Segurança Aplicada

2022/2023

CTI - Class Project

Rafael Pilré - Nº 59448
Tiago Silva - Nº 59446
Diogo Fernandes fc54458

Sistema eCommerce Communication Security

O sistema desenvolvido tem como objetivo permitir pagamentos através do serviço de transação com segurança e privacidade. O sistema é composto por três partes principais: o banco, o MBEC (Mobile Bank Entity Client) e o estabelecimento comercial.
O protocolo de comunicação entre as entidades começa com uma operação de estabelecimento que inclui um desafio (challenge). O MBEC criptografa o desafio com a chave pública do banco e envia de volta ao banco. O banco, então, decifra o desafio e envia um novo desafio ao MBEC para verificação, este processo acontece sempre que uma entidade quer estabelecer comunicação entre ambas.

```
self.challenge = str(random.randint(0,pow(10,50)))

fst_req = self.decrypt_with_prikey(data).decode()
js = json.loads(fst_req)

establishpost = {
    'op' : 'establishpost',
    'result' : str(js['challenge']),
    'challenge' : self.challenge
}

client_socket.sendall(self.encrypt_with_pubkey(establishpost)) #response from bank
```

Se a autenticação corre como esperado, isto é, a resposta ao desafio prova-se correta, então o banco partilha com o seu cliente (seja ele o mbec ou a store) a chave simétrica a ser usada dali em diante bem como uma chave hmac.

```
fst_req = self.decrypt_with_prikey(data).decode()
js = json.loads(fst_req)
op = js['op']

if op == "sharingkeyreq":

    client_socket.sendall(self.encrypt_with_pubkey(self.symmetric_key)) #response from bank
```

```
def share_hmac_key(self,client_socket):

    self.secret_key = secrets.token_bytes(32)

    client_socket.sendall(self.encrypt_with_pubkey(self.secret_key)) #response from bank
```

Todas as operações subsequentes são realizadas com essa chave simétrica e com hmacs construídos perante a chave hmac acordada. Este procedimento acontece entre todas as entidades para se autenticarem.

O formato das mensagens é definido como JSON e é constituído por uma operação (op) e um conjunto de argumentos específicos para cada operação.

```
def handle_transaction(self, client_socket, req):
```

O seguinte é responsável por proceder à encriptação e decriptação das mensagens e torná-las legíveis para operações futuras. Neste mesmo método, é feita a verificação ou composição da integridade das mensagens, com auxílio a hmacs. De maneira breve, o sock consiste no socket desejado, o mode tem duas opções - “r” ou “s” - que consiste em determinar se é suposto tratar de data recebida ou de data a enviar, k é a chave simétrica anteriormente determinada e sec_k é a chave hmac a ser utilizada.

```
def send_recv_data(self,sock,mode,data,k,sec_k):
```

Este método utiliza encriptação simétrica AES para proteger os dados e HMAC para garantir a integridade dos dados.

Quando os dados são enviados, eles são primeiro criptografados utilizando uma chave AES simétrica e um IV aleatório. A encriptação protege os dados para que só possam ser lidos pelo destinatário pretendido, enquanto o IV ajuda a garantir que o mesmo plaintext nunca seja encriptado da mesma forma duas vezes. Em seguida, uma assinatura HMAC é gerada a partir dos dados encriptados e enviada junto com os dados.

O destinatário pode verificar a assinatura HMAC para garantir que os dados não foram alterados durante a transmissão. Quando estes são recebidos, são primeiro verificados usando a assinatura HMAC para garantir que não foram alterados, se a verificação HMAC falhar, o destinatário sabe que os dados foram modificados durante a transmissão e pode descartá-los. Se for bem-sucedida, os dados são desencriptados usando a chave AES simétrica e o IV. Isto permite que o destinatário leia os dados.

Autenticação e outros detalhes

Algumas operações requerem autenticação e são protegidas usando criptografia assimétrica (RSA) para garantir a segurança e a privacidade. O ficheiro de autenticação (auth file) contém a chave pública do banco e uma chave de hmac inicial. A primeira é usada para criptografar o desafio inicial. Já a segunda serve para garantir que a partilha de chave públicas entre as partes não sofra ataques de integridade. O ficheiro do utilizador (user file) contém o pin do utilizador. O ficheiro VCC (Virtual Credit Card) contém as informações do cartão de crédito virtual, que é usado para realizar transações com segurança.

