Documentação do trabalho 2

Protocolos de Segurança

SSL → Secure Sockets Layer

- É utilizado para estabelecer uma conexão segura e criptografada entre um navegador ou o computador do usuário e um servidor ou site
- Hoje em dia é considerado obsoleto e não é mais usado
- Versões
  - 1.0: nunca foi lançada por causa de falhas na segurança
  - 2.0: vulnerável a man in the middle, falta de autenticação e vazamento de dados
  - 3.0: maior resistência a man inthe middle e suporte a autenticação de algoritmos. Vulnerável ao ataque POODLE
- Algoritmos utilizados, considerando apenas a versão mais segura
  - Troca de chaves de forma segura
    - RSA
      - Fatoração de números primos grandes
    - Diffie-Hellman
      - Duas partes geram uma chave secreta compartilhada sem enviá-la
    - Diffie-Hellman Ephemeral
      - Variante de DH que gera chaves a cada conexão, garantindo PFS
  - Criptografia Simétrica
    - RC4
      - Algoritmo de fluxo considerado vulnerável graças ao viés estatístico
    - 3DES
      - Aplicação do DES 3 vezes, lento e vulnerável
  - Integridade
    - MD5
      - Rápido, mas fraco contra ataques de colisão
    - SHA-1
      - Mais seguro que o anterior, mas ainda vulnerável a colisões
  - Assinatura Digital
    - RSA
      - Pode ser lento com chaves muito grandes
    - DSA
      - Alternativa ao RSA, mas menos popular e eficiente

TLS → Transport Layer Security

- É uma versão mais atualizada e segura do SSL
- Garante criptografia, integridade e autenticação nas comunicações
- Versões

    - 1.0: melhorou a segurança do SSL 3.0 e adotou o protocolo de autenticação HMAC, evitando falsificação de mensagens. Vulnerável ao ataque BEAST
    - 1.1: melhorias contra ataques BEAST e suporte para a troca de chaves, no entanto não trouxe melhorias significativas na segurança
    - 1.2: novos algoritmos decriptografia (AES-GCM), flexibilidade na escolhe de funções hash e maio resistência a ataques man in the middle. Ainda tem suporte para algoritmos antigos que podem ser inseguros
    - 1.3: removeu algoritmos inseguros, reduziu a latência do handshale e implementa Perfect Forward Secrecy (PFS), ou seja, mesmpque uma chave seja comprometida, as comunicações passadas ainda são seguras
- Algoritmos utilizados, considerando apenas a versão mais segura
    - Troca de chaves de forma segura
        - ECDH
            - Uso de curvas elípticas para uma troca de chaves mais rápida e segura
        - ECDHE
            - Mesmo que o anterior mas com PFS
    - Criptografia Simétrica
        - AES-GCM
            - Algoritmo simétrico com autenticação embutida
        - ChaCha20-Poly1305
            - Alternatica mais otimizada para processadores de baixa potência
    - Integridade
        - SHA-256
        - SHA-384
            - Ambos SHA-256 e SHA-384 fazem parte da família do SHA-2, seguros e apmplamente utilizados, no entanto o SHA-3 é mais seguro devido a função esponja
    - Assinatura Digital
        - RSA
        - ECDSA
            - Variante mais rápida e segura do DSA
        - EdDSA
            - Moderno e baseado em curvas elípticas, além de mais seguro e rápido que os anteriores

HTTPS → Hypertext Transfer Protocol Secure

- Extensão segura do HTTP
- Os sites que utilizam um certificado SSL/TLS podem utilizar o protocolo HTTPS para estabelecer uma comunicação segura com um servidor
- Utiliza um certificado digital para autenticar o servidor e criptografar a comunicação, impedindo ataques de Man In The Middle (interceptação da mensagem por um terceiro ouvinte)
- Modo de funcionamento
    - Solicitação do cliente para o servidor
    - O servidor envia seu certificado digital validado por uma Autoridade Certificadora, validando sua identidade
    - O cliente verifica a validação do certificado
    - Caso seja válido, o cliente e o servidor estabelecem um handshake TLS, trocando as chaves criptográficas
    - Assim se inicia uma comunicação segura

