CS426 hw9 Writeup

Cheng-En Lee

Dictionary Attack

Using john the ripper and the provided rockyou.txt wordlist, I cracked the password hashes of the three users in shadow.txt with a dictionary attack. The rest of the password hashes I cracked them with the same method, but without using a wordlist:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9$ john --format=crypt dictionary/shadow.txt --wordlist=dictionary/rockyou.txt
Loaded 5 password hashes with 5 different salts (crypt, generic crypt(3) [?/64])
Will run 8 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
123456           (kazem)
password         (qi)
qwerty123        (milad)
3g 0:00:11:21 2% 0.004402g/s 576.2p/s 1156c/s 1156C/s rock111..rob4eva
Use the "--show" option to display all of the cracked passwords reliably
Session aborted
(base) daniel@daniel-ubuntu:~/Downloads/hw9$ john --format=crypt dictionary/shadow.txt
Loaded 5 password hashes with 5 different salts (crypt, generic crypt(3) [?/64])
Remaining 2 password hashes with 2 different salts
Will run 8 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
cs426            (cs426)
root             (root)
2g 0:00:00:00 100% 1/3 12.50g/s 1193p/s 1200c/s 1200C/s root..root999999
Use the "--show" option to display all of the cracked passwords reliably
Session completed
(base) daniel@daniel-ubuntu:~/Downloads/hw9$
```

Here is the password.csv

```
username,password
root,root
cs426,cs426
milad,qwerty123
qi,password
kazem,123456
```

Mitigations

If I am the system administrator, I would set proper access controls so /etc/shadow could not be easily leaked. Additionally, I would enforce a strong password policy (Eg. more than 12 characters, must have uppercase, lowercase, number, special characters) that ensures the users' password is long and hard to brute-force. If space allows, I would also keep a list of frequently used passwords and use it to stop users from using a frequently used password.

Length Extension Attack

Length extension attack allows an attacker to forge a hash that represents a longer message without knowing prior message content. Let's assume `m` is the message text that derives to the known hash value provided by the token argument and `x` is the argument I want to inject (`&command3=UnlockAllSafes`). By directly using the compression state of the original hash and providing a padding afterwards to finish the current block, I can calculate the value of longer hashes that contains the injected parameter without knowing the prefix that the original hash represent (`H(m | pad | x)`).

However, without modifying other parts of the url, it is impossible to provide a good pad value to the hash calculation on the server. Thus, besides updating the token value and adding the third command, I modified the value of the command2 argument to include url encoded padding at the end. At server side, the message text becomes

`m' = <8-byte-password>user=<user>&command1=<command1>&command2=<command2><encoded_padding>&command3=UnlockAllSafes`. Calculating the hash `H(m')` would be `H(m | pad | x)`, which matches the new hash value I calculated.

Here is the script for crafting the new url with hash including command 3:

```
import http.client, urllib.parse, sys, urllib.request, urllib.parse, urllib.error
from pymd5 import md5, padding

url = sys.argv[1]

# Parsing parameters in the url
url_object = urllib.parse.urlparse(url)
query_dict = urllib.parse.parse_qs(url_object.query)
original_hash = query_dict["token"][0]

# We know the password is always 8 characters long
msg_length = 8
for key, value in query_dict.items():
   if key == "token":
       continue

   # For each query parameter, also count for '&' and '='
   msg_length += len(key) + len(value[0]) + 2

# Remove the leading '&' that isn't supposed to be counted
msg_length -= 1

# We only need the length of the message to calculate padding
pad = padding(msg_length * 8)

# Performing length extension attack
bits = (msg_length + len(pad)) * 8
new_hash = md5(state=bytes.fromhex(original_hash), count=bits)
suffix = "&command3=UnlockAllSafes"
new_hash.update(suffix)

# Updating parameters
query_dict["token"] = [new_hash.hexdigest()]
query_dict["command3"] = ["UnlockAllSafes"]
query_dict["command2"][0] = query_dict["command2"][0].encode() + pad


