密码学课程第 4 次实验报告

实验名称： 口令泄漏查询密码协议系统构建

学号： 2011428 姓名： 王天行 班级： 密码科学与技术

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；
7. 数据报方式不限，可基于 TCP 或 UDP；
8. 程序界面不限，可使用命令行界面或图形化界面。
9. 实验环境不限，可基于 Linux/Windows 等

二、 实验内容说明

1. 服务器预处理文件，最终成功运行。
2. 服务器端设置一个以 c3server_v3 命名的数据库。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。
3. 启动服务器，可正常运行。
4. 随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。

三、 实验原理

1. 协议描述

1) 系统架构

系统由客户端和服务器组成。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。

2）安全模型

恶意客户端。恶意客户端可能希望通过查询获取另一个用户的口令。它可能知道目标的用户名，并有能力查询服务器。值得注意的是，服务器所使用的泄露数据集实际上已经被公开了，我们仍应该保护这些数据的私密性。

诚实但好奇的服务器。诚实即为它们不会伪造数据，将按照设定的协议执行，好奇则是指它们对用户的查询内容有一定程度的好奇，试图利用客户端的查询推测出更多的秘密信息，比如用户的口令。

3) 协议框架

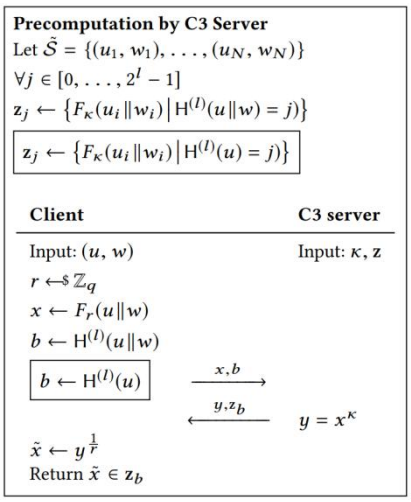


图 2 GPC 协议

本协议框架可以参考上图。

（1）创建数据库

Algorithm 1 CreateDatabase: Store a blinded and strongly hashed copy of all known breached credentials.

Require: $S = \{(u_1, p_1), \ldots, (u_n, p_n)\}$, $b = rand()$, and $n = 2$, a prefix length

```
function CREATEDATABASE(S, b, n)
    for (u_i, p_i) ∈ S do
        u'_i ← CANONCIALIZE(u_i)
        H ← HASH(u'_i, p_i)
        H^b ← BLIND(H, b)
        H_[0:n] ← BYTESUBSTRING(H, n)
        PARTITIONSTORE(H_[0:n], H^b)
    end for
end function
```

图 3

数据库预处理流程可参考图 3。

（2）客户端向服务器发送查询请求

数据库准备完毕后，用户通过与客户端执行私有集合交集（PSI），以实现信息的交互。用户先在客户端输入他的用户名和口令，客户端对输入凭证计算 x，同时计算该凭证预估所在的数据桶的桶标识符 b。并将两个数据发送给服务器。

（3）服务器向客户端发出响应

服务器用密钥 k 对来自客户端的加密凭证进行幂运算，得到 y，并根据传入的桶标识符 b 找到对应的数据桶 z_b，将 y 和数据桶 z_b 发送给客户端。

（4）客户端进行本地检验

客户端根据传入的 y 计算出 $\tilde{x}$。随后客户端检验 $\tilde{x}$ 是否在服务器传来的数据桶 z_b 中。若在数据桶中，则告知用户口令已泄露。否则，则告知用户口令没有泄露。

2.安全性分析

（1）数据安全性

在本系统中，为了防止敌手可能发起的离线字典攻击，服务器所维护的数据集与协议执行过程中所传输的数据均进行了加密，即使攻击者截获了客户端与服务器或者攻击服务器获取数据集，都无法获得明文信息，保证了数据的安全性。

（2）客户端隐私

查询者匿名性：服务器仅能从用户的查询请求中知道用户名的哈希前缀，无法在足够大的数据集中锁定用户的身份。我们可以通过控制所使用哈希前缀的长度来平衡查询者匿名性与服务器端存储数据桶的大小这两个因素。

查询内容隐私：服务器无法有效地获取所查询用户名或口令的明文。OPRF 的语义安全性和哈希函数的单向性保证了这一点。

查询结果隐私：服务器无法知道用户的凭据是否与其加密数据库中的条目匹配。这是通过 OPRF 的语义安全性来实现的，服务器无法获知 OPRF 输出的值，也就无法知道所查询的用户凭证是否与桶中元素相对应。

