ACL

The Redis ACL, short for Access Control List, is the feature that allows certain connections to be limited in terms of the commands that can be executed and the keys that can be accessed. The way it works is that, after connecting, a client is required to authenticate providing a username and a valid password: if the authentication stage succeeded, the connection is associated with a given user and the limits the user has. Redis can be configured so that new connections are already authenticated with a "default" user (this is the default configuration), so configuring the default user has, as a side effect, the ability to provide only a specific subset of functionalities to connections that are not explicitly authenticated.

In the default configuration, Redis 6 (the first version to have ACLs) works exactly like older versions of Redis, that is, every new connection is capable of calling every possible command and accessing every key, so the ACL feature is backward compatible with old clients and applications. Also the old way to configure a password, using the **requirepass** configuration directive, still works as expected, but now what it does is just to set a password for the default user.

The Redis AUTH command was extended in Redis 6, so now it is possible to use it in the two-arguments form:

```
AUTH <username> <password>
```

When it is used according to the old form, that is:

```
AUTH <password>
```

What happens is that the username used to authenticate is "default", so just specifying the password implies that we want to authenticate against the default user. This provides perfect backward compatibility with the past.

When ACLs are useful

Before using ACLs you may want to ask yourself what's the goal you want to accomplish by implementing this layer of protection. Normally there are two main goals that are well served by ACLs:

1. You want to improve security by restricting the access to commands and keys, so that untrusted clients have no access and trusted clients have just the minimum access level to the database in order to perform the work needed. For instance certain clients may just be able to execute read only commands.
2. You want to improve operational safety, so that processes or humans accessing Redis are not allowed, because of software errors or manual mistakes, to damage the data or the configuration. For instance there is no reason for a worker that fetches delayed jobs from Redis to be able to call the FLUSHALL command.

Another typical usage of ACLs is related to managed Redis instances. Redis is often provided as a managed service both by internal company teams that handle the Redis infrastructure for the other internal customers they have, or is provided in a software-as-a-service setup by cloud providers. In both such setups we want to be sure that configuration commands are excluded for the customers. The way this was accomplished in the past, via command renaming, was a trick that allowed us to survive without ACLs for a long time, but is not ideal.

Configuring ACLs using the ACL command

ACLs are defined using a DSL (domain specific language) that describes what a given user is able to do or not. Such rules are always implemented from the first to the last, left-to-right, because sometimes the order of the rules is important to understand what the user is really able to do.

By default there is a single user defined, that is called *default*. We can use the ACL LIST command in order to check the currently active ACLs and verify what the configuration of a freshly started, defaults-configured Redis instance is:

```
> ACL LIST
1) "user default on nopass ~* &* +@all"
```

The command above reports the list of users in the same format that is used in the Redis configuration files, by translating the current ACLs set for the users back into their description.

The first two words in each line are "user" followed by the username. The next words are ACL rules that describe different things. We'll show in details how the rules work, but for now it is enough to say that the default user is configured to be active (on), to require no password (nopass), to access every possible key (~*) and Pub/Sub channel (&*), and be able to call every possible command (+@all).

Also, in the special case of the default user, having the *nopass* rule means that new connections are automatically authenticated with the default user without any explicit

AUTH call needed.

ACL rules

The following is the list of the valid ACL rules. Certain rules are just single words that are used in order to activate or remove a flag, or to perform a given change to the user ACL. Other rules are char prefixes that are concatenated with command or categories names, or key patterns, and so forth.

Enable and disallow users:

- `on`: Enable the user: it is possible to authenticate as this user.
- `off`: Disable the user: it's no longer possible to authenticate with this user, however the already authenticated connections will still work. Note that if the default user is flagged as *off*, new connections will start not authenticated and will require the user to send AUTH or HELLO with the AUTH option in order to authenticate in some way, regardless of the default user configuration.

