

1. Answers

- (a) No, BG does not have the  $K_{C,TGS}$  to decrypt the necessary information.
- (b) If the lifetime of the ticket was too short, the client could not use the service for a meaningful amount of time. If the lifetime of the ticket was too long, the BG could get onto the clients machine and still use the same ticket with reprompting for the password.
- (c) Yes,  $K_{C,V}$  is used to authenticate both the client and server in round (6). Without the key, this would not be possible with the same configuration.

2. Answers

- (a) See Work

3. Answers

- (a) Timestamps need less rounds of communication to verify that the message is current. Nonces do not need to have synchronized clocks to work.
- (b)

$$\begin{aligned} p &= 1 - (64!/64^2(64-2)!) \\ &= 1 - (63/64) \\ &= 1/64 \end{aligned}$$

- (c) PGP has the user create a password and prompts for it every time the private key needs to be used. The private key is encrypted with the SHA hash of the password and is decrypted the same way.
- (d) No, once the SSL handshake finished, all data sent after the handshake is encrypted with the session key.
- (e) Cryptographic keys are generated from a random number created each handshake. This number prevents replay attacks from happening.

4. Answers

- (a) No, BG would calculate  $x \pmod{73}$  for each number with a zero in the one's digit place, making the domain the set:

$$\{x | 0 \leq x < 910, 10|x\} \tag{1}$$

This set has  $\approx 10^2 = 100$  unique values compared to the original 1000, increasing the amount of possible collisions. This calculation works because  $M \pmod{91}$  will always yield a result between 0 and 91. Multiplying that range by 10 makes the possible range for  $x$  the same as set 1. BG would then find an  $x$  where  $x \pmod{73}$  is equal to  $h(M)$  and find an  $M'$  that satisfies the equation:

$$M' = (x/10) \pmod{91}$$

BG would then replace  $M$  with  $M'$ . The following shows a collision for  $M = 103$ :

$$\begin{aligned} M &= 103 \\ h(M) &= (103 \pmod{91} * 10) \pmod{73} \\ &= (12 * 10) \pmod{73} \\ &= 120 \pmod{73} \\ &= 47 \end{aligned}$$

$$x = 850$$

$$\begin{aligned} h(M) &\stackrel{?}{=} x \pmod{73} = 850 \pmod{73} \\ h(M) &= 47 \\ M' &= (850/10) \pmod{91} = 85 \end{aligned}$$

# Programming Problem

## Self-critique

The program works as expected. I am curious if there are any restrictions for p, q, or h.

## Program Output

```
hoodz@DESKTOP-IOQ6GTP:/mnt/c/Users/hoodz/Desktop/Coding/CSCI-4534-Cryptography/
Homework-3$ ./main
Running all unit tests for DSA class
-----
Running Unit Test - Input A...
g: 2
publicKey: 4
-----
Sign Hash:
r: 2
s: 1
-----
Verify Real Hash
w: 1
u1: 0
u2: 2
v: 2
v == r: true
-----
Verify Fake Hash
w: 1
u1: 1
u2: 2
v: 1
v == r: false
-----
Unit Test - Input A -> Passed Test
Running Unit Test - Input B...
g: 25
publicKey: 27
-----
Sign Hash:
r: 16
s: 9
-----
Verify Real Hash
w: 18
u1: 21
u2: 12
v: 16
v == r: true
-----
Verify Fake Hash
w: 18
u1: 3
u2: 12
v: 4
v == r: false
-----
Unit Test - Input B -> Passed Test
Running Unit Test - Input C...
g: 8
```

```
publicKey: 19
-----
Sign Hash:
r: 8
s: 23
-----
Verify Real Hash
w: 37
u1: 7
u2: 46
v: 8
v == r: true
-----
Verify Fake Hash
w: 37
u1: 21
u2: 46
v: 1
v == r: false
-----
Unit Test - Input C -> Passed Test
-----
Finished running all tests
Passed Tests: 3
Failed Tests: 0
hoodz@DESKTOP-IOQ6GTP:/mnt/c/Users/hoodz/Desktop/Coding/CSCI-4534-Cryptography/Homework-3
$
```

## Human-readable code

### main.cpp

```
1 #include "DSA.hpp"
2 #include "UnitTest-DSA.hpp"
3
4 using namespace std;
5
6 int main(){
7     UnitTest_RunAll();
8     return 0;
9 }
```

