Computer Security Assignment 1

March 13, 2014

1 Computer Security Assignment 1

1.1 Exercise 1

1.1.1 a)

We use the Python library gmpy to verify the requirements for the ElGamal algorithm.

```
In [31]: import gmpy
         p = gmpy.mpz(63689)
         g = gmpy.mpz(14569)
         k = gmpy.mpz(11636)
```

Requirements:

1. Number `p` needs to be prime. gmpy has a function `is_prime` which returns 2 when its argument is prime. Since `p` is prime, its greatest common divisor with `g` is clearly 1.

```
In [15]: 2 == gmpy.is_prime(p) && 1 == gmpy.gcd(p, g)
```

```
Out[15]: True
```

2. Number `g` needs to be a generator. Thus it must be a member of the cyclic group $(Z_p)^*$ which in this case contains all the numbers from 2 smaller than `p` because `p` is prime. Moreover, it must be able to generate all of the numbers in this cyclic group when raised to the power of each of the elements in $(Z_p)^*$ up to $p-2$. Therefore, the set generated by `g` and the set of numbers from 1 to `p` - 1 should be equal: $\{g^1, g^2, ..., g^{p-2}\} = \{1, 2, ..., p-1\}$.

```
In [36]: g in range(2, p) and set(pow(g, i, p) for i in range(1, p - 1)) == set(range(2, p))
```

```
Out[36]: True
```

3. Furthermore, the private key `k` needs to be from the set $\{1, ..., p-2\}$.

```
In [33]: k < p - 1
```

```
Out[33]: True
```

1.1.2 b)

To find the decryption of the ciphertext we use `gmpy` again. Firstly, we are given the results of encryption: c_1 and c_2. Secondly, to decrypt we use the decryption formula from the ElGamal algorithm:

$D_{EG} = c_1^{-k} \cdot c_2 \pmod{p}$:

```
In [34]: c1, c2 = 7265, 44824
         pow(c1, p - k - 1) * c2 % p
```

```
Out[34]: mpz(2014)
```

1.2 Exercise 2

1.2.1 a)

We could use a key k of length at least as long as the message. Then for each block m_i of the message we would use a corresponding block k_i of the same size as m_i from key k.

1.2.2 b)

- Raw RSA does not satisfy this property. It is because RSA is deterministic, hence whenever we encrypt a message `m` with the same public key `pk` we always get the same ciphertext `c`. This is what the attacker could check for both messages and confirm which one encrypts to the ciphertext.
- ElGamal does satisfy the property. It uses a random number `r` from each of the participants, hence the same message does not always encrypt to the same ciphertext. Furthermore, since we are using randomness, a message `m` can be encrypted to ciphertext `c` if we select the right random number `r`.

1.2.3 c)

To show that ElGamal is malleable, we take e, c_1 which were generated in encryption phase of ElGamal for message m_1. Now we can apply the function $f(x) = 2x$ to c_1 so that we get:
$c_2 = f(c_1) = 2 \cdot m \cdot (g^d)^r \pmod{p}$.
Now, we send the tuple e, c_2 and when that is decrypted the message m_2 is received instead of m_1:

$$e^{-d} \cdot c_2 \pmod{p} = (g^r)^{-d} \cdot 2 \cdot m \cdot (g^d)^r \pmod{p} = 2 \cdot m \pmod{p}$$

1.2.4 d)

Ciphertexts are rejected when the k_1 least significant bits of the decrypted message m are not equal to 0^{k_1}, where k_1 is the number of zeros that were padded to the end of message m in the encryption stage.

1.3 Exercise 3

1.3.1 a)

