  client$status()

  # Login to the vault, using the token that we know from the server -
  # ordinarily you would use a login approach suitable for your needs
  # (see the vault documentation).
  token <- server$token
  client$login(method = "token", token = token)

  # The vault contains no secrets at present
  client$list("/secret")

  # Secrets can contain any (reasonable) number of key-value pairs,
  # passed in as a list
  client$write("/secret/users/alice", list(password = "s3cret!"))

  # The whole list can be read out
  client$read("/secret/users/alice")
  # ...or just a field
  client$read("/secret/users/alice", "password")

  # Reading non-existant values returns NULL, not an error
  client$read("/secret/users/bob")

  client$delete("/secret/users/alice")
}
```

`vault_client_audit` *Vault Audit Devices*

Description

Interact with vault's audit devices. For more details, see https://www.vaultproject.io/docs/audit/

Methods

`list` List active audit devices. Returns a `data.frame` of names, paths and descriptions of active audit devices.
Usage:

```
list()
```

`enable` This endpoint enables a new audit device at the supplied path.
Usage:

```
enable(type, description = NULL, options = NULL, path = NULL)
```

Arguments:

- `type`: Name of the audit device to enable
- `description`: Human readable description for this audit device

- options: Options to configure the device with. These vary by device. This must be a named list of strings.
- path: Path to mount the audit device. By default, type is used as the path.

disable Disable an audit device

Usage:

```
disable(path)
```

Arguments:

- path: Path of the audit device to remove

hash The hash method is used to calculate the hash of the data used by an audit device's hash function and salt. This can be used to search audit logs for a hashed value when the original value is known.

Usage:

```
hash(input, device)
```

Arguments:

- input: The input string to hash
- device: The path of the audit device

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()
  # By default no audit engines are enabled with the testing server
  client$audit$list()

  # Create a file-based audit device on a temporary file:
  path <- tempfile()
  client$audit$enable("file", options = list(file_path = path))
  client$audit$list()

  # Generate some activity on the server:
  client$write("/secret/mysecret", list(key = "value"))

  # The audit logs contain details about the activity - see the
  # vault documentation for details in interpreting this
  readLines(path)

  # cleanup
  server$kill()
  unlink(path)
}
```

vault_client_auth *Vault Authentication Configuration*

Description

Interact with vault's authentication backends.

Methods

`backends` Return a character vector of supported authentication backends. If a backend `x` is present, then you can access it with `$auth$x`. Note that vault calls these authentication *methods* but we use *backends* here to differentiate with R6 methods. Note that these are backends supported by `vaultr` and not necessarily supported by the server - the server may not have enabled some of these backends, and may support other authentication backends not directly supported by vaultr. See the `$list()` method to query what the server supports.
Usage:

```
backends()
```

`list` List authentication backends supported by the vault server, including information about where these backends are mounted.
Usage:

```
list(detailed = FALSE)
```

Arguments:

- `detailed`: Logical, indicating if detailed information should be returned

`enable` Enable an authentication backend in the vault server.
Usage:

```
enable(type, description = NULL, local = FALSE, path = NULL)
```

Arguments:

- `type`: The type of authentication backend (e.g., `userpass`, `github`)
- `description`: Human-friendly description of the backend; will be returned by `$list()`
- `local`: Specifies if the auth method is local only. Local auth methods are not replicated nor (if a secondary) removed by replication.
- `path`: Specifies the path in which to enable the auth method. Defaults to be the same as `type`.

`disable` Disable an active authentication backend.
Usage:

```
disable(path)
```

Arguments:

- `path`: The path of the authentication backend to disable.

`token` Interact with vault's token authentication. See `vault_client_token` for more information.

`github` Interact with vault's GitHub authentication. See `vault_client_auth_github` for more information.

`userpass` Interact with vault's username/password based authentication. See `vault_client_auth_userpass` for more information.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  # List configured authentication backends
  client$auth$list()

  # cleanup
  server$kill()
}
```

`vault_client_auth_approle`

Vault AppRole Authentication Configuration

Description

Interact with vault's AppRole authentication backend. For more details about this, see the vault documentation at https://www.vaultproject.io/docs/auth/approle.html

Methods

`custom_mount` Set up a `vault_client_auth_approle` object at a custom mount. For example, suppose you mounted the `approle` authentication backend at `/approle-dev` you might use `ar <- vault$auth$approle2$custom_mount("/approle-dev")` - this pattern is repeated for other secret and authentication backends.
Usage:

```
custom_mount(mount)
```

Arguments:

- `mount`: String, indicating the path that the engine is mounted at.

`role_list` This endpoint returns a list the existing AppRoles in the method.
Usage:

```
role_list()
```

`role_write` Creates a new AppRole or updates an existing AppRole. This endpoint supports both create and update capabilities. There can be one or more constraints enabled on the role. It is required to have at least one of them enabled while creating or updating a role.
Usage:

```
role_write(role_name, bind_secret_id = NULL, secret_id_bound_cidrs = NULL,
      token_bound_cidrs = NULL, policies = NULL, secret_id_num_uses = NULL,
      secret_id_ttl = NULL, token_num_uses = NULL, token_ttl = NULL,
      token_max_ttl = NULL, period = NULL, enable_local_secret_ids = NULL,
      token_type = NULL)
```

Arguments:

- role_name: Name of the AppRole
- bind_secret_id: Require secret_id to be presented when logging in using this AppRole (boolean, default is TRUE).
- secret_id_bound_cidrs: Character vector of CIDR blocks; if set, specifies blocks of IP addresses which can perform the login operation.
- token_bound_cidrs: Character vector of if set, specifies blocks of IP addresses which can use the auth tokens generated by this role.
- policies: Character vector of policies set on tokens issued via this AppRole.
- secret_id_num_uses: Number of times any particular SecretID can be used to fetch a token from this AppRole, after which the SecretID will expire. A value of zero will allow unlimited uses.
- secret_id_ttl: Duration, after which any SecretID expires.
- token_num_uses: Number of times issued tokens can be used. A value of 0 means unlimited uses.
- token_ttl: Duration to set as the TTL for issued tokens and at renewal time.
- token_max_ttl: Duration, after which the issued token can no longer be renewed.
- period: A duration; when set, the token generated using this AppRole is a periodic token; so long as it is renewed it never expires, but the TTL set on the token at each renewal is fixed to the value specified here. If this value is modified, the token will pick up the new value at its next renewal.
- enable_local_secret_ids: Boolean, if TRUE, then the secret IDs generated using this role will be cluster local. This can only be set during role creation and once set, it can't be reset later.
- token_type: The type of token that should be generated via this role. Can be service, batch, or default to use the mount's default (which unless changed will be service tokens).

role_read Reads the properties of an existing AppRole.

Usage:

```
role_read(role_name)
```

Arguments:

- role_name: Name of the AppRole

role_delete Deletes an existing AppRole from the method.

Usage:

```
role_delete(role_name)
```

Arguments:

- role_name: Name of the AppRole to delete

role_id_read Reads the RoleID of an existing AppRole.

Usage:

```
role_id_read(role_name)
```

Arguments:

- role_name: Name of the AppRole

`role_id_write` Updates the RoleID of an existing AppRole to a custom value.

Usage:

```
role_id_write(role_name, role_id)
```

Arguments:

- `role_name`: Name of the AppRole (string)
- `role_id`: Value to be set as RoleID (string)

`secret_id_generate` Generates and issues a new SecretID on an existing AppRole. Similar to tokens, the response will also contain a `secret_id_accessor` value which can be used to read the properties of the SecretID without divulging the SecretID itself, and also to delete the SecretID from the AppRole.

Usage:

```
secret_id_generate(role_name, metadata = NULL, cidr_list = NULL, token_bound_cidrs = NULL)
```

Arguments:

- `role_name`: Name of the AppRole.
- `metadata`: Metadata to be tied to the SecretID. This should be a named list of key-value pairs. This metadata will be set on tokens issued with this SecretID, and is logged in audit logs in plaintext.
- `cidr_list`: Character vector CIDR blocks enforcing secret IDs to be used from specific set of IP addresses. If `bound_cidr_list` is set on the role, then the list of CIDR blocks listed here should be a subset of the CIDR blocks listed on the role.
- `token_bound_cidrs`: Character vector of CIDR blocks; if set, specifies blocks of IP addresses which can use the auth tokens generated by this SecretID. Overrides any role-set value but must be a subset.

`secret_id_list` Lists the accessors of all the SecretIDs issued against the AppRole. This includes the accessors for "custom" SecretIDs as well.

Usage:

```
secret_id_list(role_name)
```

Arguments:

- `role_name`: Name of the AppRole

`secret_id_read` Reads out the properties of a SecretID.

Usage:

```
secret_id_read(role_name, secret_id, accessor = FALSE)
```

Arguments:

- `role_name`: Name of the AppRole
- `secret_id`: Secret ID attached to the role
- `accessor`: Logical, if `TRUE`, treat `secret_id` as an accessor rather than a secret id.

`secret_id_delete` Delete an AppRole secret ID

Usage:

```
secret_id_delete(role_name, secret_id, accessor = FALSE)
```

Arguments:

- role_name: Name of the AppRole
- secret_id: Secret ID attached to the role
- accessor: Logical, if TRUE, treat secret_id as an accessor rather than a secret id.

login Log into the vault using AppRole authentication. Normally you would not call this directly but instead use $login with method = "approle" and proving the role_id and secret_id arguments. This function returns a vault token but does not set it as the client token.
Usage:

```
login(role_id, secret_id)
```

Arguments:

- role_id: RoleID of the AppRole
- secret_id: SecretID belonging to AppRole

Examples

```
vaultr::vault_client(addr = "https://localhost:8200")$auth$approle
```

---

vault_client_auth_github
Vault GitHub Authentication Configuration

Description

Interact with vault's GitHub authentication backend. For more details, please see the vault documentation at https://www.vaultproject.io/docs/auth/github.html

Methods

custom_mount Set up a vault_client_auth_github object at a custom mount. For example, suppose you mounted the github authentication backend at /github-myorg you might use gh <- vault$auth$github2$custom_mount("/github-myorg") - this pattern is repeated for other secret and authentication backends.
Usage:

```
custom_mount(mount)
```

Arguments:

- mount: String, indicating the path that the engine is mounted at.

configure Configures the connection parameters for GitHub-based authentication.
Usage:

```
configure(organization, base_url = NULL, ttl = NULL, max_ttl = NULL)
```

Arguments:

- organization: The organization users must be part of (note American spelling).
- base_url: The API endpoint to use. Useful if you are running GitHub Enterprise or an API-compatible authentication server.

- `ttl`: Duration after which authentication will be expired
- `max_ttl`: Maximum duration after which authentication will be expired

`configuration` Reads the connection parameters for GitHub-based authentication.

Usage:

```
configuration()
```

`write` Write a mapping between a GitHub team or user and a set of vault policies.

Usage:

```
write(team_name, policies, user = FALSE)
```

Arguments:

- `team_name`: String, with the GitHub team name
- `policies`: A character vector of vault policies that this user or team will have for vault access if they match this team or user.
- `user`: Scalar logical - if `TRUE`, then `team_name` is interpreted as a *user* instead.

`read` Write a mapping between a GitHub team or user and a set of vault policies.

Usage:

```
read(team_name, user = FALSE)
```

Arguments:

- `team_name`: String, with the GitHub team name
- `user`: Scalar logical - if `TRUE`, then `team_name` is interpreted as a *user* instead.

`login` Log into the vault using GitHub authentication. Normally you would not call this directly but instead use `$login` with `method = "github"` and proving the `token` argument. This function returns a vault token but does not set it as the client token.

Usage:

```
login(token = NULL)
```

Arguments:

- `token`: A GitHub token to authenticate with.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  client$auth$enable("github")
  # To enable login for members of the organisation "vimc":
  client$auth$github$configure(organization = "vimc")
  # To map members of the "robots" team *within* that organisation
  # to the "defaut" policy:
  client$auth$github$write("development", "default")

  # Once configured like this, if we have a PAT for a member of
  # the "development" team saved as an environment variable
  # "VAULT_AUTH_GITHUB_TOKEN" then doing
  #
```

```
  #   vaultr::vault_client(addr = ..., login = "github")
  #
  # will contact GitHub to verify the user token and vault will
  # then issue a client token

  # cleanup
  server$kill()
}
```

`vault_client_auth_userpass` *Vault Username/Password Authentication Configuration*

Description

Interact with vault's username/password authentication backend. This backend can be used to configure basic username+password authentication, suitable for human users. For more information, please see the vault documentation https://www.vaultproject.io/docs/auth/userpass.html

Methods

`custom_mount` Set up a `vault_client_auth_userpass` object at a custom mount. For example, suppose you mounted the `userpass` authentication backend at `/userpass2` you might use `up <- vault$auth$userpass2$custom_mount("/userpass2")` - this pattern is repeated for other secret and authentication backends.

Usage:

```
custom_mount(mount)
```

Arguments:

- `mount`: String, indicating the path that the engine is mounted at.

`write` Create or update a user.

Usage:

```
write(username, password = NULL, policies = NULL, ttl = NULL,
      max_ttl = NULL, bound_cidrs = NULL)
```

Arguments:

- `username`: Username for the user
- `password`: Password for the user (required when creating a user only)
- `policies`: Character vector of policies for the user
- `ttl`: The lease duration which decides login expiration
- `max_ttl`: Maximum duration after which login should expire
- `bound_cidrs`: Character vector of CIDRs. If set, restricts usage of the login and token to client IPs falling within the range of the specified CIDR(s).

`read` Reads the properties of an existing username.

Usage:

```
read(username)
```

Arguments:

- `username`: Username to read

`delete` Delete a user

Usage:

```
delete(username)
```

Arguments:

- `username`: Username to delete

`update_password` Update password for a user

Usage:

```
update_password(username, password)
```

Arguments:

- `username`: Username for the user to update
- `password`: New password for the user

`update_policies` Update vault policies for a user

Usage:

```
update_policies(username, policies)
```

Arguments:

- `username`: Username for the user to update
- `policies`: Character vector of policies for this user

`list` List users known to vault

Usage:

```
list()
```

`login` Log into the vault using username/password authentication. Normally you would not call this directly but instead use `$login` with `method = "userpass"` and proving the `username` argument and optionally the `password` argument. This function returns a vault token but does not set it as the client token.

Usage:

```
login(username, password = NULL)
```

Arguments:

- `username`: Username to authenticate with
- `password`: Password to authenticate with. If omitted or `NULL` and the session is interactive, the password will be prompted for.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  root <- server$client()

  # The userpass authentication backend is not enabled by default,
```

```
  # so we need to enable it first
  root$auth$enable("userpass")

  # Then we can add users:
  root$auth$userpass$write("alice", "p4ssw0rd")

  # Create a new client and login with this user:
  alice <- vaultr::vault_client(addr = server$addr)
  # it is not recommended to login with the password like this as
  # it will end up in the command history, but in interactive use
  # you will be prompted securely for password
  alice$login(method = "userpass",
              username = "alice", password = "p4ssw0rd")
  # Alice has now logged in and has only "default" policies
  alice$auth$token$lookup_self()$policies

  # (wheras our original root user has the "root" policy)
  root$auth$token$lookup_self()$policies
}
```

`vault_client_cubbyhole` *Cubbyhole secret store*

Description

Interact with vault's cubbyhole key-value store. This is useful for storing simple key-value data without versioning or metadata (c.f. `vault_client_kv2`) that is scoped to your current token only and not accessible to anyone else. For more details please see the vault documentation https://www.vaultproject.io/docs/secrets/cubbyhole/index.html

Methods

`read` Read a value from your cubbyhole

Usage:

```
read(path, field = NULL, metadata = FALSE)
```

Arguments:

- `path`: Path for the secret to read, such as `/cubbyhole/mysecret`
- `field`: Optional field to read from the secret. Each secret is stored as a key/value set (represented in R as a named list) and this is equivalent to using `[[field]]` on the return value. The default, `NULL`, returns the full set of values.
- `metadata`: Logical, indicating if we should return metadata for this secret (lease information etc) as an attribute along with the values itself. Ignored if `field` is specified.

`write` Write data into your cubbyhole.

Usage:

```
write(path, data)
```

Arguments:

- `path`: Path for the secret to write, such as `/cubbyhole/mysecret`
- `data`: A named list of values to write into the vault at this path. This *replaces* any existing values.

`list` List data in the vault at a give path. This can be used to list keys, etc (e.g., at `/cubbyhole`).

Usage:

```
list(path, full_names = FALSE)
```

Arguments:

- `path`: The path to list
- `full_names`: Logical, indicating if full paths (relative to the vault root) should be returned.

Value: A character vector (of zero length if no keys are found). Paths that are "directories" (i.e., that contain keys and could themselves be listed) will be returned with a trailing forward slash, e.g. `path/`

`delete` Delete a value from the vault

Usage:

```
delete(path)
```

Arguments:

- `path`: The path to delete

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  # Shorter path for easier reading:
  cubbyhole <- client$secrets$cubbyhole
  cubbyhole

  # Write a value
  cubbyhole$write("cubbyhole/secret", list(key = "value"))
  # List it
  cubbyhole$list("cubbyhole")
  # Read it
  cubbyhole$read("cubbyhole/secret")
  # Delete it
  cubbyhole$delete("cubbyhole/secret")

  # cleanup
  server$kill()
}
```

`vault_client_kv1` *Key-Value Store (Version 1)*

Description

Interact with vault's version 1 key-value store. This is useful for storing simple key-value data without versioning or metadata (see `vault_client_kv2` for a richer key-value store).

