从密码学理论到密码学的工程实践与应用

一、项目选题

(一) 项目实现功能

1. 基于网页的用户注册与登录系统
    1) 使用 https 绑定证书到 ip 地址【PKI X.509 证书，openssl】
    2) 允许用户注册到系统
        a. 对用户名的合法性进行检测
        b. 对用户口令的长度和强度进行校验，禁止使用弱口令
        c. 用户的口令加密存储，即使被公开也无法还原出原始口令【PBKDF2 散列算法 慢速散列 针对散列算法（如 MD5、SHA1 等）的攻击方法 】
    3) 允许已注册的用户登录到系统
2. 基于网页的文件上传与数字签名系统
    1) 对上传文件的大小和类型进行相关限制
        a. 文件大小：<10MB
        b. 文件类型：office 文档、常见图片
    2) 匿名用户禁止上传文件
    3) 对文件进行对称加密存储到文件系统【SHA256 算法】
        a. 使用对称加密【AES-256 CBC 模式】
        b. 实现安全存储对称密钥【数字签名及验证】
    4) 系统对加密文件进行数字签名【数字签名及验证】
3. 基于网页的加密文件的下载与解密系统
    1) 已登录用户可以下载该用户上传的文件
        a. 用户下载解密后的文件【AES-256 CBC 模式】
    2) 匿名用户根据已登录用户分享的下载 url 进行文件下载，该 url 设置有下载次数，超过次数则不能下载
        a. 匿名用户下载的是打包文件，里面有包括加密过的文件，文件的数字签名，验证签名的公钥，解密文件的对成密钥
        b. 匿名用户可在客户端对加密文件进行解密，生成原始文件【AES-256 CBC 模式】
        c. 匿名用户可在客户端对数字签名进行验证【数字签名及验证】
    3) 匿名用户和登录用户可根据需要对文件散列值进行下载，供用户在本地验证文件完整性。【SHA256 算法】

(二) 个人承担工作（总工作量超过 65%）

1. 允许用户注册到系统 （ 前端 html 页面 js 代码实现）
2. 用户名的合法字符集范围：中文、英文字母、数字 类似：-、_、.等合法字符集范围之外的字符不允许使用
3. 用户口令长度限制在 36 个字符之内

4. 对用户输入的口令进行强度校验，禁止使用弱口令
5. 使用 https 绑定证书到域名而非 IP 地址
6. 限制文件大小：＜10MB
7. 限制文件类型：office 文档、常见图片类型
8. 匿名用户禁止上传文件
9. 系统对加密后文件进行数字签名
10. 客户端对下载后的文件进行数字签名验证
11. 虚拟机 web 服务器搭建
12. 用户注册、登录的接口和后端数据库操作
13. 前端 html 页面：Welcom.html,Login.html,Register.html,Upload.html
14. 项目代码整合调试（包括 web 服务器下载，上传功能实现等）
15. 项目报告编写和视频录制

二、开发环境

使用 flask 搭建 web 服务器，MySQL 作为数据库支撑。

三、关键技术

（一）口令安全存储

使用两个函数对用户口令进行处理

1. 口令加密函数：传入用户口令，使用 PBKDF2 慢速散列算法计算口令哈希值，并作为返回值返回。

```
def derivePassphrase(passphrase):
    algo = 'sha256'

    # 从人类可记忆「口令」生成面向加密算法用途的「密钥」
    password = pbkdf1(algo, passphrase.decode('latin1'),
                      salt, rounds, keylen=outlen)
    #print(binascii.hexlify(password))
    # 扩展秘钥
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=outlen,
        salt=salt,
        iterations=rounds,
        backend=backend
    )
    password = kdf.derive(passphrase)
    # 测试输出，输出加密存储的密钥
    return password
```

2. 口令验证函数：用户发起登录请求后，将输入的口令和注册时存储的口令哈希值传入验证函数，对用户输入的口令同样做 PBKDF2 散列后，将得到的值与已存储的哈希值比对，比对成功则允许用户登入，否则提示登录不成功。

```
def verifyPassphrase(passphrase, password):
    # 验证密钥
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=outlen,
        salt=salt,
        iterations=rounds,
        backend=backend
    )
    try:
        kdf.verify(passphrase, password)
        # 测试输出，相同则认证成功
        print('Key Verification OK!')
    except Exception as e:
        # 否则输出错误信息
        print(e)
```