# Constructing new url
query_string = urllib.parse.urlencode(query_dict, doseq=True)
url_object = url_object._replace(query=query_string)
new_url = urllib.parse.urlunparse(url_object)

print(new_url)
```

The script successfully tricks the server into unlocking all safes as the specified user.

Mitigations

If I am the creator of this server, I would not use MD5 for HMAC since its length extension property can potentially create undesired vulnerabilities. Instead, I would use length-extension-resistant hash functions such as SHA-3. Moreover, I would sanitize the query string from the url by removing all the null or other non-ascii bytes that are crucial for constructing the padding.

MD5 Collisions

Answer to questions:

Q1: Generate your own collision with this tool. How long did it take?

Roughly 2.993 seconds

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ time ./fastcoll -o file1 file2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'file1' and 'file2'
Using initial value: 0123456789abcdeffedcba9876543210

Generating first block: ................
Generating second block: W.......................
RUnning time: 2.97774 s

real    0m2.993s
user    0m2.983s
sys     0m0.009s
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

Q2: What are your files? Copy and paste the content of the hex dump files. Are they the same or different?

The two hex dumps are similar but different.

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ xxd -p file1
aba19ba2608e77fed3d3d403d7ec5fba2bfab645c25890db1a4a8411448c
bd138b0aefe6ee483253c71b3bdc1d68958fa439b8218066aea71d775edf
ae2d19f5102be987640d0befcf9ed88b139868edc519dc35011ac33cf2bc
b586ce2cedbd0d8876a22a0eaa26923d158288f4567dfda0356d81d0c3a1
1ee4a8ea87ae9835
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ xxd -p file2
aba19ba2608e77fed3d3d403d7ec5fba2bfab6c5c25890db1a4a8411448c
bd138b0aefe6ee483253c71b3bdc1de8958fa439b8218066aea71d775e5f
ae2d19f5102be987640d0befcf9ed88b139868edc519dcb5011ac33cf2bc
b586ce2cedbd0d8876a22a0eaa26923d15828874567dfda0356d81d0c3a1
1ee4a86a87ae9835
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ diff <(xxd -p file1) <(xxd -p file2)
1,5c1,5
< aba19ba2608e77fed3d3d403d7ec5fba2bfab645c25890db1a4a8411448c
< bd138b0aefe6ee483253c71b3bdc1d68958fa439b8218066aea71d775edf
< ae2d19f5102be987640d0befcf9ed88b139868edc519dc35011ac33cf2bc
< b586ce2cedbd0d8876a22a0eaa26923d158288f4567dfda0356d81d0c3a1
< 1ee4a8ea87ae9835
---
> aba19ba2608e77fed3d3d403d7ec5fba2bfab6c5c25890db1a4a8411448c
> bd138b0aefe6ee483253c71b3bdc1de8958fa439b8218066aea71d775e5f
> ae2d19f5102be987640d0befcf9ed88b139868edc519dcb5011ac33cf2bc
> b586ce2cedbd0d8876a22a0eaa26923d15828874567dfda0356d81d0c3a1
> 1ee4a86a87ae9835
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

Q3: What are their MD5 hashes? Are they the same or different?
The MD5 hashes are the same.

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ md5sum file1 file2
a358fe7d6c9a2de893d7e2292542a689  file1
a358fe7d6c9a2de893d7e2292542a689  file2
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

Q4: What are their SHA-256 hashes? Are they the same or different? What does this tell us about MD5 (just one sentence-long explanation is sufficient)?
The SHA256 hashes are different. This shows that MD5 is prone to collision attacks since it is too short (only 128 bits) and efficient algorithms exist that find MD5 collisions.

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ sha256sum file1 file2
6eda107c53672269f30d09f3ade50f66d4ec725cab0c9cbfee712703a907de33  file1
f3d4e3c412a5de8e92885b7e9dff9b4c56741344e043816eaa4ef9bc59d3ef62  file2
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

The Challenge:
From the provided example, I understood that fastcoll can be used to create two bash scripts of different behaviors that have the same MD5 hash. Combining the given prefix and suffix, the bash script takes the binary value generated by fastcoll and calculates its sha256 hash. With that as an inspiration, I created a python equivalent pair of prefix and suffix:

Here is prefix.py:

```
#!/usr/bin/python
# -*- coding: latin-1 -*-
data = """
```

here is suffix.py:

```
"""

import hashlib
hasher = hashlib.sha256()
hasher.update(data.encode("latin-1"))
digest = hasher.hexdigest()
print(digest)
```

I proceeded to generate the two different files with the same hash:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ time ./fastcoll -p prefix.py -o col1 col2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'col1' and 'col2'
Using prefixfile: 'prefix.py'
Using initial value: 475eeb322ae9b369edc7de7bcec90952

Generating first block: ................................................................
Generating second block: S01.....................
Running time: 16.5052 s

real    0m16.531s
user    0m16.498s
sys     0m0.017s
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

Since calculating the extended message text for MD5 is doable even if I only have the hash of the previous message text, adding the same suffix to the two python code segments with the same hash would also ensure that their hash equals to each other. After the suffix code is concatenated, I can see that the sha256 digest for the value generated by fastcoll for col1 is different from that of col2, and it is independent of what the value of suffix is:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ cat col1 suffix.py > good.py; cat col2 suffix.py > evil.py
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ python3.9 good.py
831607f305f761819f8d3ed17b8344bd763c80224082 7bbb5f25d1340c5f61c9
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ python3.9 evil.py
e186b3d62de9f9489e19f812978cdd5cd3325a96f230f4da3e3432fed53f9981
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

Given this observation, I can create an if statement in the suffix that checks whether the sha256 digest is the value that a good program should have. This means that based on different hashes, the two programs could have different lines executed. Here is the updated suffix.py:

```
"""

import hashlib
hasher = hashlib.sha256()
hasher.update(data.encode("latin-1"))
digest = hasher.hexdigest()

if digest == "94ba3429d1d5d5ac656fc8669c98cd7935b7f95301d0076e812edf53a44bc345":
   print("I mean no harm.")
else:
   print("You are doomed!")
```

As I rebuild my two python scripts and run them, I can see these two scripts prints out different strings:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ cat col1 suffix.py > good.py; cat col2 suffix.py > evil.py
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ python3.9 good.py
I mean no harm.
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ python3.9 evil.py
You are doomed!
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

While exhibiting different behavior and different sha256 digests for their file content, the two scripts have the same MD5 digest:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ md5sum good.py evil.py
598f5de0e0debb6ad4ec2aef295511ba  good.py
598f5de0e0debb6ad4ec2aef295511ba  evil.py
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$ sha256sum good.py evil.py
987ef84f9b2506a071e3a85941d43a282078910286b6cae028671e0465e6ce44  good.py
60a5df49c3d7c9594818023d63bd0f0670eeca18e21c555c9f5896045aefe381  evil.py
(base) daniel@daniel-ubuntu:~/Downloads/hw9/md5$
```

‘Leaky’ RSA

By reading the provided source code carefully, I realized that the server allows me to regenerate p two times and view the encrypted flag a single time. However, it does not regenerate q as it regenerates p. Based on this observation, I could create these equations:

1. $s_1 = p_1 - q$
2. $s_2 = p_2 - q$
3. $r_1 = p_1 \mod q$
4. $r_2 = p_2 \mod q$

whereas p_1 and p_2 are first and second p values generated by the server. s_1 and s_2 are known values obtained from asking the server the value of p - q. Similarly, r_1 and r_2 are also known values, obtained from asking the server the value of p % q.

By substitution, we can further simplify the third and fourth equation as follows:
Equation 3:

$$r_1 = (q + s_1) \mod q$$
$$r_1 = s_1 \mod q$$
$$s_1 = k_1 q + r_1$$
$$s_1 - r_1 = k_1 q$$

Equation 4:

$$r_2 = (q + s_2) \mod q$$
$$r_2 = s_2 \mod q$$
$$s_2 = k_2 q + r_2$$
$$s_2 - r_2 = k_2 q$$

whereas k_1 and k_2 are some unknown integer quotients.