### UnitTest-DSA.hpp

```
1 #ifndef UNITTEST_DSA_HPP
2 #define UNITTEST_DSA_HPP
3
4 #include <iostream>
5 #include "DSA.hpp"
6
7 using namespace std;
8
9 /// -----
10 /// @brief see if input 1 on homework works for DSA class
11 /// @return int 0 on pass, 1 on fail
12 /// -----
13 int UnitTest_InputA();
14
15 /// -----
16 /// @brief see if input 2 on homework works for DSA class
17 /// @return int 0 on pass, 1 on fail
18 /// -----
19 int UnitTest_InputB();
20
21 /// -----
22 /// @brief see if custom inputs work for DSA class
23 /// @return int 0 on pass, 1 on fail
24 /// -----
25 int UnitTest_InputC();
26
27 /// -----
28 /// @brief run all the unit tests for the DSA class
29 /// @return int 0 on pass, 1 on fail
30 /// -----
31 int UnitTest_RunAll();
32
33 #endif
```

### UnitTest-DSA.cpp

```
1 #include "UnitTest-DSA.hpp"
2
3 /// -----
4 /// @brief see if input 1 on homework works for DSA class
5 /// @return int 0 on pass, 1 on fail
6 /// -----
7 int UnitTest_InputA(){
8     try {
9         int p = 7,
10            q = 3,
11            h = 3,
12            x = 2,
13            k = 1,
14            HM1 = 3,
15            HM2 = 4;
16
17         DSA cipher(p, q, h, x, true, true);
18         cout << "-----" << endl;
19         cout << "Sign Hash:" << endl;
```

```

20     const Signature sign = cipher.signHash(HM1, k);
21
22     if (sign.s != 1 || sign.r != 2) throw "Returned signature is wrong!";
23
24     cout << "-----" << endl;
25     cout << "Verify Real Hash" << endl;
26     bool goodVerification = cipher.verifyHash(HM1, sign);
27
28     if (!goodVerification) throw "Good verification went wrong!";
29
30     cout << "-----" << endl;
31     cout << "Verify Fake Hash" << endl;
32     bool badVerification = cipher.verifyHash(HM2, sign);
33
34     if (badVerification) throw "Bad verification went wrong!";
35     cout << "-----" << endl;
36 } catch(const char* txt) {
37     cout << txt << endl; return 1;
38 } catch (...) { return 1; }
39
40 return 0;
41 }
42
43 /// -----
44 /// @brief see if input 2 on homework works for DSA class
45 /// @return int 0 on pass, 1 on fail
46 /// -----
47 int UnitTest_InputB(){
48     try {
49         int p = 47,
50             q = 23,
51             h = 5,
52             x = 7,
53             k = 13,
54             HM1 = 5,
55             HM2 = 4;
56
57         DSA cipher(p, q, h, x, true, true);
58         cout << "-----" << endl;
59         cout << "Sign Hash:" << endl;
60         const Signature sign = cipher.signHash(HM1, k);
61
62         if (sign.s != 9 || sign.r != 16) throw "Returned signature is wrong!";
63
64         cout << "-----" << endl;
65         cout << "Verify Real Hash" << endl;
66         bool goodVerification = cipher.verifyHash(HM1, sign);
67
68         if (!goodVerification) throw "Good verification went wrong!";
69
70         cout << "-----" << endl;
71         cout << "Verify Fake Hash" << endl;
72         bool badVerification = cipher.verifyHash(HM2, sign);
73
74         if (badVerification) throw "Bad verification went wrong!";
75         cout << "-----" << endl;
76     } catch(const char* txt) {
77         cout << txt << endl; return 1;
78     } catch (...) { return 1; }
79
80     return 0;
81 }
82
83 /// -----
84 /// @brief see if custom inputs work for DSA class
85 /// @return int 0 on pass, 1 on fail
86 /// -----
87 int UnitTest_InputC(){
88     try {
89         int p = 151,
90             q = 50,
91             h = 64,
92             x = 29,
93             k = 41,
94             HM1 = 11,
95             HM2 = 33;

