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 $M = 103$:

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

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

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

```
#include "DSA.hpp"
#include "UnitTest-DSA.hpp"

using namespace std;

int main(){
    UnitTest_RunAll();
    return 0;
}
```

UnitTest-DSA.hpp

```
#ifndef UNITTEST_DSA_HPP
#define UNITTEST_DSA_HPP

#include <iostream>
#include "DSA.hpp"

using namespace std;

/// ----------------------------------------------------------------------------
/// @brief see if input 1 on homework works for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------------
int UnitTest_InputA();

/// ----------------------------------------------------------------------------
/// @brief see if input 2 on homework works for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------------
int UnitTest_InputB();

/// ----------------------------------------------------------------------------
/// @brief see if custom inputs work for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------------
int UnitTest_InputC();

/// ----------------------------------------------------------------------------
/// @brief run all the unit tests for the DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------------
int UnitTest_RunAll();

#endif
```

UnitTest-DSA.cpp

```
#include "UnitTest-DSA.hpp"

/// ----------------------------------------------------------------------------
/// @brief see if input 1 on homework works for DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------------
int UnitTest_InputA(){
    try {
        int p = 7,
            q = 3,
            h = 3,
            x = 2,
            k = 1,
            HM1 = 3,
            HM2 = 4;

            DSA cipher(p, q, h, x, true, true);
            cout << "----------------------------------------------------------------------------" << endl;
            cout << "Sign Hash:" << endl;
```

```
            const Signature sign = cipher.signHash(HM1, k);

            if (sign.s != 1 || sign.r != 2) throw "Returned signature is wrong!";

            cout << "-----------------------------------------------------------------------------" << endl;
            cout << "Verify Real Hash" << endl;
            bool goodVerification = cipher.verifyHash(HM1, sign);

            if (!goodVerification) throw "Good verification went wrong!";

            cout << "-----------------------------------------------------------------------------" << endl;
            cout << "Verify Fake Hash" << endl;
            bool badVerification = cipher.verifyHash(HM2, sign);

            if (badVerification) throw "Bad verification went wrong!";
            cout << "-----------------------------------------------------------------------------" << endl;
    } catch(const char* txt) {
        cout << txt << endl; return 1;
    } catch (...) { return 1; }

    return 0;
}

/// ---------------------------------------------------------------------------
/// @brief see if input 2 on homework works for DSA class
/// @return int 0 on pass, 1 on fail
/// ---------------------------------------------------------------------------
int UnitTest_InputB(){
    try {
        int p = 47,
            q = 23,
            h = 5,
            x = 7,
            k = 13,
            HM1 = 5,
            HM2 = 4;

            DSA cipher(p, q, h, x, true, true);
            cout << "-----------------------------------------------------------------------------" << endl;
            cout << "Sign Hash:" << endl;
            const Signature sign = cipher.signHash(HM1, k);

            if (sign.s != 9 || sign.r != 16) throw "Returned signature is wrong!";

            cout << "-----------------------------------------------------------------------------" << endl;
            cout << "Verify Real Hash" << endl;
            bool goodVerification = cipher.verifyHash(HM1, sign);

            if (!goodVerification) throw "Good verification went wrong!";

            cout << "-----------------------------------------------------------------------------" << endl;
            cout << "Verify Fake Hash" << endl;
            bool badVerification = cipher.verifyHash(HM2, sign);

            if (badVerification) throw "Bad verification went wrong!";
            cout << "-----------------------------------------------------------------------------" << endl;
    } catch(const char* txt) {
        cout << txt << endl; return 1;
    } catch (...) { return 1; }

    return 0;
}

/// ---------------------------------------------------------------------------
/// @brief see if custom inputs work for DSA class
/// @return int 0 on pass, 1 on fail
/// ---------------------------------------------------------------------------
int UnitTest_InputC(){
    try {
        int p = 151,
            q = 50,
            h = 64,
            x = 29,
            k = 41,
            HM1 = 11,
            HM2 = 33;
```

```
            DSA cipher(p, q, h, x, true, true);
            cout << "----------------------------------------------------------------------------" << endl;
            cout << "Sign Hash:" << endl;
            const Signature sign = cipher.signHash(HM1, k);

            if (sign.s != 23 || sign.r != 8) throw "Returned signature is wrong!";

            cout << "----------------------------------------------------------------------------" << endl;
            cout << "Verify Real Hash" << endl;
            bool goodVerification = cipher.verifyHash(HM1, sign);

            if (!goodVerification) throw "Good verification went wrong!";

            cout << "----------------------------------------------------------------------------" << endl;
            cout << "Verify Fake Hash" << endl;
            bool badVerification = cipher.verifyHash(HM2, sign);

            if (badVerification) throw "Bad verification went wrong!";
            cout << "----------------------------------------------------------------------------" << endl;
    } catch(const char* txt) {
        cout << txt << endl; return 1;
    } catch (...) { return 1; }

    return 0;
}

/// ----------------------------------------------------------------------------
/// @brief run all the unit tests for the DSA class
/// @return int 0 on pass, 1 on fail
/// ----------------------------------------------------------------------------
int UnitTest_RunAll(){
    int passed = 0, failed = 0;

    cout << "Running all unit tests for DSA class" << endl;
    cout << "----------------------------------------------------------------------------" << endl;

    cout << "Running Unit Test - Input A..." << endl;
    if (UnitTest_InputA()) { cout << "Unit Test - Input A -> Failed Test \u274c" << endl; failed++;}
    else { cout << "Unit Test - Input A -> Passed Test \u2713" << endl; passed++; }

    cout << "Running Unit Test - Input B..." << endl;
    if (UnitTest_InputB()) { cout << "Unit Test - Input B -> Failed Test \u274c" << endl; failed++;}
    else { cout << "Unit Test - Input B -> Passed Test \u2713" << endl; passed++; }

    cout << "Running Unit Test - Input C..." << endl;
    if (UnitTest_InputC()) { cout << "Unit Test - Input C -> Failed Test \u274c" << endl; failed++;}
    else { cout << "Unit Test - Input C -> Passed Test \u2713" << endl; passed++; }


    cout << "----------------------------------------------------------------------------" << endl;
    cout << "Finished running all tests" << endl;
    cout << "\u2713 Passed Tests: " << passed << endl;
    cout << "\u274c Failed Tests: " << failed << endl;

    return 0;
}
```

