ECE 4802, Project 5

Calvin Figuereo-Supraner

December 2 2016

Versioning

```
$ python3 --version
Python 3.5.2
```

Usage

```
$ ./q1.py
$ ./q2.py
$ ./q3.py
```

Packages

The program `q3.py` requires the package below.

```
$ pip3 install gensafeprime
```

Problem 1

Script output:

```
$ ./q1.py
8.4.1:   1584
8.4.2:   0.31877641376534516
8.4.3:   1005
```

1a

The number of generators in $\mathbb{Z}_n^*$ is $\phi(n-1)$. Compute $\phi(4968)$ by:

$$|\mathbb{Z}_{4968}^*| = \boxed{1584}$$

1b

The probability of a randomly chosen element $a \in \mathbb{Z}_{4969}^*$ being a generator is computed by:

$$\frac{|\mathbb{Z}_{4968}^*|}{|\mathbb{Z}_{4969}^*|} = \boxed{0.318}$$

1c

The script `q1.py` gives $a = \boxed{1005}$.

Problem 2

Script output:

```
$ ./q2.py
           x            b            e            m
           1       235973       456789       583903
      235973       514940       228394       583903
      235973         5434       114197       583903
       26294       333206        57098       583903
       26294         2501        28549       583903
      364158       415971        14274       583903
      364158       393433         7137       583903
       79207       343607         3568       583903
       79207       583849         1784       583903
       79207         2916          892       583903
       79207       328414          446       583903
       79207       112751          223       583903
      455975        51885          111       583903
      265024       260395           55       583903
       12813       404053           27       583903
      247091       131912           13       583903
      218629       466344            6       583903
      218629       302277            3       583903
      376693       491580            1       583903
      418744       304238            0       583903
           x            b            e            m
           1 984327457683   2153489582 994348472629
           1 751837619932   1076744791 994348472629
751837619932 133647923556    538372395 994348472629
630996993289 702432827587    269186197 994348472629
981214015219 408284410699    134593098 994348472629
981214015219 518938198018     67296549 994348472629
466407490241 805940334697     33648274 994348472629
466407490241 194448987220     16824137 994348472629
921153283301 494888873439      8412068 994348472629
921153283301 255946549071      4206034 994348472629
921153283301 921556030333      2103017 994348472629
268488152375 990196147175      1051508 994348472629
268488152375  47198554029       525754 994348472629
268488152375 467841164647       262877 994348472629
644809215284 865816798398       131438 994348472629
644809215284 501639319341        65719 994348472629
 26337228643 891167495425        32859 994348472629
895642694818 467043903631        16429 994348472629
404756218143 622045366798         8214 994348472629
404756218143 522760780235         4107 994348472629
281255078862 803664929813         2053 994348472629
502271318414 852782121265         1026 994348472629
502271318414 407646260658          513 994348472629
 33851880051 760374205862          256 994348472629
 33851880051 277397929319          128 994348472629
 33851880051 535426710156           64 994348472629
 33851880051 627846812638           32 994348472629
 33851880051  48601339780           16 994348472629
 33851880051 386874218949            8 994348472629
 33851880051 560712679276            4 994348472629
 33851880051 484659064418            2 994348472629
 33851880051 551560475289            1 994348472629
331688688384 577264372846            0 994348472629
Success!
```

$$235973^{456789} \mod 583903 = \boxed{418744}$$
$$984327457683^{2153489582} \mod 994348472629 = \boxed{331688688384}$$

Problem 3

Script output:

```
$ ./q3.py
3ab8f731afa805584021cfeeb7f702ab7eb6d5cc4737fa94f32f440a
```

3a

Using the same PRNG to generate the parameters A and B is okay, because an attacker does not have enough of the PRNG output cycle to predict B based on A.

3b

The prime p does not have to be random, because the security of p depends on the ability to solve the discrete logarithm problem. If that problem cannot be solved, it is safe to reuse p; RFC 3526 even lists some popular choices for p in DHKE.

3c

Safe primes have 2 as a primitive root, so $g = 2$ can be used as a generator of order $p - 1$ for DHKE (I don't know how to follow the proof for 2 as a primitive root, but it holds true regardless).

3d

Alice's a and Bob's b must be chosen such that $a, b \in \{2, \dots, p - 2\}$.

3e

The script `q3.py` applies SHA-224 to get a symmetric-sized key.

Problem 4

4a

The ElGamal encryption scheme is non-deterministic because the computation of the ephemeral key k_E involves modular exponentiation by a randomly selected exponent i. This means there are multiple possible encryptions for the plaintext x.

4b

If the key k_M is randomly drawn from $\mathbb{Z}_p^*$, every ciphertext $y \in \{1, 2, \ldots, p-1\}$ is equally likely. Therefore, $p-1$ valid ciphertexts exist for each message m. In problem 8.13, $p = 467$, so there are 466 valid ciphertexts.

4c

Given the first message-ciphertext pair $\langle m_1, c_1 \rangle$, and the fact that $c_1 = m_1 * k_M \mod p$, k_M can be recovered multiplying both sides by m_1^{-1}. Once the attacker has k_M, they can compute m_2 from c_2 using the equation $m_2 = c_2 * k_M^{-1} \mod p$.

Source Code

q1.py

```
#!/usr/bin/env python3

from fractions import gcd

def multGroup(n):
    return {x for x in range(n) if gcd(x, n) == 1}

def totient(n):
    return len(multGroup(n))

def yieldGen(bound, n):
    z_n = multGroup(n)
    for a in range(bound, n):
        if {pow(a, p, n) for p in z_n} == z_n:
            yield a

p841 = totient(4968)
p842 = p841 / 4969
p843 = next(yieldGen(1000,4969))

print("8.4.1\t{}".format(p841))
print("8.4.2\t{}".format(p842))
print("8.4.3\t{}".format(p843))
```

q2.py

```
#!/usr/bin/env python3

def wrapper(a, b, c, d):
    print('{:>12} {:>12} {:>12} {:>12}'.format(a, b, c, d))
    return

def modexp(b, e, m):
    x = 1
    wrapper('x', 'b', 'e', 'm')
    wrapper(x, b, e, m)
    while e > 0:
        if e % 2: x = (x * b) % m
        b = (b * b) % m
        e >>= 1
```

```
        wrapper(x, b, e, m)
    return x

def main():
    tests = [{'a': 235973, 'e': 456789, 'p': 583903},
             {'a': 984327457683, 'e': 2153489582, 'p': 994348472629}]
    for t in tests:
        a, e, p = t['a'], t['e'], t['p']
        assert modexp(a, e, p) == pow(a, e, p)
    print("Success!")
    return

if __name__ == '__main__':
    main()
```

q3.py

```
#!/usr/bin/env python3

import gensafeprime
import hashlib
import random
import binascii

random.seed(12345)              # Seed the PRNG
p = gensafeprime.generate(1024) # Generate a safe prime `p`
g = random.randint(2,p-2)       # Alice and Bob agree on a base `g`
a = random.randint(2,p-2)       # Alice selects their `a`
b = random.randint(2,p-2)       # Bob selects their `b`
ga = pow(g,a,p)                 # Alice computes g^a mod p
gb = pow(g,b,p)                 # Bob computes g^b mod p
ka = pow(gb,a,p)                # Alice uses Bob's `gb` to compute the key
kb = pow(ga,b,p)                # Bob uses Alice's `ga` to compute the key
assert ka == kb                 # Alice and Bob now have the same key!

# ka is maximally 1024 bits, so convert to 128 bytes
key = hashlib.sha224(ka.to_bytes(128, 'big')).hexdigest()
print(key)
```