



密码学综合实验第四次实验

实验名称：口令泄露查询密码协议系统构建

姓 名：	柳致远
学 号：	2113683
专 业：	密码科学技术
指导老师：	邓琮弋

目录

密码学综合实验第四次实验.....	1
实验名称：口令泄露查询密码协议系统构建.....	1
一、 实验目的.....	2
二、 实验内容说明.....	3
三、 实验原理.....	3
1、 不经意伪随机函数.....	3
2、 系统架构.....	3
3、 数据库预处理.....	4
四、 实验步骤.....	5

1、 服务器数据库预处理	5
2、 功能实现.....	7
五、 实验结果分析.....	11
六、 总结感想.....	13
七、 拓展部分.....	13

一、实验目的

通过实际编程（GPC, IDB 任选其一即可）了解口令泄露查询协议的交互过程，掌握口令泄露查询协议的基本设计和分析方法。

编写简单的客户端-服务器，具体要求如下：

1) 实现最基本的口令泄露查询，用户输入用户名-口令对，可以了解到她的输入凭证泄露、未泄露。

2) 保证系统的安全性，能防范恶意客户端的蛮力搜索，又能预防诚实且好奇服务器试图获取用户输入的口令信息。

3) 保护协议主体的隐私，在交互过程中用户不会泄露她所查询的用户名和口令的明文，服务器也不会向用户泄露额外的数据。

4) 支撑多用户；

5) 服务器显示日志，记录与客户端交互过程；

6) 编程语言不限，可以使用 Python/C++/Go/Java 等；

数据报方式不限，可基于 TCP 或 UDP；

程序界面不限，可使用命令行界面或图形化界面。

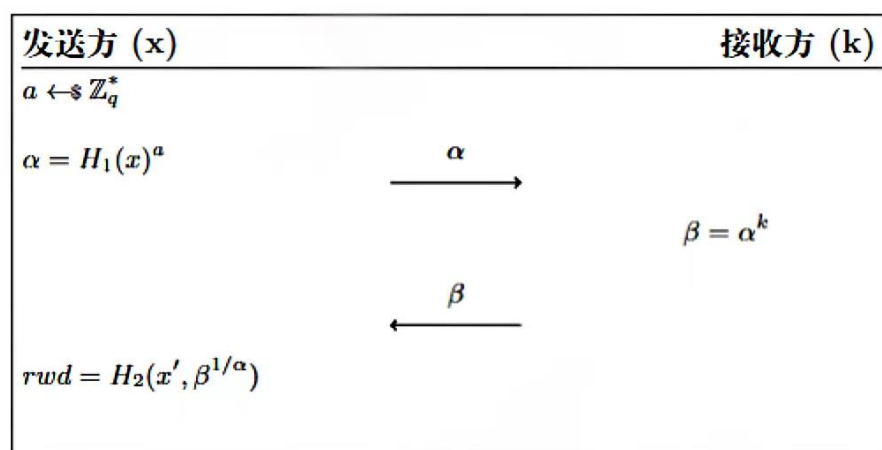
实验环境不限，可基于 Linux/Windows 等

二、实验内容说明

首先服务器预处理文件设置一个以 c3server_v3 命名的泄露数据库。它存储了加密后的用户名-口令对。加密凭据值依据用户名的哈希值前缀进行划分，具有同一哈希值用户名前缀的加密凭据划分在同一数据桶中。启动服务器后在客户端运行用于本地查询的 python 文件，输入想要查找的用户名-口令对，检查是否泄露。若未泄露显示 “none”,若已泄露则显示 “match”。

三、实验原理

1、不经意伪随机函数



2、系统架构

系统由客户端和服务端组成。服务器维护一个加密数据库，加盐存储与已泄露的用户名及口令相关的记录，根据用户名的哈希前缀划分出若干个加密数据桶；客户端通过提供用户名及口令来查询口令是否泄露。

当客户端输入要查询的用户名及口令，服务器首先根据用户名的哈希前缀判断是否存在对应的数据桶，如不存在则表明未泄露，查询结束；否则，将相应的数据桶返回给客户端。客户端根据服务器的响应来判断此次查询的口令是否泄露。