Allow and disallow commands:

- `+<command>`: Add the command to the list of commands the user can call.
- `-<command>`: Remove the command to the list of commands the user can call.
- `+@<category>`: Add all the commands in such category to be called by the user, with valid categories being like @admin, @set, @sortedset, ... and so forth, see the full list by calling the ACL CAT command. The special category @all means all the commands, both the ones currently present in the server, and the ones that will be loaded in the future via modules.
- `-@<category>`: Like `+@<category>` but removes the commands from the list of commands the client can call.
- `+<command>|subcommand`: Allow a specific subcommand of an otherwise disabled command. Note that this form is not allowed as negative like `-DEBUG|SEGFAULT`, but only additive starting with "+". This ACL will cause an error if the command is already active as a whole.
- `allcommands`: Alias for +@all. Note that it implies the ability to execute all the future commands loaded via the modules system.
- `nocommands`: Alias for -@all.

Allow and disallow certain keys:

- `~<pattern>`: Add a pattern of keys that can be mentioned as part of commands. For instance `~*` allows all the keys. The pattern is a glob-style pattern like the one of KEYS. It is possible to specify multiple patterns.
- `allkeys`: Alias for `~*`.
- `resetkeys`: Flush the list of allowed keys patterns. For instance the ACL `~foo:* ~bar:* resetkeys ~objects:*`, will result in the client only be able to access keys matching the pattern `objects:*`.

Allow and disallow Pub/Sub channels:

- `&<pattern>`: Add a glob style pattern of Pub/Sub channels that can be accessed by the user. It is possible to specify multiple channel patterns. Note that pattern matching is done only for channels mentioned by PUBLISH and SUBSCRIBE, whereas PSUBSCRIBE requires a literal match between its channel patterns and those allowed for user.
- `allchannels`: Alias for `&*` that allows the user to access all Pub/Sub channels.
- `resetchannels`: Flush the list of allowed channel patterns and disconnect the user's Pub/Sub clients if these are no longer able to access their respective channels and/or channel patterns.

Configure valid passwords for the user:

- `><password>`: Add this password to the list of valid passwords for the user. For example `>mypass` will add "mypass" to the list of valid passwords. This directive clears the *nopass* flag (see later). Every user can have any number of passwords.
- `<<password>`: Remove this password from the list of valid passwords. Emits an error in case the password you are trying to remove is actually not set.
- `#<hash>`: Add this SHA-256 hash value to the list of valid passwords for the user. This hash value will be compared to the hash of a password entered for an ACL user. This allows users to store hashes in the `acl.conf` file rather than storing cleartext passwords. Only SHA-256 hash values are accepted as the password hash must be 64 characters and only container lowercase hexadecimal characters.
- `!<hash>`: Remove this hash value from from the list of valid passwords. This is useful when you do not know the password specified by the hash value but would like to remove the password from the user.
- `nopass`: All the set passwords of the user are removed, and the user is flagged as requiring no password: it means that every password will work against this user. If this directive is used for the default user, every new connection will be immediately authenticated with the default user without any explicit AUTH command required. Note that the *resetpass* directive will clear this condition.
- `resetpass`: Flush the list of allowed passwords. Moreover removes the *nopass* status. After *resetpass* the user has no associated passwords and there is no way to authenticate without adding some password (or setting it as *nopass* later).

Note: an use that is not flagged with nopass, and has no list of valid passwords, is effectively impossible to use, because there will be no way to log in as such user.

Reset the user:

- `reset` Performs the following actions: resetpass, resetkeys, resetchannels, off, -@all. The user returns to the same state it has immediately after its creation.

Creating and editing users ACLs with the ACL SETUSER command

Users can be created and modified in two main ways:

1. Using the ACL command and its ACL SETUSER subcommand.
2. Modifying the server configuration, where users can be defined, and restarting the server, or if we are using an *external ACL file*, just issuing ACL LOAD.

In this section we'll learn how to define users using the `ACL` command. With such knowledge it will be trivial to do the same things via the configuration files. Defining users in the configuration deserves its own section and will be discussed later separately.

