Comentario técnico: CTC-092
Componente: **TLS en servidor web con ESP32 y Mongoose-OS**
Autor: Sergio R. Caprile, Senior R&D Engineer

Revisiones	Fecha	Comentarios
0	27/04/20	

Mongoose-OS incorpora un servidor HTTP con soporte TLS. Esto nos permite por un lado que quienes se conectan a él puedan verificar que se trata de ese servidor; y por otro nos da una herramienta para autenticar la identidad de esas conexiones, es decir, que realmente sean usuarios autorizados. La autenticación se realiza por certificados X.509, y luego de realizada se derivan claves que permiten encriptar la información que viaja. Esto significa que lo que se envía no puede ser observado con un sniffer y la única forma de impersonar a una entidad es mediante la posesión del certificado y la clave privada de la que se deriva.

TLS

TLS (Transport Layer Security) permite autenticar mutuamente ambas entidades en una conversación y mantener la integridad de la misma. Se trata de un esquema de autenticación más complicado, con un ente certificador, quien genera las claves y certificados que se utilizan en el proceso. Dado que TLS se deriva de SSL, el esquema original (Secure Sockets Layer), es común que la nomenclatura refiera a este nombre.
La operación más simple y tal vez la más utilizada es la autenticación del servidor, operación que sucede cuando por ejemplo nuestro navegador solicita en Internet una página mediante HTTPS (HTTP Secure). Antes de proceder a la sesión HTTP, se realiza el handshake que permite la validación de autenticidad y el establecimiento de la conexión segura[1]. El navegador conoce una serie de entes certificadores (tiene sus certificados) y puede validar mediante la firma al sitio solicitado cuando éste "le presenta sus credenciales".
La otra operación posible es realizada en muchas redes IoT, cuando un dispositivo se conecta "a la nube". En esta oportunidad no sólo se valida el certificado del servidor sino que éste a su vez valida el certificado del dispositivo, es decir, se establece una doble validación.
El proceso de validación en detalle depende del método acordado entre las contrapartes, aunque a grandes rasgos consiste en enviar una cierta información basada en la encripción con la clave pública del receptor, quien sólo puede descifrarla y así contestar si posee la clave privada correspondiente.
Si bien la autenticación es un proceso lento que involucra clave pública/privada (Public Key Cryptography), la comunicación transcurre encriptada por una clave simétrica derivada.
Como insinuamos al inicio, toda la seguridad del esquema radica en la imposibilidad de impersonar al ente certificador (Certification Authority), cuya clave privada es celosamente guardada y protegida.

Configuración

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.

1 "segura" en el sentido de que se identifica a la contraparte y se cuida la integridad de la información que viaja

```
libs:
  - origin: https://github.com/mongoose-os-libs/http-server        # incorpora el servidor web
config_schema:
  - ["http.listen_addr", "443"]       # Puerto donde atiende el servidor web
  - ["http.ssl_ca_cert", "ca.crt"]    # CA certificate (sólo para autenticación mutua)
  - ["http.ssl_cert", "server.crt"]   # Certificate file
  - ["http.ssl_key", "server.key"]    # Private key file
```

La autorización es para todo el contenido estático del servidor[2], es decir, las páginas HTML que colocaremos en el directorio *fs* del proyecto; y esta vez sí afecta a los RPC que sean accedidos por este transporte, dado que el layer TLS se establece antes de la comunicación HTTP.

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 24 17:39:25.351] esp32_wifi.c:450        WiFi AP: SSID myESP_807A98, channel 6
[Apr 24 17:39:25.421] esp32_wifi.c:507        WiFi AP IP: 192.168.4.1/255.255.255.0 gw [...]
[Apr 24 17:39:25.433] mgos_http_server.c:343  HTTP server started on [443]
[Apr 24 17:39:25.508] init.js:6               ### init script started ###
```

Con cualquier dispositivo con capacidad de conexión a una red WiFi y un navegador nos conectaremos a esa red y luego pedimos al navegador la página principal de esa dirección: *http://192.168.4.1/*, o un listado mediante RPC (por ejemplo). A fines ilustrativos, y dada la notable diferencia entre los detalles de la diversa gama de navegadores, realizaremos las pruebas mediante herramientas de línea de comandos como *curl* o *mos tool*.
Dado que conseguir un certificado válido, por razones obvias, no es algo trivial, hemos procedido a generar los nuestros. Esto es muy similar a lo que ocurre en la realidad de una pequeña empresa, por lo que iremos desarrollando los pasos a seguir.
A los fines prácticos, reutilizamos los certificados provistos en el CTC-090, con algunos detalles. La generación y manejo de claves no es algo trivial y escapa a los alcances de este texto.

Simple autenticación

De esta forma autenticamos al servidor, es decir, podemos confiar en que es el que esperamos que sea, pero el servidor no tiene idea de quién podemos llegar a ser nosotros.
Para que nuestro navegador pueda validar el certificado del web server en el ESP32, debe conocer al ente certificador. Esto obviamente no ocurre, por lo que lo primero que observaremos es esta situación.
Por ejemplo `curl` nos responde

