SSII

PSI-TRANSIT

El presente informe documenta los procedimientos seguidos y por consiguiente los resultados obtenidos para resolver las cuestiones del proyecto PSI-Transit, el cual está destinado a garantizar la integridad de los mensajes emitidos por una entidad en una comunicación cliente-servidor.

Fco. Javier Delgado Vallano

04 de Marzo del 2015

ÍNDICE

Historial de versiones..... Pág. 2
Planificación..... Pág. 2
Introducción..... Pág. 3
Tarea 1..... Pág. 4
Tarea 2..... Pág. 11
Reto..... Pág. 12
Conclusión..... Pág. 14

HISTORIAL DE VERSIONES

Versión	Fecha	Descripción
1.0	04/03/2015	Creación y estructuración del documento, desarrollo de planificación e introducción.
1.1	05/03/2015	Desarrollo de planificación, introducción y tarea 1.
1.2	08/03/2015	Desarrollo de tarea 2 y de la conclusión.
1.3	12/03/2015	Desarrollo de reto y finalización del documento.

PLANIFICACIÓN

Tareas	Tiempo estimado	Tiempo total	Tiempo desviación
Informe	6 h.	h.	h.
Tarea 1	8 h.	9 h.	1 h.
Tarea 2	1 h.	0.5 h.	0.5 h.
Reto	4 h.	6 h.	2 h.

INTRODUCCIÓN

PSI-Transit es un proyecto que busca el garantizar la integridad de los mensajes emitidos por una entidad en una comunicación cliente-servidor. Para ello se han barajado las opciones de *Python* y *Java* para desarrollar una aplicación, debido a su numerosa lista de librerías y gran comunidad que tienen a sus espaldas, pero finalmente fue el primero el elegido debido a ventajas como son:

- Código dinámico.
- Mayor eficiencia con respecto a *Java*.
- No necesita clases predefinidas, por lo que el número de líneas de código se ve reducido.

Mediante esta aplicación se intenta garantizar dicha integridad haciendo uso de la técnica MAC (código de autenticación de mensaje). Esto se hace posible mediante la comparación en el receptor del hash calculado por el mensaje en cuestión y una clave secreta compartida entre el cliente y el servidor, con el código hash del conjunto del mensaje recibido y la clave secreta, nuevamente calculado por el receptor.

En caso de que el hash calculado por el receptor no sea igual que el enviado por el emisor, el programa lo notificará al cliente por consola y en el servidor creará un archivo log en el que se almacenará dicho fallo. En cambio, si esto no sucede, se le notificará al cliente de que se ha conservado correctamente la integridad en la transacción.

TAREA 1

La primera tarea a realizar consiste en desarrollar un verificador de la integridad para la transmisión de mensajes en conexiones cliente-servidor.

Para comenzar esta tarea se han realizado dos *scripts*, "cliente.py" y "servidor.py", los cuales serán tanto emisor como receptor respectivamente. El cliente será el encargado de emitir mensajes con la siguiente estructura:

Nombre de cuenta origen, Nombre de cuenta destino, cantidad transferida

Estos mensajes serán enviados junto con un código hash resultado de la técnica de MAC, que devuelve esto tras recibir dicho mensaje y una clave secreta compartida por ambos de la cual hablaremos más adelante. Mientras que el servidor se encargará de recibir tanto el mensaje como el código hash y posteriormente realizar la verificación de si se ha violado la integridad en la transmisión.

Además de esto se incluye a dicho código la fecha y hora sin caracteres especiales, para así evitar los ataques de "*replay*", de modo que en el servidor se pueda verificar además de que este código no se vea alterado, que no se ha recibido otro igual con la misma fecha ya que esta no se puede repetir.