To start let's try the simplest ACL SETUSER command call:

```
> ACL SETUSER alice
OK
```

The `SETUSER` command takes the username and a list of ACL rules to apply to the user. However in the above example I did not specify any rule at all. This will just create the user if it did not exist, using the defaults for new users. If the user already exist, the command above will do nothing at all.

Let's check what is the default user status:

```
> ACL LIST
1) "user alice off &* -@all"
2) "user default on nopass ~* ~& +@all"
```

The just created user "alice" is:

- In off status, that is, it's disabled. AUTH will not work.
- The user also has no passwords set.
- Cannot access any command. Note that the user is created by default without the ability to access any command, so the `-@all` in the output above could be omitted, however ACL LIST attempts to be explicit rather than implicit.
- There are no key patterns that the user can access.
- The user can access all Pub/Sub channels.

New users are created with restrictive permissions by default. Starting with Redis 6.2, ACL provides Pub/Sub channels access management as well. To ensure backwards compatability with version 6.0 when upgrading to Redis 6.2, new users are granted the 'allchannels' permission by default. The default can be set to `resetchannels` via the `acl-pubsub-default` configuration directive.

Such user is completely useless. Let's try to define the user so that it is active, has a password, and can access with only the GET command to key names starting with the string "cached:".

```
> ACL SETUSER alice on >p1pp0 ~cached:* +get
OK
```

Now the user can do something, but will refuse to do other things:

```
> AUTH alice p1pp0
OK
> GET foo
(error) NOPERM this user has no permissions to access one of the keys
> GET cached:1234
(nil)
> SET cached:1234 zap
(error) NOPERM this user has no permissions to run the 'set' command
```

Things are working as expected. In order to inspect the configuration of the user alice (remember that user names are case sensitive), it is possible to use an alternative to ACL LIST which is designed to be more suitable for computers to read, while ACL LIST is more biased towards humans.

```
> ACL GETUSER alice
1) "flags"
2) 1) "on"
   2) "allchannels"
3) "passwords"
4) 1) "2d9c75..."
5) "commands"
6) "-@all +get"
7) "keys"
8) 1) "cached:*"
9) "channels"
10) 1) "*"
```

The ACL GETUSER returns a field-value array describing the user in more parsable terms. The output includes the set of flags, a list of key patterns, passwords and so forth. The output is probably more readable if we use RESP3, so that it is returned as as map reply:

```
> ACL GETUSER alice
1# "flags" => 1~ "on"
   2~ "allchannels"
2# "passwords" => 1) "2d9c75273d72b32df726fb545c8a4edc719f0a95a6fd993
3# "commands" => "-@all +get"
4# "keys" => 1) "cached:*"
5# "channels" => 1) "*"
```

Note: from now on we'll continue using the Redis default protocol, version 2, because it will take some time for the community to switch to the new one.

Using another ACL SETUSER command (from a different user, because alice cannot run the ACL command) we can add multiple patterns to the user:

```
> ACL SETUSER alice ~objects:* ~items:* ~public:*
OK
> ACL LIST
1) "user alice on >2d9c75... ~cached:* ~objects:* ~items:* ~public:*
2) "user default on nopass ~* &* +@all"
```

The user representation in memory is now as we expect it to be.

What happens calling ACL SETUSER multiple times

It is very important to understand what happens when ACL SETUSER is called multiple times. What is critical to know is that every SETUSER call will NOT reset the user, but will just apply the ACL rules to the existing user. The user is reset only if it was not known before: in that case a brand new user is created with zeroed-ACLs, that is, the user cannot do anything, is disabled, has no passwords and so forth: for safety this is the best default.