```
Peer's Certificate issuer is not recognized
```

y Firefox reporta el error

```
SEC_ERROR_UNKNOWN_ISSUER
```

Aquí en algunos casos podemos simplemente aceptar la identidad y proseguir con las pruebas. Dado que esto desvirtúa completamente la intención de este tipo de conexión y de este CTC, disponemos del certificado de la CA y lo incorporamos al navegador.
El paso siguiente es comprobar que el nombre del servidor se corresponda con el que figura en el certificado. Esto tampoco suele suceder en un ámbito de pruebas.
Por ejemplo `curl --cacert ca.crt` nos responde

```
Unable to communicate securely with peer: requested domain name does not match the
server's certificate.
```

y Firefox reporta el error

```
SSL_ERROR_BAD_CERT_DOMAIN
```

2 Otras configuraciones son posibles, aunque tienen requerimientos de desarrollo más elevados.

En este punto podemos una vez más aceptar el certificado, y proseguir. En la realidad, si todo está bien configurado y el servidor es el correcto, conectándonos al nombre correcto llegaremos al servidor correcto; para que esto suceda deberemos configurar la resolución de nombres de modo que nuestra computadora resuelva el nombre que figura en el CN (Common Name) del certificado (*changeme*) a la dirección IP que el dispositivo toma (lo cual requiere administrar servidores DNS y DHCP). Por esta vez lo hemos simulado cargándolo provisoriamente en el archivo local de resolución de nombres (generalmente */etc/hosts* en GNU/Linux). De no poder hacer esto, sólo nos queda aceptar, que en lenguaje *curl* corresponde a agregar la opción *-k* (`--cacert ca.crt -k`).

Doble autenticación

De esta forma autenticamos tanto al servidor como al cliente, es decir, ahora el servidor sabe que deberíamos ser quien nuestro certificado dice que somos. Este doble proceso requiere bastante procesamiento y lo notaremos en el tiempo de establecimiento de la conexión.
Para que el ESP32 pueda validar el certificado del navegador, hemos configurado el parámetro correspondiente (*http.ssl_ca_cert*)y alojado el certificado de la CA en el directorio *fs* del ESP32.
Para que nuestro navegador pueda identificarse de forma que el web server en el ESP32 lo pueda validar, debe poseer un certificado y clave privada generados por el ente certificador. Si esto no ocurre, observaremos que no se establece la conexión.
Por ejemplo `curl` nos responde

```
NSS: client certificate not found (nickname not specified)
```

en el log de Mongoose-OS leemos

```
TLSv1 client has no certificate
```

y Firefox reporta el error

```
PR_CONNECT_RESET_ERROR
```

A continuación agregamos al pedido de conexión el certificado y clave de cliente, los cuales están en el mismo directorio desde el que realizamos el pedido :

```
curl --cacert ca.crt --cert ./sandboxclient.crt --cert-type PEM --key
./sandboxclient.key  https://changeme/
```

De igual modo podemos pedir un RPC:

```
curl --cacert ca.crt --cert ./sandboxclient.crt --cert-type PEM --key
./sandboxclient.key  https://changeme/rpc/FS.List
```

Lo mismo también podemos hacerlo con *mos tool*, lo cual nos permite tener una amplia gama de posibilidades remotas sin altos riesgos de seguridad:

```
mos --cert-file sandboxclient.crt --key-file sandboxclient.key --port
https://changeme/rpc call FS.List
```

En el caso de un navegador, importamos el certificado PKCS12. Si nos pide un password, no lo hemos incluido, simplemente lo dejamos vacío.
Los certificados que empleamos para el CTC-090 están firmados con SHA-1 y no son aceptados por la configuración de *mbedtls* de Mongoose-OS. A tal fin, incorporamos una nueva versión del certificado de cliente para este CTC, la clave es la misma. Si por algún motivo se nos mezclan y todo “mágicamente deja de funcionar”, observaremos en el log la frase “`The certificate is signed with an unacceptable hash`”.