A continuación se muestra el código del *script* que toma la función de cliente, el cual tras haberse conectado a un socket (el del servidor) previamente especificado, envía al servidor el mensaje junto con su correspondiente código y antes de cerrar la conexión recibe el mensaje que el servidor ha recibido junto con una notificación referente a la integridad.

```
import socket
import configparser
import hmac
import pickle
import datetime

config = configparser.ConfigParser()
config.read("config.cfg")

time = str(datetime.datetime.now())
time = time.replace(" ","")
time = time.replace("-","")
time = time.replace(":","")
time = time.replace(".","")
```

```
opciones = {
    1 : 'md5',
    2 : 'sha1',
    3 : 'sha256'
}

algoritmo = config.get("S1", "Hash")

if algoritmo == opciones.get(1):
    cod = 'md5'
elif algoritmo == opciones.get(2):
    cod = 'sha1'
else:
    cod = 'sha256'

# Creando un socket TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


# Conecta el socket en el puerto cuando el servidor esté escuchando
server_address = (config.get("S1", "ip"), int(config.get("S1", "puerto")))
print ('conectando a ' + server_address[0] + ' y puerto ' +
str(server_address[1]))
sock.connect(server_address)

try:

    # Enviando datos
    message = config.get("S1", "cuentaOrigen") + "," + config.get("S1",
"cuentaDestino") + "," + config.get("S1", "cantidad")
    print ('enviando..\n ' +  message)

    keyword = config.get("S1", "key").encode('utf-8')

    mensaje = message.encode('utf-8')

    digest_maker = hmac.new( keyword, mensaje , cod)

    digest_maker.update(mensaje)

    digest = digest_maker.hexdigest() + str(time)

    msgs = [message,digest]

    sock.sendall(pickle.dumps(msgs))

    # Buscando respuesta
    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(1024)
        d = pickle.loads(data)
        amount_received += len(data)
        print ( 'recibiendo..\n' + str(d))


    del digest_maker
```

```
finally:
    print ('cerrando socket\n\n')
          sock.close()
```

La verificación que realiza el servidor se ejecuta de la siguiente manera:

- En primer lugar, tras haber recibido tanto el mensaje como el código hash en forma de lista, procedemos a calcular de nuevo la MAC con dicho mensaje recibido y la clave secreta.
- A continuación, se realiza la comparación de este código recibido con uno calculado a partir del mensaje que nos ha llegado. Además de esto, el servidor comprueba que no se ha recibido ningún mensaje con la misma fecha que está unida al código. En caso de que el resultado indique que no son iguales se notificará al cliente con un mensaje como el siguiente:

ERROR – integridad violada

Además de esto, se creará un archivo en el que se almacene tanto el momento en el que el resultado fue negativo junto con el mensaje y la notificación pertinente.

En cambio, si dicho resultado es satisfactorio se notificara con:

OK

- Finalmente, el servidor reenvía al cliente el mensaje recibido anteriormente junto con la notificación pertinente.

Con esto conseguiremos evitar los ataques de "*man in the middle*" y de "*replay*". Cabe destacar que el cliente estará informado de lo que el servidor recibe con el envío por parte del servidor a este.

A continuación se muestra el código del *script* que permitirá las funciones del servidor:

```
import socket
import configparser
import pickle
import hmac
import datetime

config = configparser.ConfigParser()
config.read("config.cfg")

outfile = open('integridad.txt', 'w')

opciones = {
    1 : 'md5',
    2 : 'sha1',
    3 : 'sha256'
```

```
}

algoritmo = config.get("S1", "Hash")



if algoritmo == opciones.get(1):
    cod = 'md5'
    tam = 32
elif algoritmo == opciones.get(2):
    cod = 'sha1'
    tam = 40
else:
    cod = 'sha256'
    tam = 64

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Enlace de socket y puerto
server_address = (config.get("S1", "ip"), int(config.get("S1", "puerto")))
print ( 'Empezando a Levantar ' + server_address[0] + ' puerto ' +
str(server_address[1]))
sock.bind(server_address)

# Escuchando conexiones entrantes
sock.listen(1)

recibidos = []

while True:
    # Esperando conexion
    print ('Esperando para conectarse\n')
    connection, client_address = sock.accept()
    outfile = open('integridad.txt', 'a')

    try:
        print ('conexion desde ' + client_address[0])

        # Recibe los datos en trozos y reetransmite
        while True:

            data = connection.recv(1024)

            d = pickle.loads(data)

            recibidos.append(d)

            print ('recibido..\n ' + str(d))

            keyword = config.get("S1", "key").encode('utf-8')

           mensaje = d[0].encode('utf-8')

           code = d[1].encode('utf-8')

            hashing = code[0:tam].decode('utf-8')

            tamCod = len(code)
```

```
            time = code[tam+1:tamCod].decode('utf-8')

            digest_maker = hmac.new( keyword, mensaje , cod)

            digest_maker.update(mensaje)

            digest = digest_maker.hexdigest()

            notifi = "OK

            if (digest != hashing and time in recibidos):

                notifi = "ERROR - Integridad violada"
                out = str(datetime.datetime.now()) + " - " + str(d) + " - " +
notifi + " \n"
                outfile.write(out)
            if data:
                print ('Enviando mensaje de vuelta al cliente')
                d.append(notifi)
                connection.sendall(pickle.dumps(d))

            else:
                print ('No hay mas datos de ' + client_address[0])

                del digest_maker

                break


    except EOFError:
        ret = []

    finally:
        # Cerrando conexion
        connection.close()
              outfile.close()
```