（3）服务器隐私

除了服务器数据库中是否存在以所输入用户名的哈希前缀为索引的桶，以及查询口令的散列值是否与桶中元素匹配之外，用户无法了解有关服务器数据库的任何信息。这是通过 OPRF 的语义安全性来实现的，由于用户不知道服务器端用于加密的私钥，即使得到桶中的数据，也只能看到一些随机数，无法解密出含义。

（4）对抗拒绝服务攻击（DoS）

攻击者可能不停地向服务器发送查询请求，耗费服务器的带宽或缓冲区，导致服务器不能正常提供泄露查询服务。

（5）对抗离线猜测攻击

服务器端数据库使用基于 2HashDH 的 OPRF 加密存储，没有服务器的私钥 k，攻击者无法有效地恢复出明文。

四、 实验步骤

1. 实验环境

编程语言：Python 3.8
程序界面：命令行界面
实验环境：Ubuntu20.04 LTS

2.pre.py 处理数据

```
def solve(self,u,p,b):#u username p pwd
    #u=hash((u,p))
    u=u+p
    m=hashlib.sha224()
    m.update(u.encode('utf-8'))
    m.digest()
    u=m.hexdigest()
    #print('u:\t',u)
    h=int(u,16)
    H=hashToCurve(h)
    print("H:\t",H)
    #取u前两个字节作为表名存储盲化后的用户名H
    #table_name=u.encode('utf-8')
    table_name=u[:2]
    table_name="bucket_"+table_name
    #TODO:将u进行盲化处理
    u=powMod(H,b,ep.p)
    #table_name=table_name.decode('utf-8', errors='ignore')
    print("table_name:\t",table_name)
    print("username:\t",u)
    #往表中插入数据：需判断表是否存在
    if self.table_exists(table_name) == 0:
        self._create_table(table_name)
    self._add_table_record(table_name,u)
```

```
def find(self):
    db = self.connect()
    cur = db.cursor()
    sqlQuery = "SELECT * FROM info"
    try:
        cur.execute(sqlQuery)
        results = cur.fetchall()
        b=10
        for data in results:
            username=data[0]
            pwd=data[1]
            print("username:"+username+" pwd:"+pwd)
            self.solve(username,pwd,b)
    except pymysql.Error as e:
        print("数据查询失败：" + str(e))
    finally:
        db.close()
```

主要思想：所用的初始数据存储在 test 数据库的 info 数据表中，从该表中读取数据，调用 solve 函数来处理数据，计算哈希值并进行盲化处理，然后根据哈希前缀进行数据桶的划分

并存储数据。
其中所涉及的其它函数：

hashToCurve（将哈希映射到椭圆曲线 NID_secp224r1 上）

```
class ECCParameters():
    def __init__(self, p, a, b, Gx, Gy, o):
        self.p = p
        self.a = a
        self.b = b
        self.Gx = Gx
        self.Gy = Gy
        self.o = o


ep = ECCParameters(
    p=0xffffffffffffffffffffffffffffffff000000000000000000000001,
    a=0xfffffffffffffffffffffffffffffffefffffffffffffffffffffffe,
    b=0xb4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4,
    Gx=0xb70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21,
    Gy=0xbd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34,
    o=0xffffffffffffffffffffffffffff16a2e0b8f03e13dd29455c5c2a3d)
```

```
def hashToCurve(x):

    #xBytes = hashlib.sha224(h.encode('utf-8')).digest()
    #x = int.from_bytes(xBytes, byteorder='big')
    print("x:\t",x)
    h=x
    for k in range(0, 100):
        # get matching y element for point
        y_parity = 0  # always pick 0,
        a = (powMod(x, 3, ep.p) + 7) % ep.p
        #print("a:\t",a)
        y = powMod(a, (ep.p + 1) // 4, ep.p)
        #print("before parity %x" % (y))
        if y % 2 != y_parity:
            y = ep.p - y

        # If x is always mod P, can R ever not be on the curve?
        try:
            R = Point(curve_secp224r1, x, y, ep.o)
        except Exception:
            x = (x + 1) % ep.p  # % P?
            continue

        if R == INFINITY or R * ep.o != INFINITY:  # is R * O != INFINITY check necessary?
            x = (x + 1) % ep.p  # % P?
            continue
        return R
    #print('hashToCurve failed for 100 tries')
    return h
```

powMod（模运算下的次方运算）

```
def powMod(x, y, z) -> int:
    # Calculate (x ** y) % z efficiently.
    number = 1
    while y:
        if y & 1:
            number = number * x % z
        y >>= 1  # y //= 2

        x = x * x % z
    return number
```

table_exists（表是否存在）

```
    def table_exists(self,table_name):        #这个函数用来判断表是否存在
        db = self._connect()
        cursor = db.cursor()
        sql = "show tables;"
        cursor.execute(sql)
        tables = [cursor.fetchall()]
        table_list = re.findall('(\'.*?\')',str(tables))
        table_list = [re.sub("'",'',each) for each in table_list]
        if table_name in table_list:
            db.close()
            return 1        #存在返回1
        else:
            db.close()
            return 0        #不存在返回0
```