(二) 文件加密存储

算法：AES-256；模式：CBC

首先创建一个类 class prpcrypt()，包括初始化函数，加密函数和解密函数。从 Crypto.Cipher 库导入 AES 算法。类中，加密函数使用 AES.new(self.key, self.mode, self.iv)生成对称密钥，此处密钥长度设置为 32，对传入的文件内容使用 len 方法得到长度，使用除法验证文本长度是否为 32 的倍数，不足则用空格'\0'添加在其后补足，使用 cryptor 库中 encrypt 方法加密补足后文本，函数返回密文。

```
class prpcrypt():
    def __init__(self, key, iv):
        self.key = key
        self.iv = iv
        self.mode = AES.MODE_CBC
        self.pad = 0

    # 加密函数，如果text不是32的倍数，那就补足为32的倍数
    def encrypt(self, text):
        cryptor = AES.new(self.key, self.mode, b'0000000000000000')
        # 这里密钥key 长度必须为16（AES-128）
        # 或24（AES-192）或32（AES-256）Bytes 长度
        # 这里使用AES-256确保安全
        length = 32
        count = len(text)
        extra = count % length
        if extra != 0:
            self.pad = length - (count % length)
            text = text + ('\0' * self.pad)
        ciphertext = cryptor.encrypt(text)
        return ciphertext

    # 解密后，去掉补足的空格用strip() 去掉
    def decrypt(self, text):
        cryptor = AES.new(self.key, self.mode, b'0000000000000000')
        plain_text = cryptor.decrypt(text)
        # plain_text = plain_text[:-self.pad]
        plain_text = plain_text.decode('utf-8').rstrip('\0')
        return plain_text
```

在类外使用文件加密函数 def decrypt_file(name, cipher)，传入的参数为文件名，以及用 os.urandom(32)生成的 32 位 key 和 iv 作为初始值传入自定义 prpcrypt 类生成的类的对象。函数中用二进制方式读入传入文件的内容，调用类中加密函数进行加密，加密完成后写回原始文件，即得到加密后文件。

```
# 文件加密函数，加密后得到名为“encrypt_原文件名”的加密文件
def encrypt_file(name, cipher):
    fs = open(name,'rb')
    text = fs.read()
    cipher_text = cipher.encrypt(binascii.hexlify(text).decode('utf-8'))
    print('8',type(cipher_text))
    fs.close()
    fc = open('encrypt_'+name,'wb')
    fc.write(cipher_text)
    fc.close()
```

该实现关键技术不足反思:

1.密码学库选用不当，对称加密的密码学库已被证明是存在安全漏洞的 PyCryto 库，使用 Crytography 或者 PyNacl 库更好;

2.对称加密或作模式选择不当，应使用符合 AEAD 标准的 GCD 模式。

3.密文 padding 方案存在密文结构设计错误，极不严谨

(三) 文件解密

prpcrypt 类中解密函数调用 cryptor 库的解密方法 decrypt，再用 strip() 去掉加密时在后面补足的空格，即得到原始文件内容。

在类外使用文件解密函数 def decrypt_file(name, cipher)，传入的参数为加密后文件的文件名以及 prpcrypt 类的一个对象，函数中用二进制方式读入传入文件的内容，调用类中解密函数进行解密，解密完成后写回该文件，即得到解密后的原始文件。

```
# 文件解密函数，解密后得到名为“decrypt_encrypt_原文件名”的解密后文件
def decrypt_file(name, cipher):
    fs = open(name,'rb')
    text = fs.read()
    print('1',text)
    cipher_text = cipher.decrypt(text)
    print('2',cipher_text)
    fs.close()
    fc = open('decrypt_'+name,'wb')
    fc.write(binascii.unhexlify(cipher_text))
    print(binascii.unhexlify(cipher_text))
    fc.close()
```