However later calls will just modify the user incrementally so for instance the following sequence:

```
> ACL SETUSER myuser +set
OK
> ACL SETUSER myuser +get
OK
```

Will result in myuser being able to call both GET and SET:

```
> ACL LIST
1) "user default on nopass ~* &* +@all"
2) "user myuser off &* -@all +set +get"
```

Playings with command categories

Setting users ACLs by specifying all the commands one after the other is really annoying, so instead we do things like that:

```
> ACL SETUSER antirez on +@all -@dangerous >42a979... ~*
```

By saying +@all and -@dangerous we included all the commands and later removed all the commands that are tagged as dangerous inside the Redis command table. Please note that command categories **never include modules commands** with the exception of +@all. If you say +@all all the commands can be executed by the user, even future commands loaded via the modules system. However if you use the ACL rule +@read or any other, the modules commands are always excluded. This is very important because you should just trust the Redis internal command table for sanity. Modules may expose dangerous things and in the case of an ACL that is just additive, that is, in the form of `+@all -...` You should be absolutely sure that you'll never include what you did not mean to.

However to remember that categories are defined, and what commands each category exactly includes, is impossible and would be super boring, so the Redis `ACL` command exports the `CAT` subcommand that can be used in two forms:

```
ACL CAT -- Will just list all the categories available
ACL CAT <category-name> -- Will list all the commands inside the cate
```

Examples:

```
 > ACL CAT
 1) "keyspace"
 2) "read"
 3) "write"
 4) "set"
 5) "sortedset"
 6) "list"
 7) "hash"
 8) "string"
 9) "bitmap"
10) "hyperloglog"
11) "geo"
12) "stream"
13) "pubsub"
14) "admin"
15) "fast"
16) "slow"
17) "blocking"
18) "dangerous"
19) "connection"
20) "transaction"
21) "scripting"
```

As you can see so far there are 21 distinct categories. Now let's check what command is part of the *geo* category:

```
> ACL CAT geo
1) "geohash"
2) "georadius_ro"
3) "georadiusbymember"
4) "geopos"
5) "geoadd"
6) "georadiusbymember_ro"
7) "geodist"
8) "georadius"
```

Note that commands may be part of multiple categories, so for instance an ACL rule like `+@geo -@read` will result in certain geo commands to be excluded because they are read-only commands.

Adding subcommands

Often the ability to exclude or include a command as a whole is not enough. Many Redis commands do multiple things based on the subcommand passed as argument. For example the `CLIENT` command can be used in order to do dangerous and non dangerous operations. Many deployments may not be happy to provide the ability to execute CLIENT KILL to non admin-level users, but may still want them to be able to run CLIENT SETNAME.

Note: the new RESP3 HELLO handshake command provides a SETNAME option, but this is still a good example for subcommand control.

In such case I could alter the ACL of a user in the following way:

```
ACL SETUSER myuser -client +client|setname +client|getname
```

I started removing the `CLIENT` command, and later added the two allowed subcommands. Note that **it is not possible to do the reverse**, the subcommands can be only added, and not excluded, because it is possible that in the future new subcommands may be added: it is a lot safer to specify all the subcommands that are valid for some user. Moreover, if you add a subcommand about a command that is not already disabled, an error is generated, because this can only be a bug in the ACL rules:

```
> ACL SETUSER default +debug|segfault
(error) ERR Error in ACL SETUSER modifier '+debug|segfault': Adding a
subcommand of a command already fully added is not allowed. Remove th
command to start. Example: -DEBUG +DEBUG|DIGEST
```

Note that subcommand matching may add some performance penalty, however such penalty is very hard to measure even with synthetic benchmarks, and the additional CPU cost is only payed when such command is called, and not when other commands are called.

+@all VS -@all

In the previous section it was observed how it is possible to define commands ACLs based on adding/removing single commands.

How passwords are stored internally

Redis internally stores passwords hashed with SHA256, if you set a password and check the output of ACL LIST or GETUSER you'll see a long hex string that looks pseudo random. Here is an example, because in the previous examples, for the sake of brevity, the long hex string was trimmed:

```
> ACL GETUSER default
1) "flags"
2) 1) "on"
   2) "allkeys"
   3) "allcommands"
   4) "allchannels"
3) "passwords"
4) 1) "2d9c75273d72b32df726fb545c8a4edc719f0a95a6fd993950b10c474ad9c9
5) "commands"
6) "+@all"
7) "keys"
8) 1) "*"
9) "channels"
10) 1) "*"
```

Also, starting with Redis 6, the old command `CONFIG GET requirepass` will no longer return the clear text password, but instead the hashed password.