Details

Up to vault version 0.12.0 this was mounted by default at `/secret`. It can be accessed from vault with either the `$read`, `$write`, `$list` and `$delete` methods on the main `vault_client` object or by the `$kv1` member of the `secrets` member of the main vault client.

Methods

`read` Read a value from the vault. This can be used to read any value that you have permission to read in this store.
Usage:

```
read(path, field = NULL, metadata = FALSE)
```

Arguments:

- `path`: Path for the secret to read, such as `/secret/mysecret`
- `field`: Optional field to read from the secret. Each secret is stored as a key/value set (represented in R as a named list) and this is equivalent to using `[[field]]` on the return value. The default, `NULL`, returns the full set of values.
- `metadata`: Logical, indicating if we should return metadata for this secret (lease information etc) as an attribute along with the values itself. Ignored if `field` is specified.

`write` Write data into the vault. This can be used to write any value that you have permission to write in this store.
Usage:

```
write(path, data)
```

Arguments:

- `path`: Path for the secret to write, such as `/secret/mysecret`
- `data`: A named list of values to write into the vault at this path. This *replaces* any existing values.

`list` List data in the vault at a give path. This can be used to list keys, etc (e.g., at `/secret`).
Usage:

```
list(path, full_names = FALSE)
```

Arguments:

- `path`: The path to list
- `full_names`: Logical, indicating if full paths (relative to the vault root) should be returned.

Value: A character vector (of zero length if no keys are found). Paths that are "directories" (i.e., that contain keys and could themselves be listed) will be returned with a trailing forward slash, e.g. `path/`

`delete` Delete a value from the vault

Usage:

```
delete(path)
```

Arguments:

- `path`: The path to delete

`custom_mount` Set up a `vault_client_kv1` object at a custom mount. For example, suppose you mounted another copy of the `kv1` secret backend at `/secret2` you might use `kv <- vault$secrets$kv1$custom_moun` - this pattern is repeated for other secret and authentication backends.

Usage:

```
custom_mount(mount)
```

Arguments:

- `mount`: String, indicating the path that the engine is mounted at.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  # Write secrets
  client$secrets$kv1$write("/secret/path/mysecret", list(key = "value"))

  # List secrets - note the trailing "/" indicates a folder
  client$secrets$kv1$list("/secret")
  client$secrets$kv1$list("/secret/path")

  # Read secrets
  client$secrets$kv1$read("/secret/path/mysecret")
  client$secrets$kv1$read("/secret/path/mysecret", field = "key")

  # Delete secrets
  client$secrets$kv1$delete("/secret/path/mysecret")
  client$secrets$kv1$read("/secret/path/mysecret")

  # cleanup
  server$kill()
}
```

vault_client_kv2 *Key-Value Store (Version 2)*

Description

Interact with vault's version 2 key-value store. This is useful for storing simple key-value data that can be versioned and for storing metadata alongside the secrets (see vault_client_kv1 for a simpler key-value store, and see https://www.vaultproject.io/docs/secrets/kv/kv-v2.html for detailed information about this secret store.

Details

A kv2 store can be mounted anywhere, so all methods accept a mount argument. This is different to the CLI which lets you try and read values from any vault path, but similar to other secret and auth backends which accept arguments like -mount-point. So if the kv2 store is mounted at /project-secrets for example, with a vault client vault one could write

```
vault$secrets$kv2$get("/project-secrets/mysecret",
                      mount = "project-secrets")
```

or

```
kv2 <- vault$secrets$kv2$custom_mount("project-secrets")
kv2$get("mysecret")
```

If the leading part of of a path to secret within a kv2 store does not match the mount point, vaultr will throw an error. This approach results in more predictable error messages, though it is a little more typing than for the CLI vault client.

Methods

config Fetch the configuration for this kv2 store. Returns a named list of values, the contents of which will depend on the vault version.
Usage:

```
config(mount = NULL)
```

Arguments:

- mount: Custom mount path to use for this store (see Details.

custom_mount Set up a vault_client_kv2 object at a custom mount. For example, suppose you mounted another copy of the kv2 secret backend at /secret2 you might use kv <- vault$secrets$kv2$custom_moun - this pattern is repeated for other secret and authentication backends.
Usage:

```
custom_mount(mount)
```

Arguments:

- mount: String, indicating the path that the engine is mounted at.

`delete` Delete a secret from the vault. This marks the version as deleted and will stop it from being returned from reads, but the underlying data will not be removed. A delete can be undone using the undelete method.
Usage:

```
delete(path, version = NULL, mount = NULL)
```

Arguments:

- `path`: Path to delete
- `version`: Optional version to delete. If `NULL` (the default) then the latest version of the secret is deleted. Otherwise, `version` can be a vector of integer versions to delete.
- `mount`: Custom mount path to use for this store (see `Details`.

`destroy` Delete a secret entirely. Unlike `delete` this operation is irreversible and is more like the `delete` operation on `vault_client_kv1` stores.
Usage:

```
destroy(path, version, mount = NULL)
```

Arguments:

- `path`: Path to delete
- `version`: Version numbers to delete, as a vector of integers (this is required)
- `mount`: Custom mount path to use for this store (see `Details`.

`get` Read a secret from the vault
Usage:

```
get(path, version = NULL, field = NULL, metadata = FALSE,
      mount = NULL)
```

Arguments:

- `path`: Path of the secret to read
- `version`: Optional version of the secret to read. If `NULL` (the default) then the most recent version is read. Otherwise this must be a scalar integer.
- `field`: Optional field to read from the secret. Each secret is stored as a key/value set (represented in R as a named list) and this is equivalent to using `[[field]]` on the return value. The default, `NULL`, returns the full set of values.
- `metadata`: Logical, indicating if we should return metadata for this secret (lease information etc) as an attribute along with the values itself. Ignored if `field` is specified.
- `mount`: Custom mount path to use for this store (see `Details`.

`list` List data in the vault at a give path. This can be used to list keys, etc (e.g., at `/secret`).
Usage:

```
list(path, full_names = FALSE, mount = NULL)
```

Arguments:

- `path`: The path to list
- `full_names`: Logical, indicating if full paths (relative to the vault root) should be returned.
- `mount`: Custom mount path to use for this store (see `Details`.

Value: A character vector (of zero length if no keys are found). Paths that are "directories" (i.e., that contain keys and could themselves be listed) will be returned with a trailing forward slash, e.g. `path/`

`metadata_get` Read secret metadata and versions at the specified path
Usage:

```
metadata_get(path, mount = NULL)
```

Arguments:

- `path`: Path of secret to read metadata for
- `mount`: Custom mount path to use for this store (see `Details`.

`metadata_put` Update metadata for a secret. This is allowed even if a secret does not yet exist, though this requires the `create` vault permission at this path.
Usage:

```
metadata_put(path, cas_required = NULL, max_versions = NULL, mount = NULL)
```

Arguments:

- `path`: Path of secret to update metadata for
- `cas_required`: Logical, indicating that if If true the key will require the cas parameter to be set on all write requests (see `put`). If `FALSE`, the backends configuration will be used.
- `max_versions`: Integer, indicating the maximum number of versions to keep per key. If not set, the backend's configured max version is used. Once a key has more than the configured allowed versions the oldest version will be permanently deleted.
- `mount`: Custom mount path to use for this store (see `Details`.

`metadata_delete` This method permanently deletes the key metadata and all version data for the specified key. All version history will be removed.
Usage:

```
metadata_delete(path, mount = NULL)
```

Arguments:

- `path`: Path to delete
- `mount`: Custom mount path to use for this store (see `Details`.

`put` Create or update a secret in this store.
Usage:

```
put(path, data, cas = NULL, mount = NULL)
```

Arguments:

- `path`: Path for the secret to write, such as `/secret/mysecret`
- `data`: A named list of values to write into the vault at this path.
- `cas`: Integer, indicating the "cas" value to use a "Check-And-Set" operation. If not set the write will be allowed. If set to 0 a write will only be allowed if the key doesn't exist. If the index is non-zero the write will only be allowed if the key's current version matches the version specified in the cas parameter.
- `mount`: Custom mount path to use for this store (see `Details`.

`undelete` Undeletes the data for the provided version and path in the key-value store. This restores the data, allowing it to be returned on get requests. This works with data deleted with `$delete` but not with `$destroy`.
Usage:

```
undelete(path, version, mount = NULL)
```

Arguments:

- `path`: The path to undelete
- `version`: Integer vector of versions to undelete
- `mount`: Custom mount path to use for this store (see `Details`.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()
  # With the test server as created by vaultr, the kv2 storage
  # engine is not enabled.  To use the kv2 store we must first
  # enable it; the command below will add it at the path /kv on
  # our vault server
  client$secrets$enable("kv", version = 2)

  # For ease of reading, create a 'kv' object for interacting with
  # the store (see below for the calls without this object)
  kv <- client$secrets$kv2$custom_mount("kv")
  kv$config()

  # The version-2 kv store can be treated largely the same as the
  # version-1 store, though with slightly different command names
  # (put instead of write, get instead of read)
  kv$put("/kv/path/secret", list(key = "value"))
  kv$get("/kv/path/secret")

  # But it also allows different versions to be stored at the same path:
  kv$put("/kv/path/secret", list(key = "s3cret!"))
  kv$get("/kv/path/secret")

  # Old versions can be retrieved still:
  kv$get("/kv/path/secret", version = 1)

  # And metadata about versions can be retrieved
  kv$metadata_get("/kv/path/secret")

  # cleanup
  server$kill()
}
```

vault_client_operator *Vault Administration*

Description

Administration commands for vault operators. Very few of these commands should be used without consulting the vault documentation as they affect the administration of a vault server, but they are included here for completeness.

Methods

`key_status` Return information about the current encryption key of Vault.
Usage:

```
key_status()
```

`is_initialized` Returns the initialization status of Vault
Usage:

```
is_initialized()
```

`init` This endpoint initializes a new Vault. The Vault must not have been previously initialized.
Usage:

```
init(secret_shares, secret_threshold)
```

Arguments:

- `secret_shares`: Integer, specifying the number of shares to split the master key into
- `secret_threshold`: Integer, specifying the number of shares required to reconstruct the master key. This must be less than or equal secret_shares

`leader_status` Check the high availability status and current leader of Vault
Usage:

```
leader_status()
```

`rekey_status` Reads the configuration and progress of the current rekey attempt
Usage:

```
rekey_status()
```

`rekey_start` This method begins a new rekey attempt. Only a single rekey attempt can take place at a time, and changing the parameters of a rekey requires cancelling and starting a new rekey, which will also provide a new nonce.
Usage:

```
rekey_start(secret_shares, secret_threshold)
```

Arguments:

- `secret_shares`: Integer, specifying the number of shares to split the master key into
- `secret_threshold`: Integer, specifying the number of shares required to reconstruct the master key. This must be less than or equal secret_shares

`rekey_cancel` This method cancels any in-progress rekey. This clears the rekey settings as well as any progress made. This must be called to change the parameters of the rekey. Note: verification is still a part of a rekey. If rekeying is cancelled during the verification flow, the current unseal keys remain valid.
Usage:

```
rekey_cancel()
```

`rekey_submit` This method is used to enter a single master key share to progress the rekey of the Vault. If the threshold number of master key shares is reached, Vault will complete the rekey. Otherwise, this method must be called multiple times until that threshold is met. The rekey nonce operation must be provided with each call.
Usage:

```
rekey_submit(key, nonce)
```

Arguments:

- `key`: Specifies a single master share key (a string)
- `nonce`: Specifies the nonce of the rekey operation (a string)

`rotate` This method triggers a rotation of the backend encryption key. This is the key that is used to encrypt data written to the storage backend, and is not provided to operators. This operation is done online. Future values are encrypted with the new key, while old values are decrypted with previous encryption keys.
Usage:

```
rotate()
```

`seal` Seal the vault, preventing any access to it. After the vault is sealed, it must be unsealed for further use.
Usage:

```
seal()
```

`seal_status` Check the seal status of a Vault. This method can be used even when the client is not authenticated with the vault (which will the case for a sealed vault).
Usage:

```
seal_status()
```

`unseal` Submit a portion of a key to unseal the vault. This method is typically called by multiple different operators to assemble the master key.
Usage:

```
unseal(key, reset = FALSE)
```

Arguments:

- `key`: The master key share
- `reset`: Logical, indicating if the unseal process should start be started again.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
```

```
  client <- server$client()

  # Our test server is by default unsealed:
  client$status()$sealed

  # We can seal the vault to prevent all access:
  client$operator$seal()
  client$status()$sealed

  # And then unseal it again
  client$operator$unseal(server$keys)
  client$status()$sealed
}
```

`vault_client_policy` *Vault Policy Configuration*

Description

Interact with vault's policies. To get started, you may want to read up on policies as described in the vault manual, here: https://www.vaultproject.io/docs/concepts/policies.html

Methods

`delete` This endpoint deletes the policy with the given name. This will immediately affect all users associated with this policy.
Usage:

```
delete(name)
```

Arguments:

- `name`: Specifies the name of the policy to delete.

`list` Lists all configured policies.
Usage:

```
list()
```

`read` Retrieve the policy body for the named policy
Usage:

```
read(name)
```

Arguments:

- `name`: Specifies the name of the policy to retrieve

`write` Create or update a policy. Once a policy is updated, it takes effect immediately to all associated users.
Usage:

```
write(name, rules)
```

Arguments:

- `name`: Name of the policy to update
- `rules`: Specifies the policy document. This is a string in HashiCorp configuration language. At present this must be read in as a single string (not a character vector of strings); future versions of vaultr may allow more flexible specification such as `@filename`.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  # The test server starts with only the policies "root" (do
  # everything) and "default" (do nothing).
  client$policy$list()

  # Here let's make a policy that allows reading secrets from the
  # path /secret/develop/* but nothing else
  rules <- 'path "secret/develop/*" {policy = "read"}'
  client$policy$write("read-secret-develop", rules)

  # Our new rule is listed and can be read
  client$policy$list()
  client$policy$read("read-secret-develop")

  # For testing, let's create a secret under this path, and under
  # a different path:
  client$write("/secret/develop/password", list(value = "password"))
  client$write("/secret/production/password", list(value = "k2e89be@rdC#"))

  # Create a token that can use this policy:
  token <- client$auth$token$create(policies = "read-secret-develop")

  # Login to the vault using this token:
  alice <- vaultr::vault_client(addr = server$addr,
                                login = "token", token = token)

  # We can read the paths that we have been granted access to:
  alice$read("/secret/develop/password")

  # We can't read secrets that are outside our path:
  try(alice$read("/secret/production/password"))

  # And we can't write:
  try(alice$write("/secret/develop/password", list(value = "secret")))

  # cleanup
  server$kill()
}
```

vault_client_secrets *Vault Secret Configuration*

Description

Interact with vault's secret backends.

Methods

`enable` Enable a secret backend in the vault server

Usage:

```
enable(type, path = type, description = NULL, version = NULL)
```

Arguments:

- `type`: The type of secret backend (e.g., `transit`, `kv`).
- `path`: Specifies the path in which to enable the auth method. Defaults to be the same as `type`.
- `description`: Human-friendly description of the backend; will be returned by `$list()`
- `version`: Used only for the `kv` backend, where an integer is used to select between `vault_client_kv1` and `vault_client_kv2` engines.

`disable` Disable a previously-enabled secret engine

Usage:

```
disable(path)
```

Arguments:

- `path`: Path of the secret engine

`list` List enabled secret engines

Usage:

```
list(detailed = FALSE)
```

Arguments:

- `detailed`: Logical, indicating if detailed output is wanted.