_add_table_record（向 tablename 表中插入数据）

```
    def _add_table_record( self, tablename,username):
        db = self._connect()
        cursor = db.cursor()
        sql = "INSERT INTO " + tablename + "(username,server_index) VALUE (%s, %s)"
        value=(username,self.amount(tablename))
        try:
            cursor.execute(sql, value)
            db.commit()
            print('数据插入成功！')
        except pymysql.Error as error:
            print("数据插入失败：" + str(error))
            db.rollback()
        finally:
            db.close()
```

Amount（计算表 table_name 中已用多少个数据，返回下一项数据对应的序号）

```
#计算表table_name中已用多少个数据，返回下一项数据对应的序号
def amount(self,table_name):
    db = self._connect()
    cursor = db.cursor()
    sql="select count(server_index) from "+ table_name
    cursor.execute(sql)
    db.commit()
    result=cursor.fetchall()[0][0]+1
    print("result:\t",result)
    return result
```

3.server.py 服务器端查询工作

```
@server.route('/api/requests', methods=['POST'])
def requests():
    start = datetime.datetime.now()
    get_data = json.loads(request.get_data(as_text=True))
    tablename=get_data['table_name']
    H=get_data['H']
    print(H)

    if table_exists(tablename) == 0:
        print(tablename+' not exists')
        end = datetime.datetime.now()
        print("using time:\t",end-start)
        return jsonify(result="none")
    else:
        db = _connect()
        cur = db.cursor()
        sqlQuery = "SELECT * FROM "+tablename
        try:
            cur.execute(sqlQuery)
            results = cur.fetchall()
            for data in results:
                if int(H) == int(data[0]):
                    db.close()
                    end = datetime.datetime.now()
                    print("using time:\t",end-start)
                    return jsonify(result="match")
            end = datetime.datetime.now()
            print("using time:\t",end-start)
            return jsonify(result="none")
        except pymysql.Error as e:
            print("数据查询失败: " + str(e))

if __name__ == '__main__':
    server.run()
```

主要思想：使用 flask 框架将其运行在 http://localhost:5000/api/requests 上，对于客户端发送的数据进行处理，并将处理后的相关结果返回给客户端。