Using SHA256 provides the ability to avoid storing the password in clear text while still allowing for a very fast AUTH command, which is a very important feature of Redis and is coherent with what clients expect from Redis.

However ACL *passwords* are not really passwords: they are shared secrets between the server and the client, because in that case the password is not an authentication token used by a human being. For instance:

```
* There are no length limits, the password will just be memorized in
* The ACL password does not protect any other thing: it will never be
* Often when you are able to access the hashed password itself, by ha
```

For this reason to slowdown the password authentication in order to use an algorithm that uses time and space, in order to make password cracking hard, is a very poor choice. What we suggest instead is to generate very strong password, so that even having the hash nobody will be able to crack it using a dictionary nor a brute force attack. For this reason

there is a special ACL command that generates passwords using the system cryptographic pseudorandom generator:

```
> ACL GENPASS
"dd721260bfe1b3d9601e7fbab36de6d04e2e67b0ef1c53de59d45950db0dd3cc"
```

The command outputs a 32 bytes (256 bit) pseudorandom string converted to a 64 byte alphanumerical string. This is long enough to avoid attacks and short enough to be easy to manage, cut & paste, store and so forth. This is what you should use in order to generate Redis passwords.

Using an external ACL file

There are two ways in order to store users inside the Redis configuration.

```
1. Users can be specified directly inside the `redis.conf` file.
2. It is possible to specify an external ACL file.
```

The two methods are *mutually incompatible*, Redis will ask you to use one or the other. To specify users inside `redis.conf` is a very simple way good for simple use cases. When there are multiple users to define, in a complex environment, we strongly suggest you to use the ACL file.

The format used inside `redis.conf` and in the external ACL file is exactly the same, so it is trivial to switch from one to the other, and is the following:

```
user <username> ... acl rules ...
```

For instance:

```
user worker +@list +@connection ~jobs:* on >ffa9203c493aa99
```

When you want to use an external ACL file, you are required to specify the configuration directive called `aclfile`, like this:

```
aclfile /etc/redis/users.acl
```

When you are just specifying a few users directly inside the `redis.conf` file, you can use CONFIG REWRITE in order to store the new user configuration inside the file by rewriting it.

The external ACL file however is more powerful. You can do the following:

```
* Use [ACL LOAD](/commands/acl-load) if you modified the ACL file man
* USE [ACL SAVE](/commands/acl-save) in order to save the current ACL
```

Note that CONFIG REWRITE does not also trigger ACL SAVE: when you use an ACL file the configuration and the ACLs are handled separately.

ACL rules for Sentinel and Replicas

In case you don't want to provide Redis replicas and Redis Sentinel instances full access to your Redis instances, the following is the set of commands that must be allowed in order for everything to work correctly.

For Sentinel, allow the user to access the following commands both in the master and replica instances:

- AUTH, CLIENT, SUBSCRIBE, SCRIPT, PUBLISH, PING, INFO, MULTI, SLAVEOF, CONFIG, CLIENT, EXEC.

Sentinel does not need to access any key in the database but does use Pub/Sub, so the ACL rule would be the following (note: AUTH is not needed since is always allowed):

```
ACL SETUSER sentinel-user on >somepassword allchannels +multi +slaveo
```

Redis replicas require the following commands to be whitelisted on the master instance:

- PSYNC, REPLCONF, PING

No keys need to be accessed, so this translates to the following rules:

```
ACL setuser replica-user on >somepassword +psync +replconf +ping
```

Note that you don't need to configure the replicas to allow the master to be able to execute any set of commands: the master is always authenticated as the root user from the point of view of replicas.

TODO list for this document

- Make sure to specify that modules commands are ignored when adding/removing categories.
- Document cost of keys matching with some benchmark.
- Document how +@all also includes module commands and every future command.
- Document backward compatibility with requirepass and single argument AUTH.