DSA.hpp

```
#ifndef DSA_HPP
#define DSA_HPP

#include <iostream>
#include "Cryptography.hpp"

using namespace std;

// see powerpoint for naming convention

struct Signature {
    int r;
    int s;
};
```

```cpp

class DSA {
private:
    // privateKey = x, publicKey = y
    int p, q, h, privateKey;    // globally known
    int publicKey, g;           // derived
    bool verbose, output;       // output flags

public:
    /// ---------------------------------------------------------------------------
    /// @brief constructor for the DSA class, generates g & public key
    /// @param p (int) a prime number
    /// @param q (int) a factor of (p-1)
    /// @param h (int) a chosen number between 0 and p-1, and fastModExponentiation(h, (p-1)/q, p) > 1
    /// @param privateKey (int) a number between 0 and q
    /// @param verbose (bool) a flag to show debugging information
    /// @param output (bool) a flag to show homework outputs
    /// ---------------------------------------------------------------------------
    DSA(int p, int q, int h, int privateKey, bool verbose = false, bool output = false);

    /// ---------------------------------------------------------------------------
    /// @brief calcuates the signature values r & s for the given hash and random number
    /// @param hash (int) the hash of a message
    /// @param k (int) a random number < q
    /// @return (Signature) the r & s values for hash and k
    /// ---------------------------------------------------------------------------
    Signature signHash(const int hash, const int k) const;

    /// ---------------------------------------------------------------------------
    /// @brief verifies that the signature matches the given hash
    /// @param hash (int) the hash of a message
    /// @param sig (Signature) the r & s values for the signature
    /// @return (bool) true if the signature is paired with hash, otherwise false
    /// ---------------------------------------------------------------------------
    bool verifyHash(const int hash, const Signature sig) const;

    ~DSA()=default;
};

#endif
```

