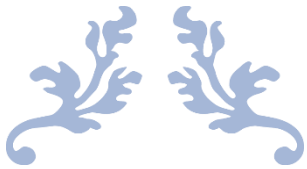


PRÁCTICA 1

DIEGO DORADO GALÁN

11 DE OCTUBRE DE 2022

UCLM

Seguridad en Redes

FUNCIONAMIENTO

Nos encontramos con dos archivos binarios ‘alice’ y ‘bob’. Analizando de forma individual los archivos podemos encontrar cierta información pero de poca utilidad para entender su funcionalidad. Por ejemplo, son ejecutables binarios LSB (Least significant Bit, es decir, Little Endian) y también son ficheros “stripped”, es decir, no contienen ningún tipo de información sobre la compilación (es decir, no se compiló con la opción -g). Además, mediante el comando ldd observamos que trabaja con distintas librerías como son las siguientes:

```
kdt23@kdt23:~/4º/Seguridad$ ldd --verbose bob
        linux-vdso.so.1 (0x00007ffe03291000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f046581b000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f04655f3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f0465835000)

        Version information:
        ./bob:
                libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
                libpthread.so.0 (GLIBC_2.3.2) => /lib/x86_64-linux-gnu/libpthread.so.0
                libpthread.so.0 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libpthread.so.0
        /lib/x86_64-linux-gnu/libpthread.so.0:
                libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
        /lib/x86_64-linux-gnu/libc.so.6:
                ld-linux-x86-64.so.2 (GLIBC_2.2.5) => /lib64/ld-linux-x86-64.so.2
                ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
                ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
```

Además, indagando con el comando *strings* obtenemos algunas rutas de lo que parece ser el ordenador del profesor, pero no sé qué hacer con ello. También observamos como el código fuente de ambos programas está escrito en el lenguaje de programación ‘Go’.

```
kdt23@kdt23:~/4º/Seguridad$ strings -a alice | grep alice
path    example.com/cmd/alice
/home/cleto/uni/secnet/lab/p1/src/cmd/alice/main.go
kdt23@kdt23:~/4º/Seguridad$ strings -a bob | grep bob
path    example.com/cmd/bob
/home/cleto/uni/secnet/lab/p1/src/cmd/bob/main.go
```

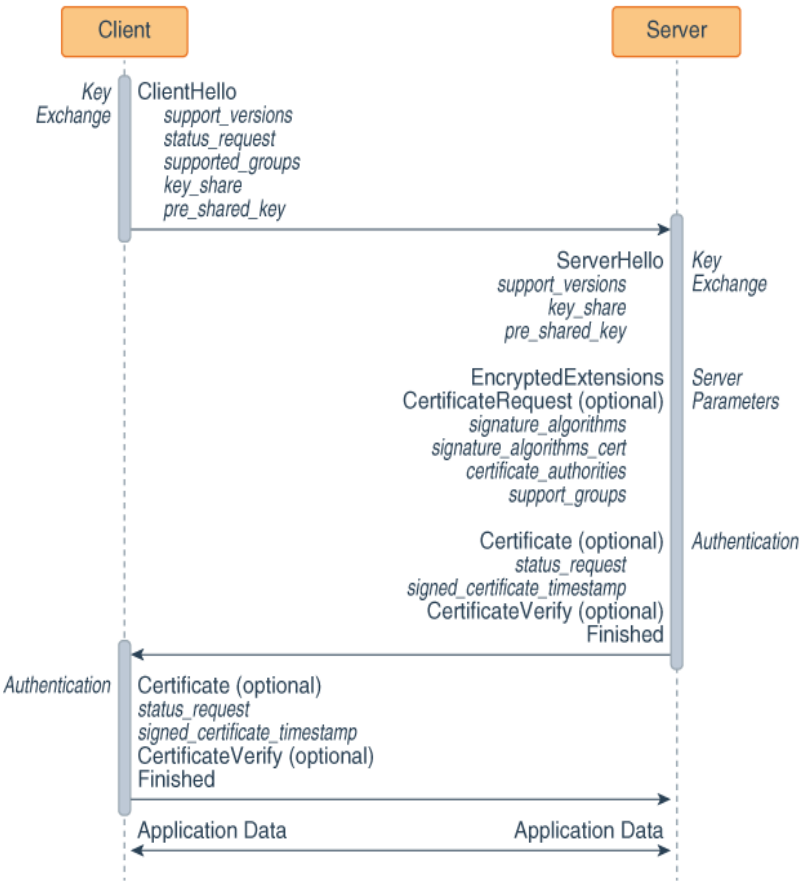
Mediante *strace* observamos las distintas llamadas al sistema que realizan los programas, pero no he podido destacar nada de ellos.