With the two simplified equations, we can see that q is both a factor of $s_1 - r_1$ and $s_2 - r_2$. As a result, it must also be a factor of $gcd(s_1 - r_1, s_2 - r_2)$. Though the q value has 512 bits, given the low time complexity of the Euclidean’s algorithm, we can still compute the gcd realistically.

There can be multiple possible values of q since there could be more than one factor in $gcd(s_1 - r_1, s_2 - r_2)$.
Here is a Python script to calculate the gcd value. s_1, s_2, r_1, r_2, and c (encrypted flag) are all collected by running the server source code locally in my computer:

```
from Crypto.Util.number import long_to_bytes
```

```
s1 =
125848899601803850622397879195293729485921116793645545230435460796865032411131470089544034636581047068382433679596468615996291342378464545988836452464916136674170514239421678872950575323566275202710691384157819660727862642056409402530584139493339689202109138891278191211198482167040415492219980023678845242028
r1 =
113193418856849981888398922299381622048622783827655764642753718787757314583338317938987022433523471588877502520553185290255902060740435737255616224455523 25
s2 =
944376151911047796798215501414432659075593901475628330373067553440054029914349837110787603775296537037044603601880134923692435724896162361647624342954095295028194741862116614709314717551402368025698406516933159124801734641241711999042575386453447562307901416028441107592491832777982513622945430900838133825 98
r2 =
383724180585848577646048532096139186598144867106621288968076448292302184910478388004518878893094442014548906889084409055977056822131828529699058379903053 9
c =
369054704483741321175840339585533837311388715388851083299467667112683782939147726944217282961417448842835775151863819726517576435857967966788580710163032646201547585156949044382860836339444735141223307452551020960184828678880032037699334392277471358179609812957976597323375505743797374471102707084740002571685371824208671390267195399764557709014179986932349177292137474272493201516331923302277402155771257416052227045359270693070455208207044744838572087363405268
e = 65537

def gcd(a, b):
   if a == 0:
       return b

   return gcd(b % a, a)

q = gcd(s1 - r1, s2 - r2)
print(bin(q)[2:], len(bin(q)[2:]))
```

We can see a 512 bit number is generated for q, with MSB being 1:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$ python gcd_calc.py
1111010011000100100011010000010100110011100111011011110000100011100001101011110011101000100010010000000000010110011000001100001011110111111110010101001101010110000001110011010101011011101100101110000011110001100011010000100010101011100110101101011110011000110101110111100110100101100010100000101010111011111011001001000101010110111010001001110101111000101000000010001011000111111011001001101001100100000111000100110101001100010111111111110011111110001011001001011010011101101101110010001001001110010110100000111010 1 512
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$
```

If q is not $gcd(s_1 - r_1, s_2 - r_2)$, then q must be $\frac{gcd(s_1 - r_1, s_2 - r_2)}{k}$ such that k is an integer greater than 1. If q is divided by 2, its bits would be shifted towards the right by one. Since getPrime(512) in my own computer always gives a 512 bit number with MSB of 1, q could not be bit shifted, thus cannot be divided by $k \geq 2$. Therefore, q must be $gcd(s_1 - r_1, s_2 - r_2)$.

Here is the updated Python script for decryption:

```
from Crypto.Util.number import long_to_bytes

s1 =
```

```
125848899601803850622397879195293729485921116793645545230435460796865032411131470089544034636581047068382433679596468615996291342378464545988836452464916136674170514239421678872950575323566275202710691384157819660727862642056409402530584139493339689202109138891278191211198482167040415492219980023678845242028
r1 =
113193418856849981888398922299381622048622783827655764642753718787757314583338317938987022433523471588877502520553185290255902060740435737255616224455523 25
s2 =
944376151911047796798215501414432659075593901475628330373067553440054029914349837110787603775296537037044603601880134923692435724896162361647624342954095295028194741862116614709314717551402368025698406516933159124801734641241711999042575386453447562307901416028441107592491832777982513622945430900838133825 98
r2 =
383724180585848577646048532096139186598144867106621288968076448292302184910478388004518878893094442014548906889084409055977056822131828529699058379903053 9
c =
369054704483741321175840339585533837311388715388851083299467667112683782939147726944217282961417448842835775151863819726517576435857967966788580710163032646201547585156949044382860836339444735141223307452551020960184828678880032037699334392277471358179609812957976597323375505743797374471102707084740002571685371824208671390267195399764557709014179986932349177292137474272493201516331923302277402155771257416052227045359270693070455208207044744838572087363405268
e = 65537