`move` Move the path that a secret engine is mounted at

Usage:

```
move(from, to)
```

Arguments:

- `from`: Original path
- `to`: New path

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  # To remove the default version 1 kv store and replace with a
  # version 2 store:
  client$secrets$disable("/secret")
  client$secrets$enable("kv", "/secret", version = 2)

  # cleanup
  server$kill()
}
```

`vault_client_token` *Vault Tokens*

Description

Interact with vault's token methods. This includes support for querying, creating and deleting tokens. Tokens are fundamental to the way that vault works, so there are a lot of methods here. The vault documentation has a page devoted to token concepts and another with commands that have names very similar to the names used here.

Token Accessors

Many of the methods use "token accessors" - whenever a token is created, an "accessor" is created at the same time. This is another token that can be used to perform limited actions with the token such as

- Look up a token's properties (not including the actual token ID)
- Look up a token's capabilities on a path
- Revoke the token

However, accessors cannot be used to login, nor to retrieve the actual token itself.

Methods

`list` List token accessors, returning a character vector
Usage:

```
list()
```

`capabilities` Fetch the capabilities of a token on the given paths. The capabilities returned will be derived from the policies that are on the token, and from the policies to which the token is entitled to through the entity and entity's group memberships.
Usage:

```
capabilities(path, token)
```

Arguments:

- `path`: Vector of paths on which capabilities are being queried
- `token`: Single token for which capabilities are being queried

`capabilities_self` As for the `capabilities` method, but for the client token used to make the request.
Usage:

```
capabilities_self(path)
```

Arguments:

- `path`: Vector of paths on which capabilities are being queried

`capabilities_accessor` As for the `capabilities` method, but using a token *accessor* rather than a token itself.
Usage:

```
capabilities_accessor(path, accessor)
```

Arguments:

- `path`: Vector of paths on which capabilities are being queried
- `accessor`: Accessor of the token for which capabilities are being queried

`client` Return the current client token
Usage:

```
client()
```

`create` Create a new token
Usage:

```
create(role_name = NULL, id = NULL, policies = NULL, meta = NULL,
      orphan = FALSE, no_default_policy = FALSE, max_ttl = NULL,
      display_name = NULL, num_uses = 0L, period = NULL, ttl = NULL,
      wrap_ttl = NULL)
```

Arguments:

- `role_name`: The name of the token role
- `id`: The ID of the client token. Can only be specified by a root token. Otherwise, the token ID is a randomly generated value
- `policies`: A character vector of policies for the token. This must be a subset of the policies belonging to the token making the request, unless root. If not specified, defaults to all the policies of the calling token.
- `meta`: A named list of strings as metadata to pass through to audit devices.
- `orphan`: Logical, indicating if the token created should be an orphan (they will have no parent). As such, they will not be automatically revoked by the revocation of any other token.
- `no_default_policy`: Logical, if `TRUE`, then the default policy will not be contained in this token's policy set.

- `max_ttl`: Provides a maximum lifetime for any tokens issued against this role, including periodic tokens. Unlike direct token creation, where the value for an explicit max TTL is stored in the token, for roles this check will always use the current value set in the role. The main use of this is to provide a hard upper bound on periodic tokens, which otherwise can live forever as long as they are renewed. This is an integer number of seconds
- `display_name`: The display name of the token
- `num_uses`: Maximum number of uses that a token can have. This can be used to create a one-time-token or limited use token. The default, or the value of 0, has no limit to the number of uses.
- `period`: If specified, the token will be periodic; it will have no maximum TTL (unless a `max_ttl` is also set) but every renewal will use the given period. Requires a root/sudo token to use.
- `ttl`: The TTL period of the token, provided as "1h", where hour is the largest suffix. If not provided, the token is valid for the default lease TTL, or indefinitely if the root policy is used.
- `wrap_ttl`: Indicates that the secret should be wrapped. This is discussed in the vault documentation. When this option is used, `vault` will take the response it would have sent to an HTTP client and instead insert it into the cubbyhole of a single-use token, returning that single-use token instead. Logically speaking, the response is wrapped by the token, and retrieving it requires an unwrap operation against this token (see the `$unwrap` method `vault_client`). Must be specified as a valid duration (e.g., `1h`).

`lookup` Returns information about the client token

Usage:

```
lookup(token = NULL)
```

Arguments:

- `token`: The token to lookup

`lookup_self` Returns information about the current client token (as if calling `$lookup` with the token the client is using.

Usage:

```
lookup_self()
```

`lookup_accessor` Returns information about the client token from the accessor.

Usage:

```
lookup_accessor(accessor)
```

Arguments:

- `accessor`: The token accessor to lookup

`renew` Renews a lease associated with a token. This is used to prevent the expiration of a token, and the automatic revocation of it. Token renewal is possible only if there is a lease associated with it.

Usage:

```
renew(token, increment = NULL)
```

Arguments:

- `token`: The token to renew

- `increment`: An optional requested lease increment can be provided. This increment may be ignored. If given, it should be a duration (e.g., `1h`).

`renew_self` Renews a lease associated with the calling token. This is used to prevent the expiration of a token, and the automatic revocation of it. Token renewal is possible only if there is a lease associated with it. This is equivalent to calling `$renew()` with the client token.
Usage:

```
renew_self(increment = NULL)
```

Arguments:

- `increment`: An optional requested lease increment can be provided. This increment may be ignored. If given, it should be a duration (e.g., `1h`).

`revoke` Revokes a token and all child tokens. When the token is revoked, all dynamic secrets generated with it are also revoked.
Usage:

```
revoke(token)
```

Arguments:

- `token`: The token to revoke

`revoke_self` Revokes the token used to call it and all child tokens. When the token is revoked, all dynamic secrets generated with it are also revoked. This is equivalent to calling `$revoke()` with the client token.
Usage:

```
revoke_self()
```

`revoke_accessor` Revoke the token associated with the accessor and all the child tokens. This is meant for purposes where there is no access to token ID but there is need to revoke a token and its children.
Usage:

```
revoke_accessor(accessor)
```

Arguments:

- `accessor`: Accessor of the token to revoke.

`revoke_and_orphan` Revokes a token but not its child tokens. When the token is revoked, all secrets generated with it are also revoked. All child tokens are orphaned, but can be revoked subsequently using /auth/token/revoke/. This is a root-protected method.
Usage:

```
revoke_and_orphan(token)
```

Arguments:

- `token`: The token to revoke

`role_read` Fetches the named role configuration.
Usage:

```
role_read(role_name)
```

Arguments:

- `role_name`: The name of the token role.

`role_list` List available token roles.

Usage:

```
role_list()
```

`role_write` Creates (or replaces) the named role. Roles enforce specific behaviour when creating tokens that allow token functionality that is otherwise not available or would require sudo/root privileges to access. Role parameters, when set, override any provided options to the create endpoints. The role name is also included in the token path, allowing all tokens created against a role to be revoked using the `/sys/leases/revoke-prefix` endpoint.

Usage:

```
role_write(role_name, allowed_policies = NULL, disallowed_policies = NULL,
     orphan = NULL, period = NULL, renewable = NULL, explicit_max_ttl = NULL,
      path_suffix = NULL, bound_cidrs = NULL, token_type = NULL)
```

Arguments:

- `role_name`: Name for the role - this will be used later to refer to the role (e.g., in `$create` and other `$role_*` methods.
- `allowed_policies`: Character vector of policies allowed for this role. If set, tokens can be created with any subset of the policies in this list, rather than the normal semantics of tokens being a subset of the calling token's policies. The parameter is a comma-delimited string of policy names. If at creation time `no_default_policy` is not set and "default" is not contained in disallowed_policies, the "default" policy will be added to the created token automatically.
- `disallowed_policies`: Character vector of policies forbidden for this role. If set, successful token creation via this role will require that no policies in the given list are requested. Adding "default" to this list will prevent "default" from being added automatically to created tokens.
- `orphan`: If `TRUE`, then tokens created against this policy will be orphan tokens (they will have no parent). As such, they will not be automatically revoked by the revocation of any other token.
- `period`: A duration (e.g., `1h`). If specified, the token will be periodic; it will have no maximum TTL (unless an "explicit-max-ttl" is also set) but every renewal will use the given period. Requires a root/sudo token to use.
- `renewable`: Set to `FALSE` to disable the ability of the token to be renewed past its initial TTL. The default value of `TRUE` will allow the token to be renewable up to the system/mount maximum TTL.
- `explicit_max_ttl`: An integer number of seconds. Provides a maximum lifetime for any tokens issued against this role, including periodic tokens. Unlike direct token creation, where the value for an explicit max TTL is stored in the token, for roles this check will always use the current value set in the role. The main use of this is to provide a hard upper bound on periodic tokens, which otherwise can live forever as long as they are renewed. This is an integer number of seconds.
- `path_suffix`: A string. If set, tokens created against this role will have the given suffix as part of their path in addition to the role name. This can be useful in certain scenarios, such as keeping the same role name in the future but revoking all tokens created against it before some point in time. The suffix can be changed, allowing new callers to have the new suffix as part of their path, and then tokens with the old suffix can be revoked via `/sys/leases/revoke-prefix`.

- bound_cidrs: Character vector of CIDRS. If set, restricts usage of the generated token to client IPs falling within the range of the specified CIDR(s). Unlike most other role parameters, this is not reevaluated from the current role value at each usage; it is set on the token itself. Root tokens with no TTL will not be bound by these CIDRs; root tokens with TTLs will be bound by these CIDRs.
- token_type: Specifies the type of tokens that should be returned by the role. If either service or batch is specified, that kind of token will always be returned. If default-service, then service tokens will be returned unless the client requests a batch type token at token creation time. If default-batch, then batch tokens will be returned unless the client requests a service type token at token creation time.

role_delete Delete a named token role

Usage:

```
role_delete(role_name)
```

Arguments:

- role_name: The name of the role to delete

tidy Performs some maintenance tasks to clean up invalid entries that may remain in the token store. Generally, running this is not needed unless upgrade notes or support personnel suggest it. This may perform a lot of I/O to the storage method so should be used sparingly.

Usage:

```
tidy()
```

login Unlike other auth backend login methods, this does not actually log in to the vault. Instead it verifies that a token can be used to communicate with the vault.

Usage:

```
login(token = NULL, quiet = FALSE)
```

Arguments:

- token: The token to test
- quiet: Logical scalar, set to TRUE to suppress informational messages.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  # There are lots of token methods here:
  client$token

  # To demonstrate, it will be useful to create a restricted
  # policy that can only read from the /secret path
  rules <- 'path "secret/*" {policy = "read"}'
  client$policy$write("read-secret", rules)
  client$write("/secret/path", list(key = "value"))

  # Create a token that has this policy
  token <- client$auth$token$create(policies = "read-secret")
```

```
  alice <- vaultr::vault_client(addr = server$addr)
  alice$login(method = "token", token = token)
  alice$read("/secret/path")

  client$token$lookup(token)

  # We can query the capabilities of this token
  client$token$capabilities("secret/path", token)

  # Tokens are not safe to pass around freely because they *are*
  # the ability to login, but the `token$create` command also
  # provides an accessor:
  accessor <- attr(token, "info")$accessor

  # It is not possible to derive the token from the accessor, but
  # we can use the accessor to ask vault what it could do if it
  # did have the token (and do things like revoke the token)
  client$token$capabilities_accessor("secret/path", accessor)

  client$token$revoke_accessor(accessor)
  try(client$token$capabilities_accessor("secret/path", accessor))

  # cleanup
  server$kill()
}
```

`vault_client_tools` *Vault Tools*

Description

Interact with vault's cryptographic tools. This provides support for high-quality random numbers and cryptographic hashes. This functionality is also available through the transit secret engine.

Methods

`random` Generates high-quality random bytes of the specified length. This is totally independent of R's random number stream and provides random numbers suitable for cryptographic purposes.
Usage:

```
random(bytes = 32, format = "hex")
```

Arguments:

- `bytes`: Number of bytes to generate (as an integer)
- `format`: The output format to produce; must be one of `hex` (a single hex string such as `d1189e2f83b72ab6`), `base64` (a single base64 encoded string such as `8TDJekY0mYs=`) or `raw` (a raw vector of length `bytes`).

`hash` Generates a cryptographic hash of given data using the specified algorithm.
Usage:

```
hash(data, algorithm = NULL, format = "hex")
```

Arguments:

- `data`: A raw vector of data to hash. To generate a raw vector from an R object, one option is to use `unserialize(x, NULL)` but be aware that version information may be included. Alternatively, for a string, one might use `charToRaw`.
- `algorithm`: A string indicating the hash algorithm to use. The exact set of supported algorithms may depend by vault server version, but as of version 1.0.0 vault supports `sha2-224`, `sha2-256`, `sha2-384` and `sha2-512`. The default is `sha2-256`.
- `format`: The format of the output - must be one of `hex` or `base64`.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  # Random bytes in hex
  client$tools$random()
  # base64
  client$tools$random(format = "base64")
  # raw
  client$tools$random(10, format = "raw")

  # Hash data:
  data <- charToRaw("hello vault")
  # will produce 55e702...92efd40c2a4
  client$tools$hash(data)

  # sha2-512 hash:
  client$tools$hash(data, "sha2-512")

  # cleanup
  server$kill()
}
```

`vault_client_transit` *Transit Engine*

Description

Interact with vault's `transit` engine. This is useful for encrypting arbitrary data without storing it in the vault - like "cryptography as a service" or "encryption as a service". The transit secrets engine can also sign and verify data; generate hashes and HMACs of data; and act as a source of random bytes. See https://www.vaultproject.io/docs/secrets/transit/index.html for an introduction to the capabilities of the `transit` engine.

Methods

`custom_mount` Set up a `vault_client_transit` object at a custom mount. For example, suppose you mounted the `transit` secret backend at `/transit2` you might use `tr <- vault$secrets$transit$custom_mount` - this pattern is repeated for other secret and authentication backends.
Usage:

```
custom_mount(mount)
```

Arguments:

- `mount`: String, indicating the path that the engine is mounted at.

`key_create` Create a new named encryption key of the specified type. The values set here cannot be changed after key creation.
Usage:

```
key_create(name, key_type = NULL, convergent_encryption = NULL,
      derived = NULL, exportable = NULL, allow_plaintext_backup = NULL)
```

Arguments:

- `name`: Name for the key. This will be used in all future interactions with the key - the key itself is not returned.
- `key_type`: Specifies the type of key to create. The default is `aes256-gcm96`. The currently-supported types are:
  - `aes256-gcm96` AES-256 wrapped with GCM using a 96-bit nonce size AEAD (symmetric, supports derivation and convergent encryption)
  - `chacha20-poly1305` ChaCha20-Poly1305 AEAD (symmetric, supports derivation and convergent encryption)
  - `ed25519` ED25519 (asymmetric, supports derivation). When using derivation, a sign operation with the same context will derive the same key and signature; this is a signing analogue to `convergent_encryption`
  - `ecdsa-p256` ECDSA using the P-256 elliptic curve (asymmetric)
  - `rsa-2048` RSA with bit size of 2048 (asymmetric)
  - `rsa-4096` RSA with bit size of 4096 (asymmetric)
- `convergent_encryption`: Logical with default of `FALSE`. If `TRUE`, then the key will support convergent encryption, where the same plaintext creates the same ciphertext. This requires derived to be set to true. When enabled, each encryption(/decryption/rewrap/datakey) operation will derive a `nonce` value rather than randomly generate it.
- `derived`: Specifies if key derivation is to be used. If enabled, all encrypt/decrypt requests to this named key must provide a context which is used for key derivation (default is `FALSE`).
- `exportable`: Enables keys to be exportable. This allows for all the valid keys in the key ring to be exported. Once set, this cannot be disabled (default is `FALSE`).
- `allow_plaintext_backup`: If set, enables taking backup of named key in the plaintext format. Once set, this cannot be disabled (default is `FALSE`).

`key_read` Read information about a previously generated key. The returned object shows the creation time of each key version; the values are not the keys themselves. Depending on the type of key, different information may be returned, e.g. an asymmetric key will return its public key in a standard format for the type.
Usage:

```
key_read(name)
```

Arguments:

- `name`: The name of the key to read

`key_list` List names of all keys
Usage:

```
key_list()
```

`key_delete` Delete a key by name. It will no longer be possible to decrypt any data encrypted with the named key. Because this is a potentially catastrophic operation, the `deletion_allowed` tunable must be set using `$key_update()`.
Usage:

```
key_delete(name)
```

Arguments:

- `name`: The name of the key to delete.

`key_update` This method allows tuning configuration values for a given key. (These values are returned during a read operation on the named key.)
Usage:

```
key_update(name, min_decryption_version = NULL, min_encryption_version = NULL,
     deletion_allowed = NULL, exportable = NULL, allow_plaintext_backup = NULL)
```

Arguments:

- `name`: The name of the key to update
- `min_decryption_version`: Specifies the minimum version of ciphertext allowed to be decrypted, as an integer (default is `0`). Adjusting this as part of a key rotation policy can prevent old copies of ciphertext from being decrypted, should they fall into the wrong hands. For signatures, this value controls the minimum version of signature that can be verified against. For HMACs, this controls the minimum version of a key allowed to be used as the key for verification.
- `min_encryption_version`: Specifies the minimum version of the key that can be used to encrypt plaintext, sign payloads, or generate HMACs, as an integer (default is `0`). Must be 0 (which will use the latest version) or a value greater or equal to `min_decryption_version`.
- `deletion_allowed`: Specifies if the key is allowed to be deleted, as a logical (default is `FALSE`).
- `exportable`: Enables keys to be exportable. This allows for all the valid keys in the key ring to be exported. Once set, this cannot be disabled.
- `allow_plaintext_backup`: If set, enables taking backup of named key in the plaintext format. Once set, this cannot be disabled.

`key_rotate` Rotates the version of the named key. After rotation, new plaintext requests will be encrypted with the new version of the key. To upgrade ciphertext to be encrypted with the latest version of the key, use the rewrap endpoint. This is only supported with keys that support encryption and decryption operations.
Usage:

```
key_rotate(name)
```

Arguments:

- name: The name of the key to rotate

key_export Export the named key. If version is specified, the specific version will be returned. If latest is provided as the version, the current key will be provided. Depending on the type of key, different information may be returned. The key must be exportable to support this operation and the version must still be valid.
Usage:

```
key_export(name, key_type, version = NULL)
```

Arguments:

- name: Name of the key to export
- key_type: Specifies the type of the key to export. Valid values are: encryption-key, signing-key and hmac-key.
- version: Specifies the version of the key to read. If omitted, all versions of the key will be returned. If the version is set to latest, the current key will be returned

Details: For more details see https://github.com/hashicorp/vault/issues/2667 where HashiCorp says "Part of the "contract" of transit is that the key is never exposed outside of Vault. We added the ability to export keys because some enterprises have key escrow requirements, but it leaves a permanent mark in the key metadata. I suppose we could at some point allow importing a key and also leave such a mark."

data_encrypt This endpoint encrypts the provided plaintext using the named key.
Usage:

```
data_encrypt(key_name, data, key_version = NULL, context = NULL)
```

Arguments:

- key_name: Specifies the name of the encryption key to encrypt against.
- data: Data to encrypt, as a raw vector
- key_version: Key version to use, as an integer. If not set, uses the latest version. Must be greater than or equal to the key's min_encryption_version, if set.
- context: Specifies the context for key derivation. This is required if key derivation is enabled for this key. Must be a raw vector.

data_decrypt Decrypts the provided ciphertext using the named key.
Usage:

```
data_decrypt(key_name, data, context = NULL)
```

Arguments:

- key_name: Specifies the name of the encryption key to decrypt with.
- data: The data to decrypt. Must be a string, as returned by $data_encrypt.
- context: Specifies the context for key derivation. This is required if key derivation is enabled for this key. Must be a raw vector.

data_rewrap Rewraps the provided ciphertext using the latest version of the named key. Because this never returns plaintext, it is possible to delegate this functionality to untrusted users or scripts.
Usage:

```
data_rewrap(key_name, data, key_version = NULL, context = NULL)
```

Arguments:

- `key_name`: Specifies the name of the encryption key to re-encrypt against
- `data`: The data to decrypt. Must be a string, as returned by `$data_encrypt`.
- `key_version`: Specifies the version of the key to use for the operation. If not set, uses the latest version. Must be greater than or equal to the key's `min_encryption_version`, if set.
- `context`: Specifies the context for key derivation. This is required if key derivation is enabled for this key. Must be a raw vector.

`datakey_create` This endpoint generates a new high-entropy key and the value encrypted with the named key. Optionally return the plaintext of the key as well.
Usage:

```
datakey_create(name, plaintext = FALSE, bits = NULL, context = NULL)
```

Arguments:

- `name`: Specifies the name of the encryption key to use to encrypt the datakey
- `plaintext`: Logical, indicating if the plaintext key should be returned.
- `bits`: Specifies the number of bits in the desired key. Can be 128, 256, or 512.
- `context`: Specifies the context for key derivation. This is required if key derivation is enabled for this key. Must be a raw vector.

`random` Generates high-quality random bytes of the specified length. This is totally independent of R's random number stream and provides random numbers suitable for cryptographic purposes.
Usage:

```
random(bytes = 32, format = "hex")
```

Arguments:

- `bytes`: Number of bytes to generate (as an integer)
- `format`: The output format to produce; must be one of `hex` (a single hex string such as `d1189e2f83b72ab6`), `base64` (a single base64 encoded string such as `8TDJekY0mYs=`) or `raw` (a raw vector of length `bytes`).

`hash` Generates a cryptographic hash of given data using the specified algorithm.
Usage:

```
hash(data, algorithm = NULL, format = "hex")
```

Arguments:

- `data`: A raw vector of data to hash. To generate a raw vector from an R object, one option is to use `unserialize(x, NULL)` but be aware that version information may be included. Alternatively, for a string, one might use `charToRaw`.
- `algorithm`: A string indicating the hash algorithm to use. The exact set of supported algorithms may depend by vault server version, but as of version 1.0.0 vault supports `sha2-224`, `sha2-256`, `sha2-384` and `sha2-512`. The default is `sha2-256`.
- `format`: The format of the output - must be one of `hex` or `base64`.

`hmac` This endpoint returns the digest of given data using the specified hash algorithm and the named key. The key can be of any type supported by the `transit` engine; the raw key will be marshalled into bytes to be used for the HMAC function. If the key is of a type that supports rotation, the latest (current) version will be used.
Usage:

```
hmac(name, data, key_version = NULL, algorithm = NULL)
```

Arguments:

- name: Specifies the name of the encryption key to generate hmac against
- data: The input data, as a raw vector
- key_version: Specifies the version of the key to use for the operation. If not set, uses the latest version. Must be greater than or equal to the key's min_encryption_version, if set.
- algorithm: Specifies the hash algorithm to use. Currently-supported algorithms are: sha2-224, sha2-256, sha2-384 and sha2-512. The default is sha2-256.

`sign` Returns the cryptographic signature of the given data using the named key and the specified hash algorithm. The key must be of a type that supports signing.
Usage:

```
sign(name, data, key_version = NULL, hash_algorithm = NULL,
      prehashed = FALSE, signature_algorithm = NULL, context = NULL)
```

Arguments:

- name: Specifies the name of the encryption key to use for signing
- data: The input data, as a raw vector
- key_version: Specifies the version of the key to use for signing. If not set, uses the latest version. Must be greater than or equal to the key's min_encryption_version, if set,
- hash_algorithm: Specifies the hash algorithm to use. Currently-supported algorithms are: sha2-224, sha2-256, sha2-384 and sha2-512. The default is sha2-256.
- prehashed: Set to true when the input is already hashed. If the key type is rsa-2048 or rsa-4096, then the algorithm used to hash the input should be indicated by the hash_algorithm parameter.
- signature_algorithm: When using a RSA key, specifies the RSA signature algorithm to use for signing. Supported signature types are pss (the default) and pkcs1v15.
- context: Specifies the context for key derivation. This is required if key derivation is enabled for this key. Must be a raw vector.

`verify_signature` Determine whether the provided signature is valid for the given data.
Usage:

```
verify_signature(name, data, signature, hash_algorithm = NULL, signature_algorithm = NULL,
      context = NULL, prehashed = FALSE)
```

Arguments:

- name: Name of the key
- data: Data to verify, as a raw vector
- signature: The signed data, as a string.
- hash_algorithm: Specifies the hash algorithm to use. This can also be specified as part of the URL (see $sign and $hmac for details).
- signature_algorithm: When using a RSA key, specifies the RSA signature algorithm to use for signature verification
- context: Specifies the context for key derivation. This is required if key derivation is enabled for this key. Must be a raw vector.

- prehashed: Set to TRUE when the input is already hashed

verify_hmac Determine whether the provided signature is valid for the given data.
Usage:

```
verify_hmac(name, data, signature, hash_algorithm = NULL, signature_algorithm = NULL,
      context = NULL, prehashed = FALSE)
```

Arguments:

- name: Name of the key
- data: Data to verify, as a raw vector
- signature: The signed data, as a string.
- hash_algorithm: Specifies the hash algorithm to use. This can also be specified as part of the URL (see $sign and $hmac for details).
- signature_algorithm: When using a RSA key, specifies the RSA signature algorithm to use for signature verification
- context: Specifies the context for key derivation. This is required if key derivation is enabled for this key. Must be a raw vector.
- prehashed: Set to TRUE when the input is already hashed

key_backup Returns a plaintext backup of a named key. The backup contains all the configuration data and keys of all the versions along with the HMAC key. The response from this endpoint can be used with $key_restore to restore the key.
Usage:

```
key_backup(name)
```

Arguments:

- name: Name of the key to backup

key_restore Restores the backup as a named key. This will restore the key configurations and all the versions of the named key along with HMAC keys. The input to this method should be the output of $key_restore method.
Usage:

```
key_restore(name, backup, force = FALSE)
```

Arguments:

- name: Name of the restored key.
- backup: Backed up key data to be restored. This should be the output from the $key_backup endpoint.
- force: Logical. If TRUE, then force the restore to proceed even if a key by this name already exists.

key_trim This endpoint trims older key versions setting a minimum version for the keyring. Once trimmed, previous versions of the key cannot be recovered.
Usage:

```
key_trim(name, min_version)
```

Arguments:

- name: Key to trim

- `min_version`: The minimum version for the key ring. All versions before this version will be permanently deleted. This value can at most be equal to the lesser of `min_decryption_version` and `min_encryption_version`. This is not allowed to be set when either `min_encryption_version` or `min_decryption_version` is set to zero.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)
if (!is.null(server)) {
  client <- server$client()

  client$secrets$enable("transit")
  transit <- client$secrets$transit

  # Before encrypting anything, create a key.  Note that it will
  # not be returned to you, and is accessed purely by name
  transit$key_create("test")

  # Some text to encrypt
  plaintext <- "hello world"

  # Encrypted:
  cyphertext <- transit$data_encrypt("test", charToRaw(plaintext))

  # Decrypt the data
  res <- transit$data_decrypt("test", cyphertext)
  rawToChar(res)

  # This approach works with R objects too, if used with serialise.
  # First, serialise an R object to a raw vector:
  data <- serialize(mtcars, NULL)

  # Then encrypt this data:
  enc <- transit$data_encrypt("test", data)

  # The resulting string can be safely passed around (e.g., over
  # email) or written to disk, and can later be decrypted by
  # anyone who has access to the "test" key in the vault:
  data2 <- transit$data_decrypt("test", enc)

  # Once decrypted, the data can be "unserialised" back into an R
  # object:
  unserialize(data2)

  # cleanup
  server$kill()
}
```

`vault_resolve_secrets` *Resolve secrets from R objects*

Description

Use vault to resolve secrets. This is a convenience function that wraps a pattern that we have used in a few applications of vault. The idea is to allow replacement of data in configuration with special strings that indicate that the string refers to a vault secret. This function resolves those secrets.

Usage

```
vault_resolve_secrets(x, ..., login = TRUE)
```

Arguments

x	List of values, some of which may refer to vault secrets (see Details for pattern). Any values that are not strings or do not match the pattern of a secret are left as-is.
...	Args to be passed to vault_client call.
login	Login method to be passed to call to vault_client.

Details

For each element of the data, if a string matches the form:

```
VAULT:<path to secret>:<field>
```

then it will be treated as a vault secret and resolved. The <path to get> will be something like /secret/path/password and the <field> the name of a field in the key/value data stored at that path. For example, suppose you have the data list(username = "alice", password = "s3cret!") stored at /secret/database/user, then the string

```
VAULT:/secret/database/user:password
```

would refer to the value s3cret!

Value

List of properties with any vault secrets resolved.

Examples

```
server <- vaultr::vault_test_server(if_disabled = message)

if (!is.null(server)) {
  client <- server$client()
  # The example from above:
  client$write("/secret/database/user",
               list(username = "alice", password = "s3cret!"))

  # A list of data that contains a mix of secrets to be resolved
  # and other data:
```

```
  x <- list(user = "alice",
            password = "VAULT:/secret/database/user:password",
            port = 5678)

  # Explicitly pass in the login details and resolve the secrets:
  vaultr::vault_resolve_secrets(x, login = "token", token = server$token,
                                addr = server$addr)

  # Alternatively, if appropriate environment variables are set
  # then this can be done more easily:
  if (requireNamespace("withr", quietly = TRUE)) {
    env <- c(VAULTR_AUTH_METHOD = "token",
             VAULT_TOKEN = server$token,
             VAULT_ADDR = server$addr)
    withr::with_envvar(env, vault_resolve_secrets(x))
  }
}
```

`vault_test_server` *Control a test vault server*

Description

Control a server for use with testing. This is designed to be used only by other packages that wish to run tests against a vault server. You will need to set `VAULTR_TEST_SERVER_BIN_PATH` to point at the directory containing the vault binary.

Usage

```
vault_test_server(https = FALSE, init = TRUE,
  if_disabled = testthat::skip)

vault_test_server_install(path = NULL, quiet = FALSE,
  version = "1.0.0", platform = vault_platform())
```

Arguments

`https`	Logical scalar, indicating if a https-using server should be created, rather than the default vault dev-mode server. This is still *entirely* insecure, and uses self signed certificates that are bundled with the package.
`init`	Logical scalar, indicating if the https-using server should be initialised.
`if_disabled`	Callback function to run if the vault server is not enabled. The default, designed to be used within tests, is `testthat::skip`. Alternatively, inspect the `$enabled` property of the returned object.
`path`	Path in which to install vault test server. Leave as NULL to use the *VAULTR_TEST_SERVER_BIN_PATH* environment variable.
`quiet`	Suppress progress bars on install

version	Version of vault to install
platform	For testing, overwrite the platform vault is being installed on, with either "windows", "darwin" or "linux".

Details

Once created with vault_test_server, a server will stay alive for as long as the R process is alive *or* until the vault_server_instance object goes out of scope and is garbage collected. Calling $kill() will explicitly stop the server, but this is not strictly needed. See below for methods to control the server instance.

The function vault_test_server_install will install a test server, but *only* if the user sets the following environmental variables:

- VAULTR_TEST_SERVER_INSTALL to "true" to opt in to the download.
- VAULTR_TEST_SERVER_BIN_PATH to the directory where the binary should be downloaded to.
- NOT_CRAN to "true" to indicate this is not running on CRAN as it requires installation of a binary from a website.

This will download a ~100MB binary from https://vaultproject.io so use with care. It is intended *only* for use in automated testing environments.

Methods

addr The vault address; this is suitable for using with vault_client (read-only).

port The vault port (read-only).

token The vault root token, from when the testing vault server was created. If the vault is rekeyed this will no longer be accurate (read-only).

keys Key shares from when the vault was initialised (read-only).

cacert Path to the https certificate, if running in https mode (read-only).

version Return the server version, as a numeric_version object.

Usage:

```
version()
```

client Create a new client that can use this server. The client will be a vault_client object.

Usage:

```
client(login = TRUE, quiet = TRUE)
```

Arguments:

- login: Logical, indicating if the client should login to the server (default is TRUE).
- quiet: Logical, indicating if informational messages should be suppressed. Default is TRUE, in contrast with most other methods.

env Return a named character vector of environment variables that can be used to communicate with this vault server (VAULT_ADDR, VAU:T_TOKEN, etc).

Usage:

```
env()
```

`export` Export the variables returned by the `$env()` method to the environment. This makes them available to child processes.
Usage:

```
export()
```

`clear_cached_token` Clear any session-cached token for this server. This is intended for testing new authentication backends.
Usage:

```
clear_cached_token()
```

`kill` Kill the server.
Usage:

```
kill()
```

Warning

Starting a server in test mode must *not* be used for production under any circumstances. As the name suggests, `vault_test_server` is a server suitable for *tests* only and lacks any of the features required to make vault secure. Please see https://www.vaultproject.io/docs/concepts/dev-server.html for more information

The `vault_test_server_install` function will download a binary from HashiCorp in order to use a vault server. Use this function with care. The download will happen from https://releases.hashicorp.com/vault (over https). This function is primarily designed to be used from continuous integration services only and for local use you are strongly recommended to curate your own installations.

Examples

```
# Try and start a server; if one is not enabled (see details
# above) then this will return \code{NULL}
server <- vault_test_server(if_disabled = message)

if (!is.null(server)) {
  # We now have a server running on an arbitrary high port - note
  # that we are running over http and in dev mode: this is not at
  # all suitable for production use, just for tests
  server$addr

  # Create clients using the client method - by default these are
  # automatically authenticated against the server
  client <- server$client()
  client$write("/secret/password", list(value = "s3cret!"))
  client$read("/secret/password")

  # The server stops automatically when the server object is
  # garbage collected, or it can be turned off with the
  # \code{kill} method:
  server$kill()
  tryCatch(client$status(), error = function(e) message(e$message))
}
```

Index