```

```

96     DSA cipher(p, q, h, x, true, true);
97     cout << "-----" << endl;
98     cout << "Sign Hash:" << endl;
99     const Signature sign = cipher.signHash(HM1, k);
100
101     if (sign.s != 23 || sign.r != 8) throw "Returned signature is wrong!";
102
103     cout << "-----" << endl;
104     cout << "Verify Real Hash" << endl;
105     bool goodVerification = cipher.verifyHash(HM1, sign);
106
107     if (!goodVerification) throw "Good verification went wrong!";
108
109     cout << "-----" << endl;
110     cout << "Verify Fake Hash" << endl;
111     bool badVerification = cipher.verifyHash(HM2, sign);
112
113     if (badVerification) throw "Bad verification went wrong!";
114     cout << "-----" << endl;
115 } catch(const char* txt) {
116     cout << txt << endl; return 1;
117 } catch (...) { return 1; }
118
119 return 0;
120 }
121
122 /// -----
123 /// @brief run all the unit tests for the DSA class
124 /// @return int 0 on pass, 1 on fail
125 /// -----
126 int UnitTest_RunAll(){
127     int passed = 0, failed = 0;
128
129     cout << "Running all unit tests for DSA class" << endl;
130     cout << "-----" << endl;
131
132     cout << "Running Unit Test - Input A..." << endl;
133     if (UnitTest_InputA()) { cout << "Unit Test - Input A -> Failed Test \u274c" << endl; failed++; }
134     else { cout << "Unit Test - Input A -> Passed Test \u2713" << endl; passed++; }
135
136     cout << "Running Unit Test - Input B..." << endl;
137     if (UnitTest_InputB()) { cout << "Unit Test - Input B -> Failed Test \u274c" << endl; failed++; }
138     else { cout << "Unit Test - Input B -> Passed Test \u2713" << endl; passed++; }
139
140     cout << "Running Unit Test - Input C..." << endl;
141     if (UnitTest_InputC()) { cout << "Unit Test - Input C -> Failed Test \u274c" << endl; failed++; }
142     else { cout << "Unit Test - Input C -> Passed Test \u2713" << endl; passed++; }
143
144     cout << "-----" << endl;
145     cout << "Finished running all tests" << endl;
146     cout << "\u2713 Passed Tests: " << passed << endl;
147     cout << "\u274c Failed Tests: " << failed << endl;
148
149     return 0;
150 }
151 }
152

```

## DSA.hpp

```

1  #ifndef DSA_HPP
2  #define DSA_HPP
3
4  #include <iostream>
5  #include "Cryptography.hpp"
6
7  using namespace std;
8
9  // see powerpoint for naming convention
10
11 struct Signature {
12     int r;
13     int s;
14 };

```

```

15
16 class DSA {
17 private:
18     // privateKey = x, publicKey = y
19     int p, q, h, privateKey; // globally known
20     int publicKey, g;        // derived
21     bool verbose, output;    // output flags
22
23 public:
24     /// -----
25     /// @brief constructor for the DSA class, generates g & public key
26     /// @param p (int) a prime number
27     /// @param q (int) a factor of (p-1)
28     /// @param h (int) a chosen number between 0 and p-1, and fastModExponentiation(h, (p-1)/q, p) > 1
29     /// @param privateKey (int) a number between 0 and q
30     /// @param verbose (bool) a flag to show debugging information
31     /// @param output (bool) a flag to show homework outputs
32     /// -----
33     DSA(int p, int q, int h, int privateKey, bool verbose = false, bool output = false);
34
35     /// -----
36     /// @brief calculates the signature values r & s for the given hash and random number
37     /// @param hash (int) the hash of a message
38     /// @param k (int) a random number < q
39     /// @return (Signature) the r & s values for hash and k
40     /// -----
41     Signature signHash(const int hash, const int k) const;
42
43     /// -----
44     /// @brief verifies that the signature matches the given hash
45     /// @param hash (int) the hash of a message
46     /// @param sig (Signature) the r & s values for the signature
47     /// @return (bool) true if the signature is paired with hash, otherwise false
48     /// -----
49     bool verifyHash(const int hash, const Signature sig) const;
50
51     ~DSA()=default;
52 };
53
54 #endif