Si ejecutamos ambos archivos de forma simultánea, aparentemente no pasa nada ya que no muestra por pantalla. Pero al finalizar el script ‘alice’ automáticamente lo hacía también el script ‘bob’. Esto me hizo pensar que ‘alice’ enviaba un flag FIN+ACK que cerraba una comunicación. De tal forma, determiné que el funcionamiento de ambos scripts podría simular una comunicación, tal como también nos invitaba a pensar la clásica analogía de Alice y Bob de Ron Rivest.
(https://es.wikipedia.org/wiki/Alice_y_Bob)

Para comprobarlo, decidí analizar el tráfico de flujo de datos al ejecutar simultáneamente ambos ficheros. La primera herramienta que se me vino a la cabeza fue Wireshark. Es una herramienta que tengo algo olvidada, por lo que seguramente se me escapé algún dato de interés.
Al ejecutar simultáneamente dichos scripts y capturar el tráfico, vi un protocolo que estaba relacionado con SSL como es TLS (mejora o evolución de este). Además, en la

misma información del paquete encontramos mensajes como ‘Client hello’ o el más característico ‘Change Cipher Spec’. Esto hace alusión a un hand-shake del protocolo TLS.
Antes de entrar en mayor detalle con TLS, también nos llama la atención que al ejecutar ‘alice’ observamos como inicia una comunicación DNS con un servidor web (gitlab.com), con cuya IP de respuesta se utiliza para iniciar una comunicación TCP al puerto 443.
Al ejecutar ambos ficheros en la interfaz *lo* (loopback) observamos también una comunicación TCP.
Por tanto, podemos determinar que el funcionamiento principal será que ‘alice’ solicita a un servidor web en *gitlab.com* cierta información que transmitirá a ‘bob.

Antes de indagar en la comunicación entre ‘alice’ y ‘bob’ debemos profundizar en el protocolo TLS. El proceso del handshake TLS son una serie de mensajes intercambiados entre un cliente y un servidor para establecer una comunicación segura. A continuación expondré las fases principales de este mecanismo para establecer una conexión adjuntando las capturas obtenidas para ejemplificarlo.
Gráficamente, la secuencia de mensajes es la siguiente:



En primer lugar, nos encontraremos con el mensaje ‘client hello’, donde el cliente inicializa el “hand-shake” enviando un saludo al servidor. Este mensaje incluirá la versión TLS que soporta el cliente, los algoritmos de cifrado permitidos, los algoritmos o métodos de compresión soportados y un string de bytes random que aparecen como “client random” (grosso modo, es la forma que tiene el cliente de determinar si el servidor con

