Comentario técnico: CTC-094
Componente: **Autorización en RPC de Mongoose-OS con ESP32**
Autor: Sergio R. Caprile, Senior R&D Engineer

Revisiones	Fecha	Comentarios
0	08/05/20	

Mongoose-OS incorpora una API del tipo RPC (Remote Procedural Call) que podemos aprovechar y extender. La misma, tanto a través del servidor HTTP como en otros transportes, incorpora una forma de Digest Authentication que podemos aprovechar para limitar quiénes pueden acceder y qué es lo que pueden hacer. La identificación es simplemente por una pareja usuario/password, mientras que una ACL (Access Control List) decide cuáles usuarios pueden ejecutar cada RPC. Esto no evita que lo que se envía pueda ser observado con un sniffer, tema que requiere TLS, abordado en el CTC-092. El password no es revelado pues la identidad se valida mediante un intercambio de hashes.

Digest Authentication

Hemos visto en el CTC-091 la autorización por HTTP Digest, sin entrar en detalles. La empleada en Mongoose-OS para las RPC es muy similar. De hecho cuando el transporte utilizado es HTTP es idéntica. Si se utiliza WebSocket, la autenticación puede realizarse en los headers HTTP al solicitar el upgrade; sino, al igual que en los otros transportes posibles como UART, MQTT, BLE, el dispositivo devuelve un mensaje similar al que se entregaría en HTTP, pero dentro del objeto JSON que transporta la información de RPC. El cliente responde entonces con un hash MD5 de la información provista en dicho objeto y las credenciales del usuario, volviendo a solicitar la RPC deseada. Veremos un detalle más adelante.

Configuración

Recordemos que la autorización del servidor HTTP no afecta a las RPC, así como la de éstas no afecta al servidor. Si queremos además autenticar al servidor web como tal, deberemos seguir las instrucciones dadas en el CTC-091.
Una característica fundamental de este esquema, al menos a la fecha, es que una vez habilitado opera sobre todos los transportes. Si bien es posible desactivar un transporte, una vez habilitado el esquema de autorización, todos los transportes que deseemos usar deberán implementar Digest Authentication. Veremos un detalle más adelante.
A continuación, configuramos el ESP32 con Mongoose-OS para operar como un Access Point con un web server. La operación como AP nos resulta la forma tal vez más rápida y simple de realizar las pruebas y explicar la operación. La configuración puede realizarse manualmente mediante RPC, en un archivo de configuración JSON; o definirla en el archivo YAML que describe el proyecto. Para las pruebas elegimos esta última opción.

```
config_schema:
  - ["rpc.auth_domain", "myESP"]         # Nombre de dominio (realm)
  - ["rpc.auth_file", "myauth.txt"]      # Archivo conteniendo los pares usuario/password
  - ["rpc.acl_file", "rpc_acl.json"]     # Lista de control de acceso (ACL)
```

Como vemos, al igual que lo observado en el CTC-091, no es algo que resulte muy amigable para modificaciones dinámicas; sino que debe preverse o al menos idear un mecanismo apropiado.

El archivo de configuración tiene el mismo formato utilizado por la autorización en el servidor HTTP, que a su vez es igual al que utilizan los servidores HTTP como Apache, por lo que lo generamos mediante *htdigest* como explicamos en el CTC-091.
La lista de control de acceso tiene el formato:

```
[
  {"method": "<RPC>", "acl": "+/-<usuario>"}[, ...]
]
```

y es leída en forma secuencial, por lo que una vez encontrada una coincidencia en una RPC no se sigue la lectura. Es decir, si queremos autorizar a un usuario a todas las RPC y a otros solamente a algunas RPC, el comodín que representa a todas las RPC debe ubicarse al final de la lista. Los siguientes son ejemplos de listas de acceso; en el primero, el usuario *bob* puede acceder a todas las RPC de `FS`, es decir, `FS.List`, etc. Nadie más. En el segundo, todos los usuarios que se identifiquen correctamente pueden acceder a todas las RPC. En el tercero, *alice* puede acceder a todo mientras que *bob* sólo a `FS`; observemos que debimos incluir a *alice* en la primera línea.

```
[
  {"method": "FS.*", "acl": "+bob"},
  {"method": "*", "acl": "-*"}
]

[
  {"method": "*", "acl": "+*"}
]

[
  {"method": "FS.*", "acl": "+bob,+alice"},
  {"method": "*", "acl": "+alice"},
]
```

Finalmente, deberemos ubicar dichos archivos en el directorio *fs* del proyecto; serán empaquetados al compilar y colocados dentro del dispositivo al grabar el firmware.

Operación

Luego de compilado el código (`mos build`) y grabado el microcontrolador (`mos flash`) mediante *mos tool*, observaremos en el log la dirección del Access Point y el nombre.

```
[Apr 21 18:39:50.935] esp32_wifi.c:450        WiFi AP: SSID myESP_807A98, channel 6
[Apr 21 18:39:51.802] esp32_wifi.c:507        WiFi AP IP: 192.168.4.1/255.255.255.0 gw [...]
[Apr 21 18:39:51.813] mgos_http_server.c:343  HTTP server started on [80]
[Apr 21 18:39:51.880] init.js:6               ### init script started ###
```

Por comodidad, hemos empleado el servidor web para utilizar el transporte HTTP, podemos reemplazarlo o utilizarlo junto a otro; en otros Comentarios Técnicos iremos desarrollando estos temas. También por comodidad hemos utilizado *curl* para las pruebas.
Entonces, nos conectaremos a esa red y luego pedimos el URL correspondiente a una RPC en esa dirección, por ejemplo: *http://192.168.4.1/rpc/FS.List*, y no obtendremos respuesta. Si utilizamos un navegador, observaremos el clásico pedido de autorización.

```
$ curl http://192.168.4.1/rpc/FS.List
$
```