```

## DSA.cpp

```

1 #include "DSA.hpp"
2
3 /// -----
4 /// @brief constructor for the DSA class, generates g & public key
5 /// @param p (int) a prime number
6 /// @param q (int) a factor of (p-1)
7 /// @param h (int) a chosen number between 0 and p-1, and fastModExponentiation(h, (p-1)/q, p) > 1
8 /// @param privateKey (int) a number between 0 and q
9 /// @param verbose (bool) a flag to show debugging information
10 /// @param output (bool) a flag to show homework outputs
11 /// -----
12 DSA::
13 DSA(int p, int q, int h, int privateKey, bool verbose, bool output) :
14 p(p), q(q), h(h), privateKey(privateKey), verbose(verbose), output(output) {
15     g = fastModExponentiation(h, (p-1)/q, p); //  $h^{(p-1)/q} \bmod p$ 
16     publicKey = fastModExponentiation(g, privateKey, p); //  $g^x \bmod p$ 
17
18     if (verbose || output){
19         cout << "g:          " << g << endl;
20         cout << "publicKey:    " << publicKey << endl;
21     }
22 }
23
24 /// -----
25 /// @brief calculates the signature values r & s for the given hash and random number
26 /// @param hash (int) the hash of a message
27 /// @param k (int) a random number < q
28 /// @return (Signature) the r & s values for hash and k
29 /// -----
30 Signature DSA::
31 signHash(const int hash, const int k) const{

```

```

32     int r = fastModExponentiation(g, k, p) % q;           // (g ^ k mod p) mod q
33     int inverseK = modInverse(k, q);                     // k^-1 mod q
34     int s = (inverseK * (hash + (privateKey * r))) % q;   // [k^-1 (hash + privateKey * r)] mod q
35
36     if (verbose || output){
37         cout << "r:      " << r << endl;
38         cout << "s:      " << s << endl;
39     }
40
41     return Signature {
42         r,
43         s
44     };
45 }
46
47 /// -----
48 /// @brief verifies that the signature matches the given hash
49 /// @param hash (int) the hash of a message
50 /// @param sig (Signature) the r & s values for the signature
51 /// @return (bool) true if the signature is paired with hash, otherwise false
52 /// -----
53 bool DSA::
54 verifyHash(const int hash, const Signature sig) const{
55     int w = modInverse(sig.s, q);
56     int u1 = (hash * w) % q;
57     int u2 = (sig.r * w) % q;
58     int v = ((fastModExponentiation(g, u1, p) * fastModExponentiation(publicKey, u2, p)) % p) % q;
59
60     if (verbose || output){
61         cout << "w:      " << w << endl;
62         cout << "u1:      " << u1 << endl;
63         cout << "u2:      " << u2 << endl;
64         cout << "v:      " << v << endl;
65         cout << "v == r:   " << boolalpha << (v == sig.r) << endl;
66     }
67     return v == sig.r;
68 }

```

## Cryptography.hpp

```

1  #ifndef CRYPTOGRAPHY_HPP
2  #define CRYPTOGRAPHY_HPP
3
4  using namespace std;
5
6  /// -----
7
8  /// @brief      helper function to calculate modular inverse, ax = 1 mod n from modLinearEquationSolver
9  /// @param a      (int) positive known term in the above equation
10 /// @param n      (int) modulus value of the equation
11 /// @return      (int) modular inverse of a mod n
12 /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
13 /// -----
14
15 int modInverse(int a, int n);
16
17 /// -----
18
19 /// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
20 /// @param a      (int) positive known term in the above equation
21 /// @param b      (int) positive congruent term in the above equation
22 /// @param n      (int) modulus value of the equation
23 /// @return      (int) the solution x to the equation ax = b mod n
24 /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
25 /// -----
26
27 int modLinearEquationSolver(int a, int b, int n);
28
29 /// -----

```



```

26 /// @brief      the extended Euclidean algorithm to find the greatest common denominator
27 ///            where remainder = ax + by = GCD(a, b)
28 /// @param a      (int) the first term to find the GCD with
29 /// @param b      (int) the second term to find the GCD with
30 /// @param x      (int*) return parameter to return Bezout coefficient x
31 /// @param y      (int*) return parameter to return Bezout coefficient y
32 /// @return       (int) the GCD of a & b, as well as Bezout coefficients in x & y
33 /// @cite         https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode
34 /// @date         4 Mar 2024
35 ///
-----
36 int gcdExtended(int a, int b, int* x, int* y);
37
38 ///
-----
39 /// @brief      solve a ^ b mod n fast using less multiplications
40 /// @param a      (int) positive base of the equation
41 /// @param b      (int) positive exponent of the equation
42 /// @param n      (int) modulus value of the equation
43 /// @return       (int) result of a ^ b mod n
44 /// @note        relies on the idea that modding smaller parts gets same result
45 /// @example     a^5 mod n = a^2 mod n * a^2 mod n * a mod n
46 ///
-----
47 int fastModExponentiation(int a, int b, int n);
48
49 #endif