Precomputation by C3 Server

Let $\tilde{S} = \{(u_1, w_1), \dots, (u_N, w_N)\}$

$\forall j \in [0, \dots, 2^l - 1]$

$z_j \leftarrow \{F_\kappa(u_i \| w_i) \mid H^{(l)}(u \| w) = j\}$

$z_j \leftarrow \{F_\kappa(u_i \| w_i) \mid H^{(l)}(u) = j\}$

Client

Input: (u, w)

$r \leftarrow \$\mathbb{Z}_q$

$x \leftarrow F_r(u \| w)$

$b \leftarrow H^{(l)}(u \| w)$

$b \leftarrow H^{(l)}(u)$

$\tilde{x} \leftarrow y^{\frac{1}{r}}$

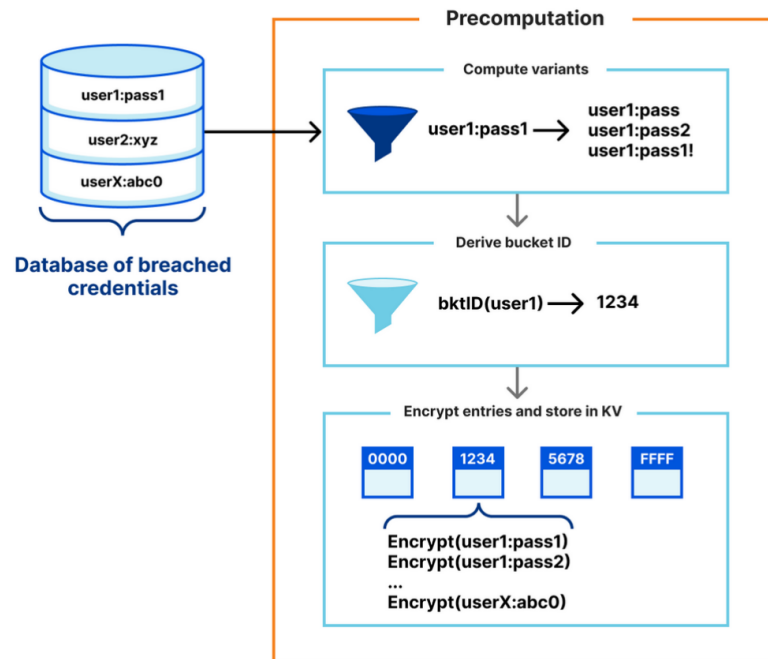
Return $\tilde{x} \in z_b$

C3 server

Input: κ, z

$\xrightarrow{x, b}$
 $\xleftarrow{y, z_b}$ $y = x^\kappa$

3、数据库预处理



四、实验步骤

1、服务器数据库预处理

```
def preduce_data(count,k):

    #数据库预处理

    start_time = time.time()

    # 创建一个 MySQL 引擎
    engine = create_engine('mysql://root:root@localhost/c3server_v3')

    # 定义一个 users 表
    pre_data = Table('pre_data', metadata,
        Column('id', Integer, primary_key=True),
        Column('username', String(255)),
        Column('password', String(255)),
        Column('username_hash', String(255)),
        Column('password_hash', String(255)))

    hash_data = Table('hash_data', metadata,
        Column('id', Integer, primary_key=True),
        Column('username_hash', String(255)))

    # 创建表
    metadata.create_all(engine)

    prefix_table_mapping = {}

    # 准备生成的用户名数量
    num_users = 1000
```

```

# 生成随机用户名和密码并插入到数据库中

with engine.connect() as conn:
    for i in range(num_users):

        username = ''.join(random.choices(string.ascii_lowercase, k=8))

        password = ''.join(random.choices(string.ascii_letters + string.digits, k=12))

        username_hash = hashlib.sha256(username.encode()).hexdigest()
        # 计算密码的哈希值
        password_hash = hashlib.sha256(password.encode()).hexdigest()
        # 计算用户名||密码的 hash 值的 k 次方
        hash_and_powered = hash_and_power(username, password, k)
        # 检查哈希前缀是否已经存在于映射中， 如果不存在则创建新表
        if username_hash[:2] not in prefix_table_mapping:
            # 动态创建新表
            table_name = f"{username_hash[:2].lower()}"
            prefix_table_mapping[username_hash[:2]] = Table(
                table_name, metadata,
                Column('id', Integer, primary_key=True),
                Column('username_hash', String(255))
            )
            metadata.create_all(engine) # 创建新表

        # 插入数据到 pre_data 表
        conn.execute(pre_data.insert().values(
            username=username, password=password,
            username_hash=username_hash, password_hash=password_hash))

        conn.execute(prefix_table_mapping[username_hash[:2]].insert().values(
            username_hash=username_hash))

    end_time = time.time()

    execution_time = end_time - start_time

    print(f"数据库预处理 time: {execution_time} seconds")
    print('服务器已启动， 数据库预处理完成')