O sistema implementa medidas de proteção para garantir a segurança e privacidade dos utilizadores. Essas medidas incluem o uso de criptografia assimétrica para autenticação, o uso de criptografia simétrica para proteger a comunicação e a implementação de medidas de segurança adicionais para proteger a chave privada do banco e hmac para proteger a integridade das mensagens. O sistema foi projetado para garantir que as informações dos utilizadores sejam mantidas em sigilo e que as transações sejam realizadas com segurança e privacidade. O código apresentado contém várias validações de input que são realizadas utilizando expressões regulares para garantir que os argumentos passados para o programa sejam válidos e seguros.

Ataques que o sistema mitiga

Eavesdropping attack - Este tipo de ataque consiste na monitorização e intercepção de mensagens entre duas partes sem o consentimento ou conhecimento das mesmas

De modo a evitá-lo, usamos criptografia assimétrica de maneira a que as mensagens fossem completamente escondidas de eventuais sistemas que as tentam interceptar. Mesmo no momento onde se passa para uma criptografia simétrica de mensagens, tanto a chave simétrica como a chave hmac foram partilhadas com auxílio a criptografia assimétrica

```python
def decrypt_with_prikey(self,data):

    res = self.private_key.decrypt(
    data,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    ))

    return res

def encrypt_with_pubkey(self,data):

    f_data = data if isinstance(data,bytes) else (json.dumps(data, ensure_ascii=False)).encode()

    req = self.cli_pubkey.encrypt(
    f_data,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    ))

    return req
```

Man-in-The-Middle attack - Este tipo de ataque envolve um atacante interceptando e modificando mensagens entre duas partes para obter acesso não autorizado ao sistema. Para evitar esse tipo de ataque, a nossa implementação usa criptografia assimétrica para trocar chaves simétricas de forma segura entre o banco e o cliente. Isso é implementado no método "`share_cipher_param`" na classe Bank, onde a chave pública do banco é usada para criptografar a chave simétrica antes de enviá-la para o banco. Para além disso, foram implementadas medidas de prevenção contra ataques de integridade como, por exemplo, a utilização de Hmacs.

```python
sock.shutdown(socket.SHUT_RDWR) if not hmac.compare_digest(s_hmac_digest,hmac_digest) else None
```

Cross-channel attack- Este ataque consiste em utilizar informações obtidas num canal para conduzir um ataque noutro. Neste caso, um mbec podia tentar obter um challenge do banco de forma legítima e o atacante poderia tentar utilizar esse desafio para se autenticar noutro canal.

De modo a evitar este tipo de ataques, o nosso sistema gera um novo challenge para cada novo cliente, sendo, portanto, impossível reutilizar desafios entre iterações

```python
def handle_auth(self, client_socket, data):

    self.challenge = str(random.randint(0,pow(10,50)))
```

Know-Plaintext-Attacks - Este tipo de ataque ocorre quando um atacante tem em posse tanto a mensagem original como a mensagem encriptada correspondente. Por conta de ser conhecido o formato das mensagens (esclarecido anteriormente), existe a possibilidade deste ataque ser conduzido.

No entanto, e de forma a evitá-lo, o sistema conta com várias contra-medidas como o facto da partilha da chave simétrica ocorrer de uma forma segura (para isso é utilizada criptografia assimétrica), o IV ser aleatório e sempre diferente a cada iteração e o facto de cada mensagem que é partilhada ter associado a ela um fator aleatório (permite que seja mais difícil prever a mensagem)

```
fst_req = self.decrypt_with_prikey(data).decode()
js = json.loads(fst_req)
op = js['op']

if op == "sharingkeyreq":

    client_socket.sendall(self.encrypt_with_pubkey(self.symmetric_key)) #response from bank
```

```
iv = os.urandom(16)
```

```
res = {
    "op" : None,
    "result" : None,
    "sequence" : None,
    "balance" : None,
    "pin" : None,
    "rand" : str(random.randint(0,pow(10,50))) #introduce randomness in message
}
```

O que gostaríamos de ter implementado

Ao nível da implementação, o grupo gostaria de ter implementado uma dinâmica de obtenção e validação das chaves públicas diferente da atual.

Por conta da limitação de ficheiros a serem criados, o grupo não conseguiu implementar uma validação externa das chaves públicas do mbec nem da store, diferente do que aconteceu no bank. Por isso, consideramos isso um ponto crítico na implementação, pois permite que um agente malicioso, personificando o mbec ou mesmo a store, tenha acesso a informações críticas guardadas no sistema bancário.