el que contacta es legítimo y no es, por ejemplo, un intermediario que intenta interceptar la comunicación simulando ser el servidor):

```
▾ TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 275
  ▾ Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 271
      Version: TLS 1.2 (0x0303)
      Random: c4bbae3366924e21d55f8e0e921ba15f99be535fb3f67c3f35327752c2281d08
      Session ID Length: 32
      Session ID: f6eb64f680811d64881631a60f90b3401db678e009f9bf6e9ab753e128764fe4
      Cipher Suites Length: 38
    ▸ Cipher Suites (19 suites)
      Compression Methods Length: 1
    ▾ Compression Methods (1 method)
        Compression Method: null (0)
      Extensions Length: 160

  Extensions Length: 160
▸ Extension: server_name (len=15)
▸ Extension: status_request (len=5)
▸ Extension: supported_groups (len=10)
▸ Extension: ec_point_formats (len=2)
▸ Extension: signature_algorithms (len=26)
▸ Extension: renegotiation_info (len=1)
▸ Extension: application_layer_protocol_negotiation (len=14)
▸ Extension: signed_certificate_timestamp (len=0)
▸ Extension: supported_versions (len=9)
▾ Extension: key_share (len=38)
    Type: key_share (51)
    Length: 38
  ▾ Key Share extension
      Client Key Share Length: 36
    ▾ Key Share Entry: Group: x25519, Key Exchange length: 32
        Group: x25519 (29)
        Key Exchange Length: 32
        Key Exchange: dbd99c705fd4f9c22f663b52d3182abc3cab1e18111fa9c96306ec4cfc97bb52
```

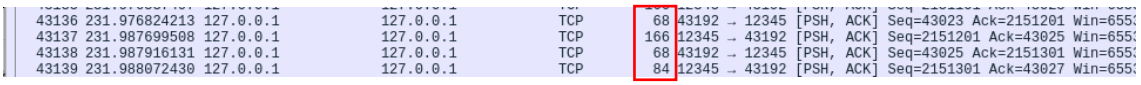
En segundo lugar, en respuesta al mensaje “Client hello”, el servidor envía “Server hello” y el protocolo de acuerdo de clave elegido si es compatible con TLS1.3.

```
    TCP payload (219 bytes)
▾ Transport Layer Security
  ▾ TLSv1.3 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 128
    ▾ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 124
        Version: TLS 1.2 (0x0303)
        Random: 3aaba55580604a48e820cc5dd641299019c72dd67ebd36854de646397af736f0
        Session ID Length: 32
        Session ID: ba5f8b1b2ceccbcc26dc42531e9dd73d9a47860ec7b96162f80b35170ede198d
        Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
        Compression Method: null (0)
        Extensions Length: 52
      ▾ Extension: pre_shared_key (len=2)
          Type: pre_shared_key (41)
          Length: 2
        ▾ Pre-Shared Key extension
            Selected Identity: 0
      ▾ Extension: key_share (len=36)
          Type: key_share (51)
          Length: 36
        ▾ Key Share extension
          ▾ Key Share Entry: Group: x25519, Key Exchange length: 32
              Group: x25519 (29)
              Key Exchange Length: 32
              Key Exchange: 90c5bd07334a7c5955e7075d64ad7db74eaeb9dbb9e978db10eae4579453ec75
      ▾ Extension: supported_versions (len=2)
          Type: supported_versions (43)
          Length: 2
          Supported Version: TLS 1.3 (0x0304)
        [JA3S Fullstring: 771,4866,41-51-43]
        [JA3S: f590053ff246338aff7c203dbe7164d6]
  ▸ TLSv1.3 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
  ▸ TLSv1.3 Record Layer: Application Data Protocol: http-over-tls
```

Este mensaje contiene el ID de Sesión, el random number de 28 bytes, el método de algoritmo de cifrado, la clave compartida del servidor, su certificado y el mensaje de "Server Finished", con el que empezará a cifrar la información.
A partir de aquí, el servidor empezará a enviar la información cifrada.

En el tercer paso, el cliente verifica el certificado enviado por el servidor, generará las claves simétricas y envía los mensajes de "Change Cipher Spec" y "Client Finished". A partir de aquí, cliente y servidor empiezan la comunicación con mensajes cirados como son los distintos mensajes observados en Wireshark en protocolo TLS1.3 con información Application Data.