```

在这里我通过 python 语言接连 mysql 数据库， 直接在 Navicat 上生成 1000

个随机用户名，密码对元组，再分别进行 `hash` 运算并存储。因为我这里只生成了 1000 个数据，数据个数比较少，所以就以 `hash` 值的前 2 位作为前缀进行分装，同时输出数据预处理的运行时间。在数据库初始化的时候我直接将表名命名为 `hash` 值的前 2 位，更加方便查找。

2、功能实现

```
def hash_and_power(u, w, r):  
    # 连接字符串 u 和 w  
    concat_str = u + w  
  
    # 对连接后的字符串进行 SHA-256 哈希运算  
    hashed = hashlib.sha256(concat_str.encode()).hexdigest()  
  
    # 将哈希结果转换为整数  
    hashed_int = int(hashed, 16)  
  
    # 对整数进行 k 次方运算  
    result = pow(hashed_int, r)  
  
    return result
```

我在这里定义了一个函数，用于对数据库中的用户名和密码做连 `sha-256` 的 `hash` 运算后的结果再取 `r` 次方，这里的 `r` 是客户端随机生成的盲因子用于对输入到服务端的用户名和密码起到加密的操作。

```
hashed_value = hashlib.sha256(u.encode()).hexdigest()  
b = hashed_value[:2]  
client_socket.sendto(f'{b}'.encode(), server_address)
```

这里对输入的用户名取 `sha-256` 的 `hash` 运算之后，取前两位赋值给 `b`，用于服务端查找对应的表。

```

x = hash_and_power(u, w, r)
# 将 x 转换为字节串
x_bytes = x.to_bytes((x.bit_length() + 7) // 8, 'big')
# 将 x 字节串拆分成多个段，并逐个发送
segment_size = 4096
num_segments = int((len(x_bytes) + segment_size - 1) // segment_size)
client_socket.sendto(f'{num_segments}'.encode(), server_address)
for i in range(num_segments):
    start = int(i * segment_size)
    end = int(min((i + 1) * segment_size, len(x_bytes)))
    segment_data = x_bytes[start:end]
    # 在数据前面添加一个4字节的序号，用来标识数据包的顺序
    packet = struct.pack('I', i) + segment_data
    client_socket.sendto(packet, server_address)

```

这里由于 x 的值过于大了，使用不同的数据类型均无法传给服务端，故我在这里采用了分段传输的方法。将 x 划分为多个大小为 4096 的数据包，编号后分段传输给服务端。

```

# 接收所有分段数据
received_data = b""
received_packets = {}
data, client_address = server_socket.recvfrom(4096)
num_segments = data.decode() # 将接收到的字节串解码为字符串
num_segments = int(num_segments) # 将字符串转换为整数类型
while len(received_packets) < num_segments:
    # 接收数据包
    packet, client = server_socket.recvfrom(4096)
    # 解析序号和数据
    seq_number, data = struct.unpack('I', packet[:4])[0], packet[4:]
    # 将数据包存储到字典中
    received_packets[seq_number] = data
# 将接收到的数据按顺序组装
for i in range(num_segments):
    received_data += received_packets[i]
# 解析接收到的数据
x = struct.unpack('Q', received_data[:8])[0]
print("成功接收 x")

