NAME

Authenticator – GMA Mapper Service Authenticator

SYNOPSIS

Python usage:

import SoftwareAlchemy.GMA.Mapper.Authentication

auth **= Authenticator(***secret***=None,** *client***=None,** *gm_secret***=None,** *username***=None)**
challenge **=** *auth***.generate_challenge()**
*auth***.accept_challenge(***challenge***)**
response **=** *auth***.generate_response()**
*auth***.validate_response(***response***)**
*auth***.reset()**
*username***,** *client* **=** *auth***.client_info()**

Go usage:

```
type Authenticator struct {
  Secret []byte
  GmSecret []byte
  Challenge []byte
  Client string
  Username string
  GmMode bool
}
```

Type **Authenticator**
  func (*a* ***Authenticator) CurrentChallenge() string**
  func (*a* ***Authenticator) GenerateChallenge() (string, error)**
  func (*a* ***Authenticator) Reset()**
  func (*a* ***Authenticator) SetSecret(***secret* **[]byte) bool**
  func (*a* ***Authenticator) ValidateResponse(***response* **string) (bool, error)**

DESCRIPTION

The challenge/response negotiation between the mapper server and clients is implemented in this object. A client wishing to authenticate should create an authenticator as

auth **= Authenticator(***secret***,** *client***, username=***username***)**

where *secret* is the shared secret (password) used by clients to authenticate to the server, and *client* is a string describing what sort of client this is. If the *username* parameter is passed, it will be sent to the server to identify the user (as shown in chat messages, etc.). Otherwise, the username will be taken from their local computing environment.

Servers should create their authenticator object as

s_auth **= Authenticator(***secret***, gm_secret=***gm_secret***)**

where *secret* is as described above, and *gm_secret* is the password used exclusively by the GM.

The calculation method used for this authentication mechanism is documented in the protocol specification in **mapper**(6).

Note that the *secret* and *gm_secret* parameters are **bytes** values. If you have a password as a regular string value, you'll need to convert it to **bytes** (such as by calling its **encode()** method).

Negotiation Process

On the server side, an **Authenticator** object *s_auth* is constructed for each incoming connection. It issues a challenge unique to its session, and expects to see the corresponding response to that challenge, while other simultaneous incoming connections have their own *s_auth* objects negotiating with their incoming clients.

The server calls *s_auth***.generate_challenge()** to create a random challenge, which is sent to the client.

The client then calls *auth***.accept_challenge()** with the challenge it received from the server, and then calls *auth***.generate_response()** to calculate the expected response. This is sent to the server.

The server then calls *s_auth*.**validate_response()** to determine if the response was valid.

Methods

The following methods are available for any **Authenticator** object:

challenge=*auth*.**generate_challenge()**
: (server) Generate a random challenge for the client. This is stored internally for later validation. Returns a base-64 encoded representation of the challenge.

auth.**accept_challenge(***challenge***)**
: (client) Accept a server's challenge as a base-64 string, storing it internally.

response=*auth*.**generate_response()**
: (client) Generate a response to the previously-accepted challenge. This is returned as a base-64 encoded string value.

bool=*auth*.**validate_response(***response***)**
: (server) Returns **True** if the *response* value matches the response expected for the challenge this object had previously generated. If the response was created using the GM's password, then the *auth*.**gm_mode** attribute is set to **True**, otherwise it will be **False**.

auth.**reset()**
: Resets the internal state of the *auth* object, so it may be used to negotiate a new authentication.

user,*client*=*auth*.**client_info()**
: Returns the username and client currently in use for this authenticator. Nothing in the negotiation provides these; they must have been given to the constructor (or, in the case of the username, determined by looking at the system environment).

Go API

The Go API equivalents work similarly, except as noted below, assuming the variable *auth* is an **Authenticator** type value.

challenge = *auth*.**CurrentChallenge()**
: Returns the base-64 encoded challenge value most recently generated via the **GenerateChallenge()** method. The *auth*.**Challenge** struct member holds a binary representation of the challenge, not the encoded string.

auth.**Reset()**
: Works identically to the Python API.

challenge, *err* = *auth*.**GenerateChallenge()**
: The *challenge* value is only defined if *err* is **nil**; otherwise an error occurred while trying to generate the challenge.

valid, *err* = *auth*.**ValidateResponse(***response***)**
: If an error occurred, *err* will have a non-**nil** value; otherwise *valid* indicates whether or not the authentication attempt succeeded. The *auth*.**GmMode** struct member is also set to indicate the privilege level granted as a result.

At the time of the writing of this document, only the server-side part of the authentication process is implemented in the Go version.

SEE ALSO

socketinterface(3), **mapper**(6).

AUTHOR

Steve Willoughby / steve@alchemy.com.

HISTORY

The Go port appeared in November, 2020, in GMA version 4.2.2.

BUGS

COPYRGHT