Fluxo de funcionamento do SSL/TLS

- Autenticação do cliente e do servidor e criação e compartilhaento de uma chave de sessão
- Uso de criptografia simétrica depois que a chave foi estabelecida por sermais rápida
- Os dados que são transmitidos são criptografados (proteção contra espionagem) e protegidos contra alteração por função de hash
- Ao finalizar, cliente e servidor envim mensagens de encerramento seguro e as chaves são descartadas

Implementação

Cliente

```
import requests
import os
import time

PORT = 8080
URL = "https://localhost:" + str(PORT)
CERT_FILE_RECEIVED = "received_cert.pem"


def get_certificado():

    cert = None

    if not os.path.exists(CERT_FILE_RECEIVED):
        open(CERT_FILE_RECEIVED, "w").close()

    with open(CERT_FILE_RECEIVED, "rb") as f:
        cert = f.read()

    if not cert:
        print("Requisitando certificado...")

        url = URL + "/cert"
        response = requests.get(url=url, verify=False)
        if response.status_code == 200:
            with open(CERT_FILE_RECEIVED, "wb") as f:
                f.write(response.content)
            print("Certificado recebido com sucesso!")
            print("Certificaod: "+ response.text)
        else:
            print("Erro ao receber certificado: " + str(response.status_code))
            return None
    else:
        print("Certificado já recebido!")

    return CERT_FILE_RECEIVED


def secure_request():
    cert_file = get_certificado()

    if not cert_file:
        print("Certificado não recebido!")
        return
    print("Requisitando resposta...")
```

```
    response = requests.get(url=URL, verify=cert_file)

    if response.status_code == 200:
        print("Resposta segura recebida com sucesso!\n")
        print(f"Resposta: {response.text}\n")
    else:
        print("Erro ao receber resposta: " + str(response.status_code))



if __name__ == "__main__":
    # Requisição GET #
    for _ in range(10):
        secure_request()
        time.sleep(5)
```

- O código acima faz 10 requisições com um intervalo de 5 segundos entre cada uma
- Caso ocliente já possua o certificado do servidor armazenado, ele não é solicitado e apenas faz realiza uma requisição GET
- Caso contrário, o cliente faz uma requisição no endpoind “/cert”
- Apesar de testada e funcionar a implementação acima apresenta algumas brechas
  - Caso o certificado esteja expirado ou incorreto, o cliente não tenta solicitar o certificado novamente
  - Outro problema é na primeira comunicação com o servidor. O parâmetro “verify=False” desabilita a verificação do certificado na primeira requisição ao servidor, o que deixa a comunicação vulnerável a ataques de terceiros
- A biblioteca requests foi utilizada para fazer as requisições do lado do cliente