DSA.cpp

```cpp
#include "DSA.hpp"

/// ---------------------------------------------------------------------------
/// @brief constructor for the DSA class, generates g & public key
/// @param p (int) a prime number
/// @param q (int) a factor of (p-1)
/// @param h (int) a chosen number between 0 and p-1, and fastModExponentiation(h, (p-1)/q, p) > 1
/// @param privateKey (int) a number between 0 and q
/// @param verbose (bool) a flag to show debugging information
/// @param output (bool) a flag to show homework outputs
/// ---------------------------------------------------------------------------
DSA::
DSA(int p, int q, int h, int privateKey, bool verbose, bool output) :
p(p), q(q), h(h), privateKey(privateKey), verbose(verbose), output(output) {
    g = fastModExponentiation(h, (p-1)/q, p);              // h ^ ([p-1]/q) mod p
    publicKey = fastModExponentiation(g, privateKey, p);    // g ^ x mod p

    if (verbose || output){
        cout << "g:            " << g << endl;
        cout << "publicKey:    " << publicKey << endl;
    }
}

/// ---------------------------------------------------------------------------
/// @brief calcuates the signature values r & s for the given hash and random number
/// @param hash (int) the hash of a message
/// @param k (int) a random number < q
/// @return (Signature) the r & s values for hash and k
/// ---------------------------------------------------------------------------
Signature DSA::
signHash(const int hash, const int k) const{
```

```
    int r = fastModExponentiation(g, k, p) % q;             // (g ^ k mod p) mod q
    int inverseK = modInverse(k, q);                        // k^-1 mod q
    int s = (inverseK * (hash + (privateKey * r))) % q;     // [k^-1 (hash + privateKey * r)] mod q

    if (verbose || output){
        cout << "r:     " << r << endl;
        cout << "s:     " << s << endl;
    }

    return Signature {
        r,
        s
    };
}

/// ----------------------------------------------------------------------------
/// @brief verifies that the signature matches the given hash
/// @param hash (int) the hash of a message
/// @param sig (Signature) the r & s values for the signature
/// @return (bool) true if the signature is paired with hash, otherwise false
/// ----------------------------------------------------------------------------
bool DSA::
verifyHash(const int hash, const Signature sig) const{
    int w = modInverse(sig.s, q);
    int u1 = (hash * w) % q;
    int u2 = (sig.r * w) % q;
    int v = ((fastModExponentiation(g, u1, p) * fastModExponentiation(publicKey, u2, p)) % p)  % q;

    if (verbose || output){
        cout << "w:         " << w << endl;
        cout << "u1:        " << u1 << endl;
        cout << "u2:        " << u2 << endl;
        cout << "v:         " << v << endl;
        cout << "v == r:    " << boolalpha << (v == sig.r) << endl;
    }
    return v == sig.r;
}
```

Cryptography.hpp

```
#ifndef CRYPTOGRAPHY_HPP
#define CRYPTOGRAPHY_HPP

using namespace std;

///
    ----------------------------------------------------------------------------------------------

/// @brief      helper function to calculate modular inverse, ax = 1 mod n from modLinearEquationSolver
/// @param a    (int) positive known term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) modular inverse of a mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
///
    ----------------------------------------------------------------------------------------------

int modInverse(int a, int n);

///
    ----------------------------------------------------------------------------------------------

/// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
/// @param a    (int) positive known term in the above equation
/// @param b    (int) positive congruent term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) the solution x to the equation ax = b mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
///
    ----------------------------------------------------------------------------------------------

int modLinearEquationSolver(int a, int b, int n);

///
    ----------------------------------------------------------------------------------------------
```

```
/// @brief      the extended Euclidean algorithm to find the greatest common denominator
///             where remainder = ax + by = GCD(a, b)
/// @param a    (int) the first term to find the GCD with
/// @param b    (int) the second term to find the GCD with
/// @param x    (int*) return parameter to return Bezout coefficient x
/// @param y    (int*) return parameter to return Bezout coefficient y
/// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
/// @cite       https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode
/// @date       4 Mar 2024
/// -------------------------------------------------------------------------------------------------------------
int gcdExtended(int a, int b, int* x, int* y);

/// -------------------------------------------------------------------------------------------------------------
/// @brief      solve a ^ b mod n fast using less multiplications
/// @param a    (int) positive base of the equation
/// @param b    (int) positive exponent of the equation
/// @param n    (int) modulus value of the equation
/// @return     (int) result of a ^ b mod n
/// @note       relies on the idea that modding smaller parts gets same result
/// @example    a^5 mod n = a^2 mod n * a^2 mod n * a mod n
/// -------------------------------------------------------------------------------------------------------------
int fastModExponentiation(int a, int b, int n);

#endif
```