```

同时，服务端这里也是采用分段接收的方法，将从客户端发送过来的多个

数据包按照序号重新合并为 x 。再取 k 次方运算得到 y ，这里的 k 为服务端的密钥，用于加密数据库中的数据。

```
b, client_address = server_socket.recvfrom(4096)
print("成功接收 b: " + b.decode())

# 将 b_bytes 解码为十六进制字符串 b
b = b.hex()
# 指定要查询的表名
byte_string = bytes.fromhex(b)

# 将字节对象解码为字符串
table_name = byte_string.decode('utf-8')
print("table_name:" + table_name)
#table_name = str(b, 'utf-8')

# 从数据库中获取指定表的元数据

table = Table(table_name, metadata, autoload_with=engine)

# 连接数据库并执行查询
with engine.connect() as conn:
    # 执行 SELECT 查询
    query = table.select().with_only_columns(table.c.username_hash)

    result = conn.execute(query)

    # 获取查询结果
    rows = result.fetchall()

    # 使用 pickle 序列化查询结果
    serialized_rows = pickle.dumps(rows)
    # 发送序列化后的查询结果给客户端
    server_socket.sendto(serialized_rows, client_address)
```

这里服务端将从客户端传来的 b （用户名和密码连接的 sha-256 运算后的前 2 位字符）通过服务端的数据库查找得到相应的元数据列打包后发给客户端，等待客户端对 y 解码后自己进行对照查找并得到对应的匹配结果。

```

y = pow(x,k)
# 将 x 转换为字节串
y_bytes = y.to_bytes((y.bit_length() + 7) // 8, 'big')
# 将 x 字节串拆分成多个段，并逐个发送
segment_size = 4096
num_segments = int((len(y_bytes) + segment_size - 1) // segment_size)
server_socket.sendto(f'{num_segments}'.encode(), client_address)
for i in range(num_segments):
    start = int(i * segment_size)
    end = int(min((i + 1) * segment_size, len(y_bytes)))
    segment_data = y_bytes[start:end]
    # 在数据前面添加一个4字节的序号，用来标识数据包的顺序
    packet = struct.pack('I', i) + segment_data
    server_socket.sendto(packet, client_address)

```

服务端对从客户端传来的 x 取 k 次方运算之后得到的 y 同样无法直接传给客户端，我在这里直接套用客户端向服务端传递 x 的步骤分段转发的方式，传给服务端。

```

received_data = b""
received_packets = {}
data, server_address = client_socket.recvfrom(4096)
num_segments = data.decode() # 将接收到的字节串解码为字符串
num_segments = int(num_segments) # 将字符串转换为整数类型
while len(received_packets) < num_segments:
    # 接收数据包
    packet, client = client_socket.recvfrom(4096)
    # 解析序号和数据
    seq_number, data = struct.unpack('I', packet[:4])[0], packet[4:]
    # 将数据包存储到字典中
    received_packets[seq_number] = data
# 将接收到的数据按顺序组装
for i in range(num_segments):
    received_data += received_packets[i]
# 解析接收到的数据
y = struct.unpack('Q', received_data[:8])[0]
print("成功接收 y")

```

同样这里客户端对 y 的接收工作我也直接套用了服务端对 x 的分段接收方

式，这里不再赘述。

```
y_float = float(y)
x_ = pow(y_float, 1/r)
x_ = pow(x_, 1/7)
x_str = str(x_)
print("x_:" + x_str)

# 在查询结果中查找特定数据的示例
target_data = 'x_'
for row in rows:
    if target_data in row:
        print("match")
        break
else:
    print("none")

# 关闭连接
client_socket.close()
```