Si en cambio agregamos las credenciales de un usuario válido, por ejemplo: *http://**bob:hello@**192.168.4.1/rpc/FS.List*, obtendremos la respuesta deseada:

```
$ curl --digest http://bob:hello@192.168.4.1/rpc/FS.List
["init.js","api_net.js","api_i2c.js",[...]]
```

Como todos los transportes requieren autorización, si intentamos utilizar *mos tool* recibiremos un error:

```
$ mos call FS.List
Using port /dev/ttyUSB0
Error:  /src/go/src/github.com/mongoose-os/mos/cli/rpccreds/rpc_creds.go:53:  Failed  to  get
username and password: wrong RPC creds spec
[...]
```

para poder operar necesitaremos proveer las credenciales de un usuario válido:

```
$ mos --rpc-creds=bob:hello call FS.List
```

En los archivos auxiliares hemos incluido además una RPC en mJS (el JavaScript de Mongoose-OS, en *init.js*) y una en C (en el directorio *src*), las mismas son simples ejemplos mìnimos de la estructura necesaria para su confección, donde además podemos observar que no requerimos realizar nada respecto a la autorización. Es posible accederlas como `Do.S` y `Do.C` , respectivamente.
En la nota de aplicación CAN-106 encontraremos un ejemplo más afinado del uso de las RPC en ambos lenguajes de programación.

Digest Authentication en detalle

Como comentamos, al pedirse la ejecución de una RPC el dispositivo devuelve un mensaje que es un objeto JSON; dentro de dicho objeto se envía otro objeto JSON (pero como texto) que provee la información necesaria para proceder con el handshake de autorización. El cliente debe entonces realizar una serie de hashes MD5 con la información provista en dicho objeto y las credenciales del usuario, el cual será enviado volviendo a solicitar la RPC deseada. Veamos un ejemplo:

cliente: `{"src":"mos-1588871456","id":1274131828662,"method":"FS.List"}`

dispositivo: `{"id":1274131828662,"src":"esp32_807A98","dst":"mos-1588871456","error":`
`{"code":401,"message":"{\"auth_type\": \"digest\", \"nonce\": 100, \"nc\": 1, \"realm\": \"myESP\"}"}}`

cliente: `{"src":"mos-1588871456","id":1274131828662,"method":"FS.List","auth":`
`{"realm":"myESP","username":"bob","nonce":100,"cnonce":764787733,"response":"103683d2fa1a1d7db617fe537c8d5eb6"}}`

dispositivo: `{"id":1274131828662,"src":"esp32_807A98","dst":"mos-1588871456","result":`
`["init.js","api_net.js","api_i2c.js", [...]]}`

Observando el diálogo podemos observar las unidades detalladas en el parrafo anterior.
El campo *cnonce* (client nonce) es un contenido aleatorio generado por el cliente. El contenido del campo *response* se genera de acuerdo a la especificación del proceso, como podemos aprender en Wikipedia[1]. En general hay parámetros que en una aplicación web son enviados por el servidor, que aquí están fijos. Algunos detalles de la implementación son:

- *qop = auth*
- dado que en la mayoría de los transportes no existen algunos campos como *method* o *uri*, se utiliza *dummy_method* y *dummy_uri* en su lugar[2].

1 https://en.wikipedia.org/wiki/Digest_access_authentication
2 … tomamos la información del código fuente de *mos tool*, aquí:
https://github.com/mongoose-os/mos/blob/master/common/mgrpc/mgrpc.go

El procedimiento de hashing es el siguiente (hemos señalado las variables en *bastardilla*):

HA1 = MD5(*username*:*realm*:*password*)
HA2 = MD5(dummy_method:dummy_uri)
RESPONSE = MD5(*HA1*:*nonce*:*nc*:*cnonce*:*auth*:*HA2*)

Por ejemplo en Python:

```
import json
import hashlib
import random

username = "bob"
password = "hello"

[ recibimos message ]

    msg = json.loads(message)
    msgid = msg["id"]
    if "error" in msg:                                  # mensaje de error luego de haber hecho un pedido
        error = msg["error"]
        hint = json.loads(error["message"])             # extraemos el objeto JSON con la info necesaria
          if error["code"] == 401:                      # código 401, “unauthorized”
            method = "FS.List"                          # lo que queremos pedir
              respmsg = dodigest(msgid, method, hint)   # hacemos el cálculo del Digest
[ enviamos respmsg ]
[...]

def dodigest(msgid, method, hint):
        realm = hint["realm"]                           #obtenemos realm, nonce y nc
        nonce = hint["nonce"]
        nc = hint["nc"]
        cnonce = str(random.randint(0, 2E64-1))         # generamos un cnonce arbitrario y aleatorio, de manera simple
        str1 = username + ":" + realm + ":" + password  # generamos el primer hash
        ha1 = hashlib.md5(str1.encode())
        str2 = "dummy_method:dummy_uri"                 # generamos el segundo hash
        ha2 = hashlib.md5(str2.encode())
        rstr = ha1.hexdigest() + ":" + str(nonce) + ":" + str(nc) + ":" + cnonce + ":auth:" + ha2.hexdigest()
        response = hashlib.md5(rstr.encode())           # generamos el hash final
        respmsg = {                                     # armamos el objeto con la respuesta
            "id": msgid,
            "method": method,
            "auth": {"realm": realm,
                     "username": username,
                     "nonce": nonce,
                     "cnonce": cnonce,
                     "response": response.hexdigest()
            }
        }
        return respmsg
```

Desarrollaremos estos ejemplos en mayor detalle cuando abordemos otros transportes, en otros Comentarios Técnicos.