Pasamos ahora a explicar el tráfico de red en la interfaz loopback. Analizando las tramas TCP, encontramos que el tráfico en si es curioso, ya que mezcla algunos elementos aparentemente cifrados con mensajes de OK. La explicación es sencilla, es la emisión de las distintos paquetes por parte de 'alice' y los OK son los mensajes de confirmación por parte de 'bob'. Los puertos origen y destino son 12345 y 43192, que aparentemente no tienen mayor relevancia y son simplemente puertos TCP.
Si observamos el tamaño de las tramas nos percatamos de que existen principalmente dos tamaños de trama: las tramas de OK de 68 bytes, y las tramas con el mensaje cifrado de 166 bytes. Sin embargo, la última trama de mensaje cifrado tiene un tamaño distinto: 84 bytes.



Esto, me hizo pensar que estábamos ante distintos bloques de tamaño fijo de un cifrado simétrico por bloques.
Observando las tramas me quede algo bloqueado ya que no sabía que hacer exactamente con ellas. Decidí tratar de examinar y buscar ciertos patrones entre ellas hasta que me llamó especialmente la atención que muchos símbolos '{' iban seguidos en multitud de veces de 'm', y algunos caracteres se repetían con frecuencia (podrían ser las vocales cifradas) lo que me hizo sospechar que nos encontrábamos ante un Cifrado César o Cifrado Vigenère, ya que son los métodos de cifrado por sustitución más sencillos y que se caracterizan principalmente por seguir un patrón.

```
i{(lw{..kw{i{({m(pittiv(kwv{}uiliumv|m(mv(L}tkqvmi4(xwzy}m(mv({mz(pmzuw{i(vqvOKo}
vi..tm(qo}iti4(.(mv(ti(j}mvi(niui4(xwki{(tm(ttmoiv6(a(xizi(kwvkt}qz(kwv(|
wlw4...w(quioqvw(y}m(|wlwOK(tw(y}m(lqow(m{(i{..4({qv(y}m({wjzm(vq(nit|
m(viliC(...x..v|wti(mv(uq(quioqvikq..v(kwuw(ti(lm{mw4(i{.OK.
(mv(ti(jmttm.i(kwuw(mv(ti..xzqvkqxitqlil4(.(vq(ti(ttmoi(Mtmvi4(vq(ti(itkiv.i(T}
kzmkqi4(vq(w|zi(ito}OKvi..lm(ti{(niuw{i{(u}rmzm{(lm(ti{(mlilm{(xzm|..zq|
i{4(ozqmoi4(j..zjizi(w(ti|qvi6..a(lqoi(kili(}vw(twOK(y}m(y}q{qmzmC(y}m({q(xwz(m{|
w(n}mzm(zmxzmpmvlqlw(lm(tw{..qovwziv|m{4(vw({mz..(ki{|qoilw(lm(tw{(zqo}
OKzw{w{6....5Lqow(y}m(mv(|wlw(|qmvm(~}m{|zi(umzkml(zi...v(5zm{xwvlq..([ivkpw54(.
```

Por ello, decidí realizar un script que tratase de descifrar un posible texto César. Pero necesitaba una clave (número de rotaciones), por lo que con simple prueba y error determiné que el número de rotaciones o de clave eran 8. Adjunto el script que me ayudó a resolver y descifrar el texto, aunque no es exacto debido a que no trabaja de forma completa con la tabla ASCII, por lo que algún carácter puede variar. Adjunto también un código simple que elimina los mensajes 'OK' para así poder pasarlo a una página web que realice el descifrado (como por ejemplo https://www.dcode.fr/caesar-cipher).

El código cifrado que se transmiten ambos ficheros es **El Quijote**.