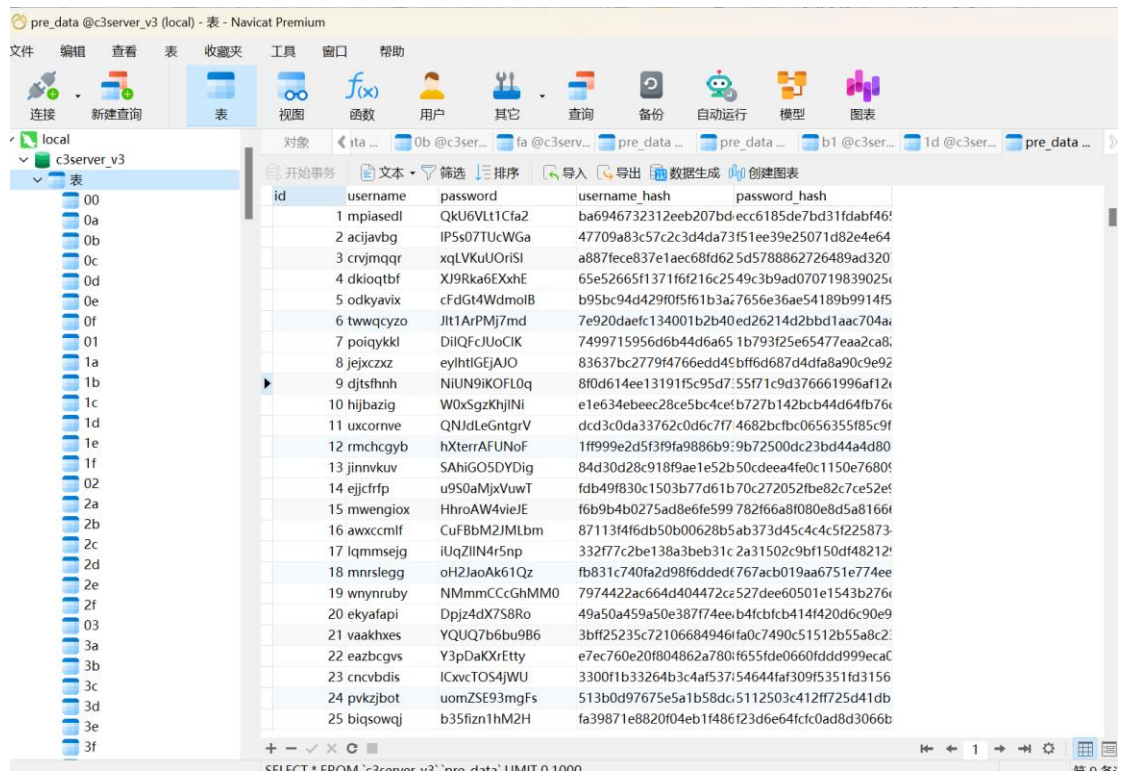
最后就是客户端在接收到 y 之后，对 y 取 r 次方根操作（ r 为客户端定义的盲因子），得到客户端输入的用户名和密码连接的 sha-256 运算后的 k 次方结果记为 $x_$ ，再将 $x_$ 与服务端发来的元数据列表相对应，若查找成功，返回“match”，若查找失败，返回“none”。

五、实验结果分析

服务器运行结果

```
C:\Users\土豆丝\Desktop\密码学综合实验\实验4>python server.py
服务器已启动，等待客户端连接...
请输入 密钥：7
数据库预处理 time: 23.41628861427307 seconds
服务器已启动，数据库预处理完成
数据库初始化完成
按下回车键继续...
成功接收x
成功接收b: ba
table_name:ba
```