Cryptography.cpp

```
#include "Cryptography.hpp"

/// -------------------------------------------------------------------------------------------------------------
/// @brief      helper function to calculate modular inverse, ax = 1 mod n from modLinearEquationSolver
/// @param a    (int) positive known term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) modular inverse of a mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
/// -------------------------------------------------------------------------------------------------------------
int
modInverse(int a, int n) {
    return modLinearEquationSolver(a, 1, n);
}

/// -------------------------------------------------------------------------------------------------------------
/// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
/// @param a    (int) positive known term in the above equation
/// @param b    (int) positive congruent term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) the solution x to the equation ax = b mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
/// -------------------------------------------------------------------------------------------------------------
int
modLinearEquationSolver(int a, int b, int n) {
    int x,              // first Bezout coefficient from gcdExtended
        y;              // second Bezout coefficient from gcdExtended

    int gcd = gcdExtended(a, n, &x, &y);    // get the greatest common denominator and the Bezout coefficients
    if (b % gcd == 0){
        return ((x % n + n) % n);           // make sure solution x is within modulus range and return it
    }

```

```
    throw "Modular Linear Equation Solver cannot return a value";
}

/// ----------------------------------------------------------------------------------------------------------
/// @brief      the extended Euclidean algorithm to find the greatest common denominator
///             where remainder = ax + by = GCD(a, b)
/// @param a    (int) the first term to find the GCD with
/// @param b    (int) the second term to find the GCD with
/// @param x    (int*) return parameter to return Bezout coefficient x
/// @param y    (int*) return parameter to return Bezout coefficient y
/// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
/// @cite       https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode
/// @date       4 Mar 2024
/// ----------------------------------------------------------------------------------------------------------
int
gcdExtended(int a, int b, int* x, int* y){
    int remainder   = a, newRemainder   = b;
    int bezoutX     = 1, bezoutXCalc    = 0;
    int bezoutY     = 0, bezoutYCalc    = 1;
    int quotient    = 0, temp           = 0;

    while (newRemainder != 0){                                  // while there is something to divide
        quotient = remainder / newRemainder;                    // get the quotient result
        temp = newRemainder;
        newRemainder = remainder - (quotient * newRemainder);   // calculate next remainder
        remainder = temp;

        temp = bezoutXCalc;
        bezoutXCalc = bezoutX - (quotient * bezoutXCalc);       // calculate next x Bezout coefficient
        bezoutX = temp;

        temp = bezoutYCalc;
        bezoutYCalc = bezoutY - (quotient * bezoutYCalc);       // calculate next y Bezout coefficient
        bezoutY = temp;
    }

    // cout << "Quotients of GCD: (" << bezoutYCalc << ", " << bezoutXCalc << ")" << endl;
    // cout << "Final GCD: " << remainder << endl;
    // cout << "Bezout coefficients: (" bezoutX << ", " << bezoutY << ")" << endl;

    *x = bezoutX; *y = bezoutY;                                 // assign return parameters and return GCD
    return remainder;
}

/// ----------------------------------------------------------------------------------------------------------
/// @brief      solve a ^ b mod n fast using less multiplications
/// @param a    (int) positive base of the equation
/// @param b    (int) positive exponent of the equation
/// @param n    (int) modulus value of the equation
/// @return     (int) result of a ^ b mod n
/// @note       relies on the idea that modding smaller parts gets same result
/// @example    a^5 mod n = a^2 mod n * a^2 mod n * a mod n
/// ----------------------------------------------------------------------------------------------------------
int
fastModExponentiation(int a, int b, int n) {
    int result = 1;
    a = a % n;

    while (b > 0){
        if (b & 1) result = (result * a) % n;       // if b is odd, set the new result and mod it by n (result * [a mod n])
        b >>= 1;                                    // another a^2 mod n is added, divide exponent b by 2
        a = (a * a) % n;                            // a = a^2 mod n
    }

    return result;
}
```