def gcd(a, b):
   if a == 0:
       return b

   return gcd(b % a, a)

# q must divides gcd. q is probably gcd since it has 512 bytes
q = gcd(s1 - r1, s2 - r2)

p2 = q + s2

d = pow(e, -1, (p2 - 1) * (q - 1))
m = pow(c, d, p2 * q)

print(long_to_bytes(m))
```

The script now successfully decrypts the test flag without knowing it:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$ python gcd_calc.py
b'REDACTED'
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$
```

I proceeded to collect s_1, s_2, r_1, r_2, and c on the actual server:
Here are the values of s_1 and r_1:

```
lee4649@data:~$ nc armor3.cs.purdue.edu 8080

Welcome to CS 426 HW9.

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
> 2

> p - q = 109781258867800243642385184813296939763352140533258754339101751566822605644403967200384284016734
331754927009646267403127410969141075217912228703921843740294160119339126250041814056741556899629570478528
816822733316898530357321823738087334676097745992598242001272864553465141667040686965918129654265949893106 68
4

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
> 3

> p % q = 40867973447536533398049420852005444884324828659418169382918819963532655934013803516380741815545 5
1485621995749183176787096187353941913356150008877626133 68

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
>
```

Here are the values of s_2 and r_2:

```
4. Get (encrypted) flag
5. Check Answer
6. Exit
> 1

New p and n (but not q (!)). Hot and fresh out of the CPU!

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
> 2

> p - q = 144828824668121856207204808841192592402466831392203005844287904108530497945052144429706578017255
362401972094933918412612074195239862480549607406713783982793332754195930865932196225557500126727641505907 6
643872591673385348810895672256405685822852786597502783052212199580936730148481233135585210478146433882399 3
4

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
> 3

> p % q = 366187832301005046314543731015457998496460802824954800260443852634347598372180113280634775288725
681239099191445062639111486270448746865816881397341653075

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
>
```

Here is the value of c (encrypted flag):

```
1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
> 4

c = 795680133850450298532555756829549111871227553219447811436262554706245347453725373839653574575038653637
434029641933369489016631082853814284980505266507783217979005102662964203109375625718415912341562981196425 5
593527699160706694883161340141925046885972308391720980096371439542876844050220766513050447059391986025138 4
472148636576853726255544358949426458673161793097942855264394696106475406438001099124811773350340496327178 4
796400370297589823637562273556951554883853

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
>
```

However, after I run my script with this collected information, I saw malformed decrypted data:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$ python gcd_calc.py
b'\x04\xdb\xb3\x89U\xda8\xdc4\xd4N\x8a\x8b\xe8m{\xf9\xf3v\x15\xde\x8e\xb8\xa3\xd1\xfd\x97\x94\x94\x87GX\xe6\xf8\xa5TF\xf5\xfc\xb8I\xb5\x15\x9f\x15\xbc\xe0\x86m\x96\x82
\xd2\xf1Y<\x9ee\xa2\x9ca\r\xdd\t|I\x0c\xfc\xa8\x1a\xfbBl3\xda\xc5\x87]\xac\xa8\xb7\xd3\x00\x1e"*\x0b\n\xc7\x96\x0cs\xea2\xc1\xf5\x92\x04\x15\xe0\xe9CO\xf1\rf\x98\xa8\x
96\xa92a\xdc\'\xa0\x8e\xc7\xe8B\x8c\x0cx\x8f~\xad\xffCI\xafg\xe9\x12SHy\xa4WM\x00\x95)\xfd-Z\xb67b\xd68%$%=_\xa9\x12\xd3HU57\xa9\x9f\xc8A\xab\xf8\xcb\x860Q\x9d\xdc/\xb
e\x03\xff*B\xcfM\x1b+\x0b\x0244\x85yZ\x1a\x04\xf2\xa4'
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$
```

By careful verification I confirmed that $q = gcd(s_1 - r_1, s_2 - r_2)$ does not apply on the server. Since q is known to be large, I believed that the quotient k that makes $gcd(s_1 - r_1, s_2 - r_2)$ divisible by q must be easily brute-forcible. To find q, I modified the script to brute force the factor k:

```
from Crypto.Util.number import long_to_bytes

s1 =
109781258867800243642385184813296939763352140533258754339101751566822605644403967200384284016734331754927009646267403127410969141075217912228703921843740294160119339126250041814056741556899629570478528816822733316898530357321823738087334676097745992598242001272864553465141667040686965918129654265949893106684
r1 =
408679734475365333980494208520054448843248286594181693829188199635326559340138035163807418155455148562199574918317678709618735394191335615000887762613368
s2 =
144828824668121856207204808841192592402466831392203005844287904108530497945052144429706578017255362401972094933918412612074195239862480549607406713783982793332754195930865932196225557500126727641505907664387259167338534881089567225640568582285278659750278305221219958093673014848123313558521047814643388239934
r2 =
366187832301005046314543731015457998496460802824954800260443852634347598372180113280634775288725681239099191445062639111486270448746865816881397341653075
c =
795680133850450298532555756829549111871227553219447811436262554706245347453725373839653574575038653637434029641933369489016631082853814284980505266507783217979005102662964203109375625718415912341562981196425559352769916070669488316134014192504688597230839172098009637143954287684405022076651305044705939198602513844721486365768537262555443589494264586731617930979428552643946961064754064380010991248117733503404963271784796400370297589823637562273556951554883853
e = 65537

def gcd(a, b):
   if a == 0:
       return b

   return gcd(b % a, a)

k = 1
gcd_result = gcd(s1 - r1, s2 - r2)

while k <= gcd_result:
   q = gcd_result // k

   p2 = q + s2

   d = pow(e, -1, (p2 - 1) * (q - 1))
   m = pow(c, d, p2 * q)
   text = long_to_bytes(m)

   # We know the flag must start with "CS426"
   if b"CS426" in text:
       print(f"k={k}: {text}")
       break
```

```
    k += 1
```

Running the script successfully recovers the flag:

```
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$ python gcd_calc.py
k=23: b'CS426{W311_D0N3_M1L4D_Q1_4ND_K4Z3M_4R3_PR0UD_0F_Y0U!}'
(base) daniel@daniel-ubuntu:~/Downloads/hw9/rsa$
```

```
1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
> 5

Are you done? Nice! What did you get?
Input format: CS426{WHATEVER}
CS426{W311_D0N3_M1L4D_Q1_4ND_K4Z3M_4R3_PR0UD_0F_Y0U!}
You said:  CS426{W311_D0N3_M1L4D_Q1_4ND_K4Z3M_4R3_PR0UD_0F_Y0U!}
Terrific. That is correct! GG!

1. Regenerate prime p
2. Print p - q
3. Print p % q
4. Get (encrypted) flag
5. Check Answer
6. Exit
>
```

Mitigations:

If I am the creator of the server, I would make sure that both p and q would be generated every time when requested. By doing this, the system of equations is no longer valid. Thus, this removes the possibility for the user to brute force the private key by exploiting the fact that q is a factor of $gcd(s_1 - r_1, s_2 - r_2)$. Even better, I would remove the functionality to show p - q and p % q if it is unnecessary. The vulnerability suggests that the security of RSA (or potentially other asymmetric encryption protocols) relies on big p, q values, as well as no reuse of any of them in later encryptions.

Collaboration Statement

Individuals consulted:

- None

Online resources used:

- None