Además de lo anteriormente descrito, el proyecto cuenta con un archivo de configuración llamado “config.cfg”. Este archivo será el encargado de almacenar las variables necesarias para correr el programa siendo de este modo, el socket compartido por el cliente y el servidor, el hash a utilizar, las componentes del mensaje que se van a intercambiar ambos y la clave secreta.

Por tanto, antes de poner en funcionamiento el proyecto debemos dirigirnos a dicho archivo de configuración para definir estos parámetros como podemos observar a continuación.

```
[S1]
hashType: md5, sha1, sha256
hash: sha1
ip: localhost
puerto: 10000
cuentaOrigen: 123123123
cuentaDestino: 456456456
cantidad: 100
key: clave
```

En este archivo podemos observar una única sección y los parámetros a completar para poner en funcionamiento nuestra comunicación:

- "S1":
  - hash: Tipo de *hashing* a usar.
  - ip: Dirección ip del servidor.
  - puerto: Puerto abierto del servidor.
  - cuentaOrigen: Número de cuenta origen.
  - cuentaDestino: Número de cuenta destino.
  - cantidad: Cantidad de dinero a traspasar desde la cuenta origen a la cuenta destino.
  - key: Clave secreta compartida por cliente y servidor.

Por último, centrándonos en la clave secreta y a la vez compartida por el cliente y el servidor, esta debe ser negociada mediante un canal seguro, comunicación vía móvil, video llamada, intercambio de pendrive o mecanismos similares; o incluso y a ser posible en un encuentro en persona. La cuestión es que la negociación de dicha clave debe ser lo más seguro posible ya que gracias a esto podremos asegurar la integridad. También sería conveniente la renovación de dicha clave en periodos cortos de tiempo, además de que esta cumpla las siguientes normas, para así evitar que esta pueda ser descubierta:

- Tener mínimo 10 caracteres.
- Contener letras tanto en mayúsculas como minúsculas.
- Contener al menos dos carácteres numéricos.
- Contener al menos un símbolo como los siguientes: ` ! @ # $ % & *^( ) - = { } [ ] \ : " ; ' < > ? , . /

Como nota última, cabe mencionar que el modo utilizado en este proyecto para compartir la clave entre ambos no sería suficiente ya que esta por ejemplo debería ser cifrada antes de ser recogida del archivo de configuración para así mantenerse más segura y confidencial, también debemos tener en cuenta que para el presente proyecto sólo contamos con un archivo de configuración mediante el cual, cliente y servidor pueden funcionar; cuando en un proyecto real se debería de incluir otro igual, de modo que tanto cliente como servidor puedan configurarse sin tener dependencia alguna.

TAREA 2

La segunda tarea propuesta consiste en aportar un *feedback* a la entidad financiera en la que se aporten posibles mejoras que se puedan realizar al producto entregado, es decir, al verificador de la integridad en la transmisión.