Hay ciertos temas en los que no he podido profundizar pero de los que quiero hablar:

Respecto a la conexión de 'alice' con GitLab, existen métodos para entrometerse en la comunicación y actuar como un observador que intercepta o escucha la comunicación. Estos ataques son comúnmente conocidos como Man-in-the-middle (MIDT) o Sniffing y son aplicables a todo tipo de protocolos, y por supuesto a HTTPS.
Para realizar un ataque de estas características es recomendable el uso de algún software especializado. En mi caso únicamente conozco Burp Suite ya que es el más popular y se utiliza con frecuencia en ánalisis o ataques de servicios/aplicaciones web y creo que sería útil para el caso que nos concierne.
Como herramientas para realizar Sniffing nos encontramos, por ejemplo, 'WolfSSL' o 'SSLSniff' que pueden llegar a resultar útiles siempre y cuando se realicen bajo un entorno controlado.

Sinceramente no me ha dado tiempo a intentar analizar el tráfico entre 'alice' y GitLab ya que no estoy familiarizado del todo con esta herramienta, pero me gustaría haber tenido capacidad para ello. A continuación adjunto una guía sobre cómo realizar un análisis del protocolo TLS mediante Burp Suite:

https://portswigger.net/burp/documentation/desktop/tools/proxy/using

Además, adjunto también un repositorio con otros tipos de ataques MITM contra TLS:

https://github.com/tanc7/Practical-SSL-TLS-Attacks/blob/master/TLS-mitm-methods.md

En cuanto a la técnica de suplantar la identidad de unos de los integrantes de la comunicación (Spoofing) creo que sería complicado de realizar ya que un impostor difícilmente podrá obtener la clave privada utilizada en la comunicación asimétrica. Esto se discute en el siguiente foro:

https://security.stackexchange.com/questions/20059/is-spoofing-a-ca-signed-certificate-possible

HERRAMIENTAS

- Wireshark
- Visual Studio Code
- Comandos mencionados para análisis de ficheros (strace, strings, ldd)
- Página de descifrado César: https://www.dcode.fr/caesar-cipher
- Scripts realizados: https://github.com/23kdt/SeguridadRedes

En dicho repositorio adjunto también una captura de Wireshark, 'caesar.py' (algoritmo de Descifrado César) y otros archivos de utilidad.

ERRORES COMETIDOS

Mi principal error fue básicamente de enfoque y de precipitación. Erróneamente, pensaba que el objetivo de la práctica era obtener el código fuente de esos archivos, creyendo que era el propio código fuente el que estaba cifrado .

Me aventuré inconscientemente a tratar de analizar ambos ficheros sin ni si quiera saber que tipo de fichero era. Por ello, traté de evaluar el código fuente de un archivo binario algo que, como es lógico, fue ridículo. Dicho código fuente no era legible y pensaba que estaría cifrado en base64 o un esquema de codificación similar, algo que salvando las distancias sería similar (ya que el contenido de las tramas TCP si que estaría cifrado pero mediante Cifrado César). De hecho realicé un script básico para tratar de descifrar el código fuente haciendo uso del módulo '*Fernet'* de la librería '*Cryptography'*, para así descifrar un cifrado simétrico probando contraseñas simples e incluso pensando que debía realizar un algoritmo de fuerza bruta para ello.

Tras todo ello, decidí releer el enunciado del problema buscando otro enfoque. Rápidamente, se me ocurrió asociar los nombres de los scripts con la clásica terminología usada en la ejemplificación de comunicaciones entre dos entidades, tal y como ya he mencionado. Dando permisos de ejecución a los ficheros llegué al procedimiento que explicado con anterioridad.

BIBLIOGRAFÍA E INFOGRAFÍA WEB DE REFERENCIA

https://www.cs.usfca.edu/~ejung/courses/686/lectures/10SSL.pdf

https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/

https://security.stackexchange.com/questions/157684/why-does-the-ssl-tls-handshake-have-a-client-random

https://www.ibm.com/docs/en/sdk-java-technology/8?topic=handshake-tls-13-protocol

https://security.stackexchange.com/questions/20059/is-spoofing-a-ca-signed-certificate-possible