```

## Cryptography.cpp

```

1  #include "Cryptography.hpp"
2
3  ///
-----
4  /// @brief      helper function to calculate modular inverse, ax = 1 mod n from modLinearEquationSolver
5  /// @param a      (int) positive known term in the above equation
6  /// @param n      (int) modulus value of the equation
7  /// @return       (int) modular inverse of a mod n
8  /// @throw       if the modular inverse cannot be found, a const char* message notification is thrown
9  ///
-----
10 int
11 modInverse(int a, int n) {
12     return modLinearEquationSolver(a, 1, n);
13 }
14
15 ///
-----
16 /// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
17 /// @param a      (int) positive known term in the above equation
18 /// @param b      (int) positive congruent term in the above equation
19 /// @param n      (int) modulus value of the equation
20 /// @return       (int) the solution x to the equation ax = b mod n
21 /// @throw       if the modular inverse cannot be found, a const char* message notification is thrown
22 ///
-----
23 int
24 modLinearEquationSolver(int a, int b, int n) {
25     int x,          // first Bezout coefficient from gcdExtended
26     y;              // second Bezout coefficient from gcdExtended
27
28     int gcd = gcdExtended(a, n, &x, &y);    // get the greatest common denominator and the Bezout coefficients
29     if (b % gcd == 0){
30         return ((x % n + n) % n);          // make sure solution x is within modulus range and return it
31     }
32

```

```

33     throw "Modular Linear Equation Solver cannot return a value";
34 }
35
36 ///
-----
37 /// @brief      the extended Euclidean algorithm to find the greatest common denominator
38 ///             where remainder = ax + by = GCD(a, b)
39 /// @param a     (int) the first term to find the GCD with
40 /// @param b     (int) the second term to find the GCD with
41 /// @param x     (int*) return parameter to return Bezout coefficient x
42 /// @param y     (int*) return parameter to return Bezout coefficient y
43 /// @return      (int) the GCD of a & b, as well as Bezout coefficients in x & y
44 /// @cite        https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode
45 /// @date        4 Mar 2024
46 ///
-----
47 int
48 gcdExtended(int a, int b, int* x, int* y){
49     int remainder = a, newRemainder = b;
50     int bezoutX = 1, bezoutXCalc = 0;
51     int bezoutY = 0, bezoutYCalc = 1;
52     int quotient = 0, temp = 0;
53
54     while (newRemainder != 0){ // while there is something to divide
55         quotient = remainder / newRemainder; // get the quotient result
56         temp = newRemainder;
57         newRemainder = remainder - (quotient * newRemainder); // calculate next remainder
58         remainder = temp;
59
60         temp = bezoutXCalc;
61         bezoutXCalc = bezoutX - (quotient * bezoutXCalc); // calculate next x Bezout coefficient
62         bezoutX = temp;
63
64         temp = bezoutYCalc;
65         bezoutYCalc = bezoutY - (quotient * bezoutYCalc); // calculate next y Bezout coefficient
66         bezoutY = temp;
67     }
68
69     // cout << "Quotients of GCD: (" << bezoutYCalc << ", " << bezoutXCalc << ")" << endl;
70     // cout << "Final GCD: " << remainder << endl;
71     // cout << "Bezout coefficients: (" << bezoutX << ", " << bezoutY << ")" << endl;
72
73     *x = bezoutX; *y = bezoutY; // assign return parameters and return GCD
74     return remainder;
75 }
76
77 ///
-----
78 /// @brief      solve a ^ b mod n fast using less multiplications
79 /// @param a     (int) positive base of the equation
80 /// @param b     (int) positive exponent of the equation
81 /// @param n     (int) modulus value of the equation
82 /// @return      (int) result of a ^ b mod n
83 /// @note        relies on the idea that modding smaller parts gets same result
84 /// @example     a^5 mod n = a^2 mod n * a^2 mod n * a mod n
85 ///
-----
86 int
87 fastModExponentiation(int a, int b, int n) {
88     int result = 1;
89     a = a % n;
90
91     while (b > 0){
92         if (b & 1) result = (result * a) % n; // if b is odd, set the new result and mod it by n (result
          * [a mod n])
93         b >>= 1; // another a^2 mod n is added, divide exponent b by 2
94         a = (a * a) % n; // a = a^2 mod n
95     }
96
97     return result;
98 }

```