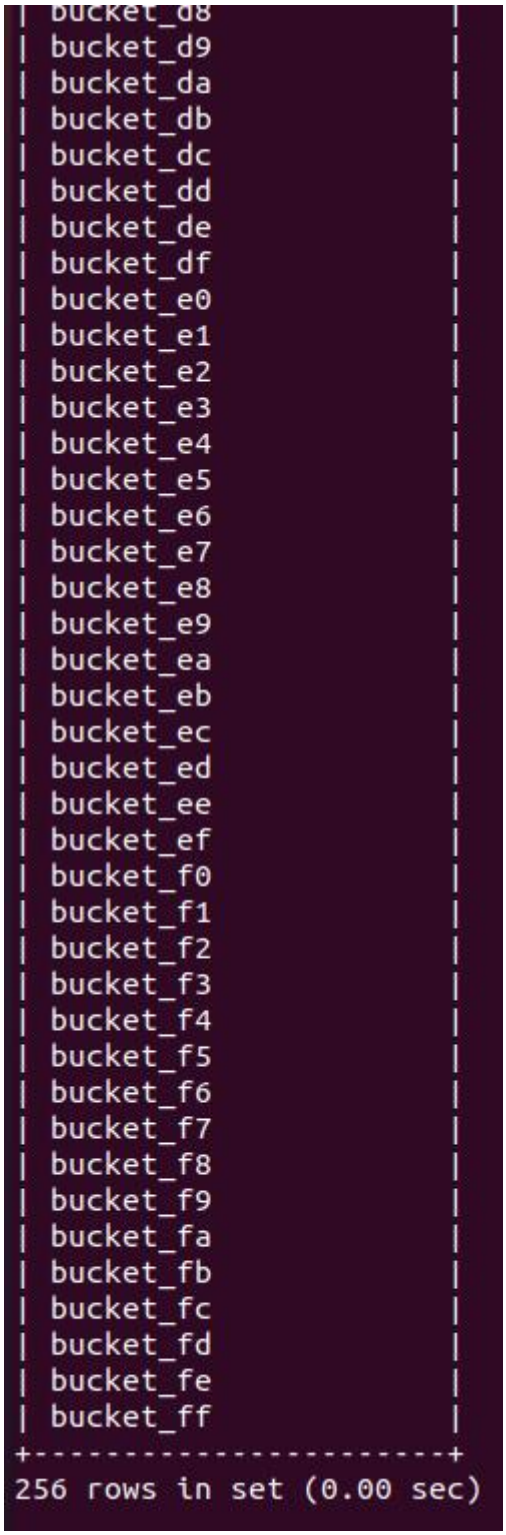
4.new_client.py 客户端查询

```
def request(tablename,H):
    url = "http://localhost:5000/api/requests"
    headers = {'content-type': 'application/json'}
    requestData = {"table_name": tablename, "H": H}
    ret = requests.post(url, json=requestData, headers=headers)
    if ret.status_code == 200:
        text = json.loads(ret.text)
        print(text)
    return text

print("please input username:")
u=input()
print("please input password:")
p=input()

start = datetime.datetime.now()

##client
b=10

u=u+p
m=hashlib.sha224()
m.update(u.encode('utf-8'))
m.digest()
u=m.hexdigest()
#print('u:\t',u)
h=int(u,16)
H=hashToCurve(h)
print("H:\t",H)
#取u前两个字节作为表名存储盲化后的用户名H
#table_name=u.encode('utf-8')
table_name=u[:2]
table_name="bucket_"+table_name
#TODO:将u进行盲化处理
u=powMod(H,b,ep.p)
#table_name=table_name.decode('utf-8', errors='ignore')
print("table_name:\t",table_name)
print("username:\t",u)

result=request(table_name,u)

end = datetime.datetime.now()
print("using time:\t",end-start)
##server
```

主要思想：客户运行客户端并输入想要查询的用户名和口令，和预处理数据相似地进行数据处理。对于服务器端返回的信息进行处理并输出结果。

五、 实验结果分析

1.处理数据后数据库的样子



由于虚拟机环境限定，尝试加入 10w 条数据，虚拟机会崩溃，所以数据没有使用特别多条，

仅插入 1w 条，所以选取哈希的前两位作为表名。

其中一张数据表的数据情况如下：

```
mysql> select * from bucket_61;
+------------------------------------------------------------------------+--------------+
| username                                                               | server_index |
+------------------------------------------------------------------------+--------------+
| 123622264769388456317455066760688052826528518988651057968442606805599 | 1            |
| 10379298749311128473356314262710457725591415593973507168455153243 22  | 2            |
| 139408173820112873463613253016090326274058908752150809977229519863 16 | 3            |
| 46673068256518325419924279264681349593690845395048226423739116182 10  | 4            |
| 21194229201083854303969470136729314488938747270723386657371619983 06  | 5            |
| 17369501549061363785482073432503690349264231313477830731371131314903 | 6            |
| 17689289407076498722054084974212085502084417201307718385759324998601 | 7            |
| 67817718255566756937415458356556831153483771392914362554917533779 56  | 8            |
| 13359736773910730612828878523362810372671839131160551061241787551 9   | 9            |
| 24811493216027697573210467185619530233731470257601412755746041104161 | 10           |
| 14371197464600072535685380959844218535445516970545879245936134221031 | 11           |
| 12327427743593697319047478550251296569086857186065724124448611027816 | 12           |
| 17139245811772584213981036535139328226973504246226000543663127992080 | 13           |
| 18496814080104651315195260568510735273615435481045205878684347738763 | 14           |
| 26437444080796766577590785452293275885370621429331937740375273637324 | 15           |
| 18173904506009123525029022386532279781409505806436427359396789178274 | 16           |
| 10679941373463912061072131760011310670051715836267983396290135633409 | 17           |
| 89014785154636565101530355286746496776608765921418207466971160493 81  | 18           |
| 48238703277914171953505572749175439423185006013656904770248643046 22  | 19           |
| 13311085321736551977842042953200749394736505937106865400335566037064 | 20           |
| 98837060878104880723109475728473690742279666641120077058015684503 72  | 21           |
| 10626944061349862983764782115732242286796964252771040468866013648 85  | 22           |
| 23334184945898054733733601393107621452852837255159888438834305620088 | 23           |
| 14743758550907249960338620202559736725047746893492820011514125681599 | 24           |
| 17178230450043272526712054941928203556711891620058000081048019296613 | 25           |
| 16800474005677986291801659590250152541622231170040091083814155072193 | 26           |
| 17552066309567815852407795337994481366838007287004035880193234999970 | 27           |
| 70828664736936536870784671785679681063467062868639809161555416121 53 | 28           |
| 12466643479218593130357478016345764002738123747260032797582242880940 | 29           |
| 15458818162105182658997872973265690912844133728657764493329339738007 | 30           |
| 21546271492646831555606680648355311293792100590695859054352305881586 | 31           |
| 17349938307933185037728513411627535560002831499151498675888593155 53  | 32           |
| 25888270783171311447698460836066585141483295007637965338455410793345 | 33           |
| 63935395870960419547200479824210803712472201499696127764534283964 86  | 34           |
| 20005239123833880050101119628323190155671573159087244998915681561732 | 35           |
| 72182815839438487572890240686492498663977791121727246110467755295 79  | 36           |
| 10202093301873394249905538510393291391619044364066468224170979118257 | 37           |
```