在服务器中还利用 Navicat 生成了 1000 个随机数据库数据，生成的数据如图所示：



id	username	password	username_hash	password_hash
1	mpiasedl	QkU6VLt1Cfa2	ba6946732312eeb207bd-ecc6185de7bd31fdabf46f	
2	acjavbg	IP5s07TucWGa	47709a83c57c2c3d4da73f51ee39e25071d82e4e64	
3	crvmqqr	xqLVKuUOrISl	a887fece837e1aec68fd625d5788862726489ad320	
4	dkioqtb	XJ9Rka6EXhE	65e52665f1371f6f216c2549c3b9ad070719839025c	
5	odkyavix	cFdGt4WdmolB	b95bc94d429f0f5f61b3a27656e36ae54189b9914f5	
6	twwqcyzo	Jl1tArPMj7md	7e920daefc134001b2b40ed26214d2bbd1aac704a	
7	poiokykl	DilQFcJUoCIK	7499715956d6b44d6a651b793f25e65477eaa2ca8	
8	jejxczcx	eylhtlGEjAJO	83637bc2779f47f66edd45bfff6d687d4dfa8a90c9e92	
9	djtsfhn	NiUN9iKOFLOq	8f0d614ee13191f5c95d755f71c9d376661996af12c	
10	hijbazig	W0xSgzKhjINi	e1e634ebee28ce5bc4cefb727b142bcb44d64fb76c	
11	uxcornve	QNJdLeGntgrV	dcd3c0da33762c0d6c7f74682bcbfbc065635f85c9f	
12	rmchcgyb	hXterrAFUNoF	1ff999e2d5f3f9fa9886b959b72500dc23bd44a4d80	
13	jinnvkuv	SAhiG05DYDig	84d30d28c918f9ae1e52b50cdeea4fe0c1150e7680f	
14	ejjcftrp	u9S0aMjxVuWt	fdb49f830c1503b77d61b70c272052f8e82c7ce52e6	
15	mwengiox	HhroAW4vieJE	f6b9b4b0275ad8e6fe599782f66a8f080e8d5a8166f	
16	awxcmlf	CuFBbM2JMLbm	87113f4f6db50b00628b5ab373d45c4c4c5f225873	
17	lqmmsejg	iUqZiIN4r5np	332f77c2be138a3beb31c2a31502c9bf150df48212f	
18	mnrslgg	oH2JaoAk61Qz	fb831c740fa2d98f6ddedf767ac019aa6751e774ee	
19	wnynruby	NMmmCCcGhMM0	7974422ac664d404472c527dee60501e1543b276c	
20	ekyafapi	Dpjz4dX7S8Ro	49a50a459a50e387f74eeb4fcbfcb414f420d6c90e9	
21	vaakhxes	YQUQ7b6bu9B6	3bff25235c72106684946ffa0c7490c51512b55a8c2	
22	eazbcgvs	Y3pDaKXrEtty	e7ec760e20f804862a780f655fde0660fdd999ecaC	
23	cncvbdis	ICxvTOS4JWU	3300f1b33264b3c4af53754644faf309f5351fd3156	
24	pvkzjbot	uomZSE93mgFs	513b0d97675e5a1b58dc5112503c412ff725d41db	
25	biqsowqj	b35fznlhM2H	fa39871e8820f04eb1f486f23d6e64fcfc0ad8d3066b	

可以看到以不同 2 位 16 进制数表名的表，在实验过程中利用客户端生成的 b 与之匹配，并将对应的元数据表返回给客户端。

客户端运行结果如下：

```
C:\Users\土豆丝\Desktop\密码学综合实验\实验4>python client.py
请输入 用户名：mpiasedl
请输入 密码：QkU6VLt1Cfa2
成功接收 column
成功接收 y
x_:1.0915670831399042
none
```

由于是随机输入的用户名及密码，未能与数据库中的数据匹配，故这里返回 “none”，表示匹配失败。

4 dkioqtbF

XJ9Rka6EXxhE

65e52665f1371f6f216c2549c3b9ad0707198390256

```
请输入用户名: dkioqtbF
请输入密码: XJ9Rka6EXxhE
成功接收column
成功接收y
match
```

当输入数据库中存在的数据库时，客户端返回 match。

六、总结感想

通过本次实验，我熟悉了不经意伪随机函数的实现过程，熟悉了当前 c3 服务器的系统架构，同时学会利用 python 和 mysql 实现实验数据的生成，并对数据库进行查找操作，再一次对 udp 的通信传输工作进行了复习。本次实验依旧存在众多不足，由于实验的时间比较紧张，从构造数据到功能实现，实验完成的稍显仓促。

七、拓展部分

问：如果一个恶意的服务器想要返回错误的查询结果，客户端能察觉吗？思考并设计新的协议来解决这一问题，无需编程。

答：客户端无法察觉，客户端只能比对返回元数据列表的前几位字符，若不匹配则客户端可以发现是错误的匹配结果。若匹配，则客户端无法分辨该查询结果是否是正确的。可以采用让客户端对数据库直接的查询功能，对此查询功能做仅查询无法更改的限制。跳过服务端，以此就可以避免不诚实的服务端。