函数调用方式如下。首先生成 32 位随机数保存在文本文件中，为后面对称密钥的保存做准备，再将其作为初始化参数传入类中，然后可直接调用文件加解密函数。

```
if __name__ == '__main__':
    #生成32位随机数
    key = os.urandom(32)
    fr = open('key.txt', 'w')
    fr.write(binascii.hexlify(key).decode('utf-8'))
    fr.close()

    fs = open('key.txt', 'rb')
    text = fs.read()
    print(binascii.unhexlify(text))
    fs.close()
    # 初始化密钥
    cipher = prpcrypt(key,key)
    # 设置文件名
    filename = 't1.pdf'
    # 文件加密函数，cipher为对称密钥，需要加密后保存在数据库
    encrypt_file(filename,cipher)
    #文件解密函数，cipher为对称密钥，解密后得到原文件
    decrypt_file('encrypt_'+filename,cipher)
```

（四）静态文件的散列值计算

选用 SHA256 算法保证安全性。

```
import hashlib

# SHA256计算散列值的函数
# filename: 文件路径
def sha256(filename):
    f = open(filename, 'rb')
    sh = hashlib.sha256()
    sh.update(f.read())
    # 将散列值输出到“文件名_sha256.txt”中
    fr = open(filename+'_sha256.txt', 'w')
    fr.write(sh.hexdigest())
    f.close()
    return

sha256('test.pdf') #调用时输入文件路径
```

（五）公私钥对生成

用户需要通过用户密码导出自己的私钥，在导出用户私钥的过程中通过将用户密码转化为组合成组成密码的字符处字符的 ASCll 码值和作为生成 exponent 值。

```
#以用户密码为参数生成public_exponent
def generate_exponent(password):
    base=65537

    increase=0
    for i in password:
        increase=increase+ord(i)
    print(increase)
    exponent=base+increase
    while juge(exponent)!=True:
        exponent = exponent + 1
    return exponent
```

私钥生成

```
key = rsa.generate_private_key(
    public_exponent=exponent,
    key_size=2048,
    backend=default_backend(),
)
```

公钥生成

```
#生成用户公钥
pub = key.public_key()
```

导出 pem 存储用于各种功能的实现

```
key_pem = key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.TraditionalOpenSSL,
    encryption_algorithm=serialization.NoEncryption(),
)

pub_pem=pub.public_bytes(
    encoding=serialization.Encoding.PEM,
    format = serialization.PublicFormat.SubjectPublicKeyInfo,
    #format=serialization.PublicFormat,
)
```

(六) 文件签名与验证

文件签名用已经导出的 key.pem,通过 seralization.load_pem_pprivate_key 把 pem 文件转化为签名所需要的私钥。

```
# 从PEM文件数据中加载私钥
private_key = serialization.load_pem_private_key(
    private_key,
    password=None,
    backend=default_backend()
)
```

用于实现签名的填充方式为 PKCS1v15、hash 方式为 sha256

```
signature = private_key.sign(
    data,
    padding.PKCS1v15(),
    hashes.SHA256()
)
```

(七) 对称秘钥加密与解密

对称秘钥加密文件，需要用户的公钥加密对称秘钥，保护对称秘钥的安全性，公钥加密数据，使用 PKCS1 v1.5 的填充方式

```
out_data = public_key.encrypt(
    data,
    padding.PKCS1v15()
)
```

私钥解密数据同样使用 PKCS1 v1.5 的填充方式

```
out_data = private_key.decrypt(
    data,
    padding.PKCS1v15()
)
```

（八）openssl 生成局域网内 CA 根证书，服务器证书

1. CA 证书

   创建私钥：openssl genrsa -out ca-key.pem 1024

   创建 ca 根证书请求：Openssl req -new -out ca-req.csr -key ca-key.pem

   产生自签署证书：openssl x509 -req -in ca-req.csr -out ca-cert.pem -signkey ca-key.pem -days 3650

2. SERVER 证书

   创建私钥：openssl genrsa -out ca-key.pem 1024

   创建 server 证书请求：openssl req -new -out ca-req.csr -key ca-key.pem

   自签署证书：openssl x509 -req -in ca-req.csr -out ca-cert.pem -signkey ca-key.pem -days 3650

（九）生成证书应用

将 ca 根证书导入浏览器，虚拟机 flask 运行服务器注明 ssl_context