Servidor

```
import http.server as server
import ssl
import cryptography.hazmat.primitives.hashes as hashes
import cryptography.hazmat.primitives.asymmetric.rsa as rsa
import cryptography.hazmat.primitives.serialization as serialization
import cryptography.x509 as x509
import datetime as dt
import os

KEY_FILE = "server.key"
CERT_FILE = "server.pem"
PORT = 8080

## Geração de chave e certificado ##

# Geração de chave privada #
def generate_key(public_exponent, key_size):
    return rsa.generate_private_key(public_exponent=public_exponent, key_size=key_size)

# Informações do certificado #
def generate_cert(key):

    cert = 0

    if not os.path.exists(CERT_FILE):
        open(CERT_FILE, "w").close()
```

```
    with open(CERT_FILE, "rb") as f:
        cert = f.read()

    # Se já existe um certificado, retorna ele, se não, gera um novo #
    if cert:
        return cert

    subject = issuer = x509.Name([
        x509.NameAttribute(x509.NameOID.COUNTRY_NAME, "BR"),
        x509.NameAttribute(x509.NameOID.STATE_OR_PROVINCE_NAME, "Brasília"),
        x509.NameAttribute(x509.NameOID.LOCALITY_NAME, "Cidade"),
        x509.NameAttribute(x509.NameOID.ORGANIZATION_NAME, "UnB"),
        x509.NameAttribute(x509.NameOID.COMMON_NAME, "localhost"),
    ])

    # Geração do certificado #
    certificado = (
        x509.CertificateBuilder()
        .subject_name(subject)
        .issuer_name(issuer)
        .public_key(key.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(dt.datetime.now())
        .not_valid_after(dt.datetime.now() + dt.timedelta(days=365))
        .add_extension(x509.BasicConstraints(ca=True, path_length=None), critical=True)
        .add_extension(x509.SubjectAlternativeName([x509.DNSName("localhost")]), critical=False)
        .sign(key, hashes.SHA256())
    )

    with open(KEY_FILE, "wb") as f:
        f.write(key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption()
        ))

    with open(CERT_FILE, "wb") as f:
        f.write(certificado.public_bytes(serialization.Encoding.PEM))

class MySimpleHTTPRequestHandler(server.SimpleHTTPRequestHandler):
    def do_GET(self):
        if self.path ==  "/cert":
            with open(CERT_FILE, "rb") as f:
                self.send_response(200)
                self.send_header("Content-type", "application/x-x509-ca-cert")
                self.end_headers()
                self.wfile.write(f.read())
        else:
            self.send_response(200)
            self.send_header("Content-type", "text/plain")
            self.end_headers()
            self.wfile.write(b"Mensagem segura!")

if __name__ == "__main__":
    key = generate_key(public_exponent=65537, key_size=4096)

    generate_cert(key)
```

```
server_address = ('localhost', PORT)
httpd = server.HTTPServer(server_address, MySimpleHTTPRequestHandler)

# Configura o SSL para o servidor
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.set_ecdh_curve("secp384r1")
context.load_cert_chain(certfile=CERT_FILE, keyfile=KEY_FILE)

httpd.socket = context.wrap_socket(httpd.socket, server_side=True)

print(f"Servidor HTTPS rodando em https://localhost:{PORT}")
httpd.serve_forever()
```

- Para a geração da chave, foi utilizada a biblioteca criptography
  - A partir da função de geração de chaves do módulo do rsa é gerada uma chave privada armazenada
  - A chave é gerada com o expoente público de valor 65537 e de tamanho de 4096 bits
  - OBS: inicialmente o tamanho da chave usada seria de 1024 bits, no entanto a biblioteca usada possui verificações de segurança que não consideram esse tamanho como seguro, gerando um erro de tamanho de chave
- Para a geração do certificado foi utilizada a biblioteca x509
  - Primeiro são informados alguns atributos do certificado
  - Depois através do CertificateBuilder são informadas informações sobre sua validade
  - Após tudo isso o certificado é assinado com a chave gerada e a partir de um algoritmo escolhido, no caso o sha256
  - Assim ele é salvo, assim como a chave, num arquivo
- Na função main, o servidor cria um socket onde irá escutar por conexões https
  - É criado um contexto que utiliza o protocolo TLS
  - Além disso, ele também utiliza o protocolo ECDH (Curva Elíptica de Diffie-Hellman), que permite que duas partes, cada uma com um par de chaves pública-privada de curva elíptica, estabeleçam um segredo compartilhado em um canal seguro
  - Dessa forma o socket é criado e p servidor começa a ouvir na porta 8080

Fontes

- Biblioteca Python: https://docs.python.org/pt-br/3.8/library/ssl.html#module-ssl
- Como configurar um servidor HTTPS: https://awari.com.br/python-como-configurar-um-servidor-https/
- Definições: https://www.digicert.com/pt/what-is-ssl-tls-and-https
- Definições: https://www.hostinger.com.br/tutoriais/o-que-e-ssl
- X509: https://cryptography.io/en/latest/x509/reference/#x-509-certificate-builder