2.被泄露的情况下：

Client

```
wtx@ubuntu:/mnt/hgfs/code-2$ python new_client.py
please input username:
1000
please input password:
1000
x:       102254318931913793625930817097503759002583477886661077340434925816 53
H:       102254318931913793625930817097503759002583477886661077340434925816 53
table_name:      bucket_61
username:        123622264769388456317455066760688052826528518988651057968442606
80599
{'result': 'match'}
using time:      0:00:00.033306
```

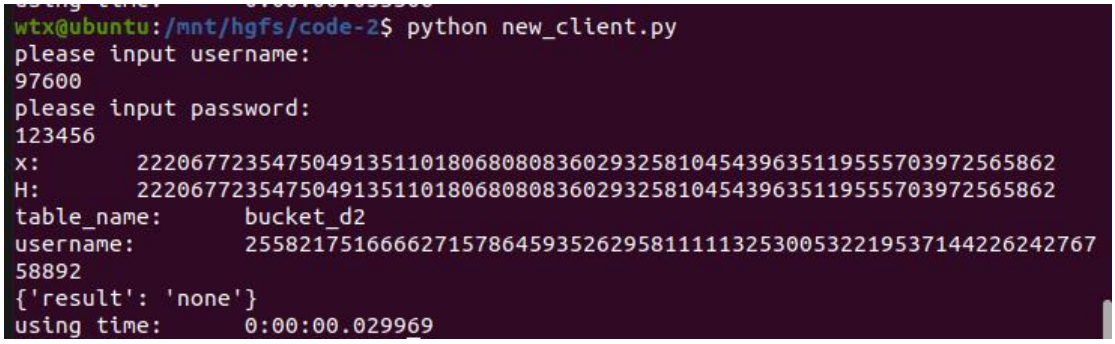
客户端接收到 match 信息，标志该口令被泄露，用时 33.306ms

Server

服务器端运行在 http://127.0.0.1/5000，用时 12.373ms。数据存储在 mysql 数据库中。

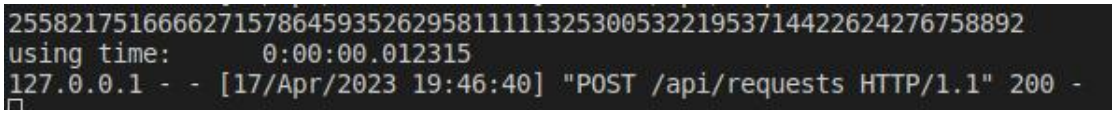
3.没有被泄露的情况下：

Client



客户端接收到 none 信息，标志该口令没有被泄露，用时 29.969ms

Server



服务器端运行在 http://127.0.0.1/5000，用时 12.315ms。

六、 总结感想

Q.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。

A.客户端由于没有对于返回信息进行验证处理，所以无法察觉。

新的协议：服务器对是否泄露的结果进行签名，并将签名后的结果发送给客户端，客户端在

确认服务器身份后选择接收该消息。

实验遇到的问题：

1. server.py 中用户名和表中数据比较时，两个数据打印出来一致，但是比较结果为 FALSE：两个数据类型不一致，都转为 int 型进行比较
2. Flask 框架中对于数据表的查询不支持表名可变的情况：改为使用 pymysql 来进行数据库相关操作