Algunas de las mejoras aplicables pueden ser:

- Desarrollar una interfaz de usuario que permita realizar la configuración inicial del programa de manera más visual, cómoda y sencilla para cualquier usuario.
- Desarrollar un segundo archivo de configuración a partir del que ya está creado para que tanto cliente como servidor cuenten con su propia configuración sin dependencia alguna entre ambos.
- Cifrar los mensajes intercambiados para así mantener la confidencialidad de ambos.
- Modificar el servidor para que se inicie automáticamente con el inicio del sistema.
- Mejorar el nonce añadiéndole un índice aleatorio al final e incluso alterar el orden de las caracteres del código MAC para aumentar de este modo en mayor grado la seguridad.
- Almacenar la clave secreta cifrada y obligar al usuario a cambiarla cada cierto tiempo.
- Eliminar cada cierto tiempo los mensajes almacenados tanto en cliente como en servidor.

RETO

El reto propuesto para el presente proyecto consiste en averiguar la clave utilizada para una MAC determinada:

8b6a017282290f4ec9a26598bbdc93d2d09094f0

De entrada nos aportan la información de que el tamaño de la clave es excesivamente pequeño, sólo de 24 bits. Tras recoger la información aportada se ha llevado a cabo la elaboración de un *script* para demostrar la rapidez y la facilidad con la que se puede descubrir la clave en cuestión. A continuación se muestra el código de dicho *script*, llamado "hack.py":

```
import hmac
import itertools
import codecs
import time

mensaje =
"CuentaOrigen_12345_a_CuentaDestino_67890_la_cantidad_de_3000_euros"

mac = "8b6a017282290f4ec9a26598bbdc93d2d09094f0"

def tipoHash(password):

    tam = len(password)

    if(tam == 32):
        hash = "md5"
    elif(tam == 40):
        hash = "sha1"
    elif(tam == 56):
        hash = "sha224"
    elif(tam == 64):
        hash = "sha256"
    elif(tam == 96):
        hash = "sha384"
    else:
        hash = "sha512"

    return hash
```

```
def hack(code):

    inicio = time.time()

    h = tipoHash(code)

    msg = mensaje.encode('utf_8')

    l = "abcdef1234567890"

    #generar secuencia de 6 caracteres en hexadecimal
    for seq in itertools.product( l , repeat = 6):

        key = "".join(seq)

        #traducimos de hexadecimal a datos binarios
        p=codecs.decode(key ,"hex")

        digest_maker = hmac.new( p , msg , h).hexdigest()

        if (hmac.compare_digest(code,digest_maker)):

            print ("La contraseña es: " + key)
            break

        del digest_maker

    fin = time.time()

    tiempo_total = fin - inicio

    print( "Tiempo de ejecucion : " + tiempo_total )


if __name__ == '__main__':


    hack(mac)
```

Este código básicamente va generando una secuencia de cadenas en hexadecimal, de las cuales una de ellas será la clave en cuestión; y calculando a partir de estas y junto con el mensaje aportado, nuevas MAC.

Posteriormente, esta nueva MAC es comparada con la propuesta, hasta que finalmente encuentra la coincidencia entre estas y nos devuelve el la clave buscada.

Dicha clave es la siguiente:

27f108

Para encontrar dicha clave, el compilador tarda una media de 80 segundos, por lo que como podemos comprobar lo inseguro que resulta ser esta clave. Es por ello que debemos asegurarnos de que esta tenga un mayor tamaño y por tanto resulte más compleja para el atacante.

Una posible solución para evitar que un intruso intercepte la clave secreta, es generando una aleatoria en el servidor y mediante un canal seguro compartirla con el cliente. Dicha clave debe ser construida respetando las directrices aportadas en el apartado de TAREA 1 y cifrada para así proporcionarle confidencialidad. Además de esto, debemos asegurarnos de que no puedan interceptar la MAC, añadiéndole un nonce aleatorio a esta entre sus caracteres.

Cabe destacar que todos los procedimiento llevados a cabo para asegurar tanto la confidencialidad como la integridad de esta se llevarán a cabo en cliente y en servidor.

CONCLUSIÓN

La integridad en la transmisión debe ser lo más segura posible ya que es en este proceso donde ocurren más ataques, por lo que debemos asegurar tanto la integridad de los mensajes intercambiados como la confidencialidad de los participantes de cualquier comunicación.

Además de esto, cabe mencionar el aprendizaje de las técnicas existentes para mantener dicha integridad, así como su funcionamiento como también su implementación; e incluso también, la importancia y la dificultad de elegir un buen nonce, de modo que gracias a este el riesgo de ataque se vea enormemente reducido.