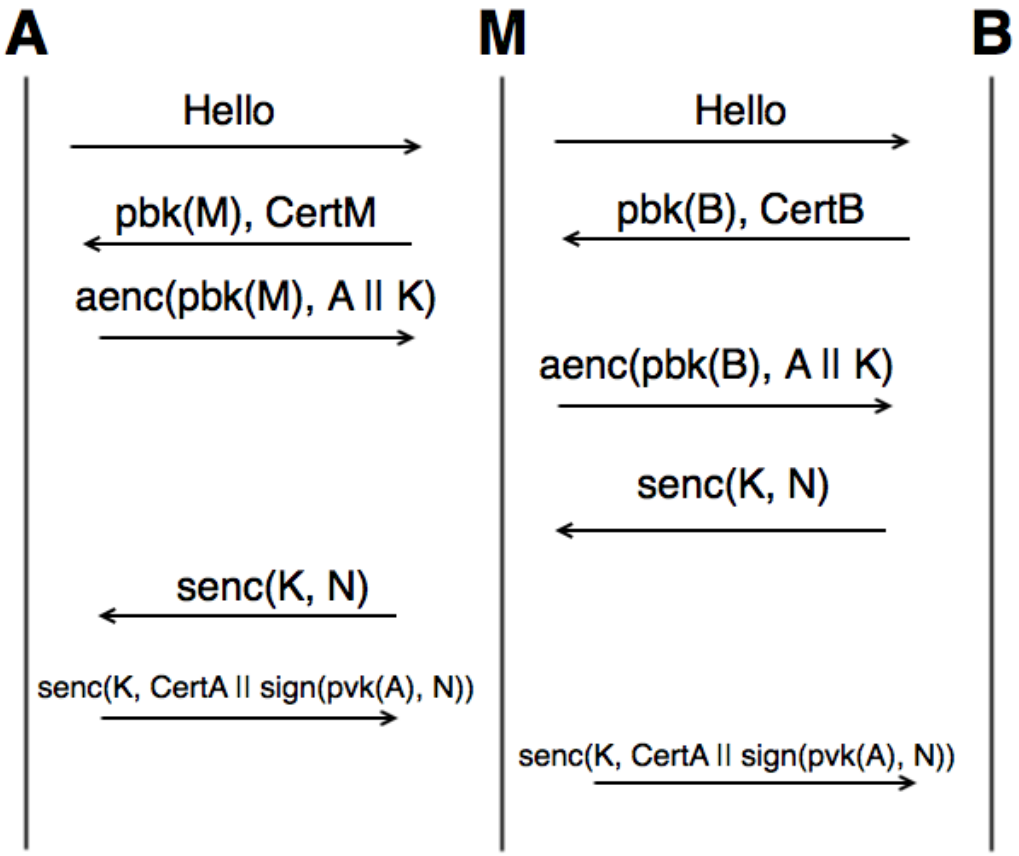
We use a variant of Lowe's attack where `M` is the malicious user who manages to convince `B` that he is `A`. This attack is only possible if `A` tries to participate in this protocol with `M`:

1. Let's say that `A` generates key `K` for communication with `M`.
2. After that `M` initiates an exchange with `B` with `A`'s identity and re-using the key `K` generated by `A` which `M` can read because it was encrypted by `M`'s public key.
3. Now, `B` generates a nonce `N` and sends it encrypted with key `K` to `M`.
4. `M` passes `N` encrypted with `K` to `A` who responds with his certificate and signed nonce `N` all encrypted with `K`.
5. Now, `M` only needs to pass this along to `B` for `B` to think that it is authenticating with `A` instead of `M`.
6. After the last step, all communication between `M` and `B` is encrypted with key `K` with `B` thinking that it is talking to `A` and `M` being able to decrypt the messages.

The attack is described on the diagram below:

```
In [3]: Image('lowe_attack.png')
```

```
Out[3]:
```



1.3.2 b)

We could fix this issue by adding in `B`'s (the receiver's) identity in the last step of the protocol so that `A` sends `B` $senc(K, Cert_A || sign(pvk(A), < N, B >))$. Thus `M` would not be able to send this message along to `B` pretending to be `A` because it would be his identity included in the signature as a receiver and not `B`'s identity.

Furthermore, `M` would not be able to change it to `B`'s identity since the signature is signed by the private key of `A`.

1.4 Exercise 4

1.4.1 a)

Properties

- Both participants get an equal say in what the resulting number should be. Thus none of them is able to force it (as required).
  - My protocol achieves this by getting a random number from both participants and XOR-ing it together to form a new number. There is no way for any of the participants to force some number because they will not know what number the other person has chosen until they commit to a number themselves.
- The participants should not be able to change the number they generated based on the other participant's number.
  - My protocol achieves this by forcing the players to commit to the hashes of their numbers. Thus the participants are not able to change their numbers after they reveal their hashes because the number would not hash to the same hash (we use a collision-resistant function for hashing). Furthermore, they are not able to deduce what number the other participant generated before they commit to their number because the hashing function is one-way.

Requirements

- The participants have agreed on a CRF hash function $H(x)$ beforehand which they can use to verify that the other participant's hash corresponds to their number.
- The participants are able to hash the number they generated with $H(x)$ and are able to verify other players' hashes.

Assumptions

- The randomly generated number does not need to be secret so the players do not mind an attacker eavesdropping.
- The phones are tamper-free and the Man in the Middle attack is not possible. Otherwise, the attacker could pretend to be the other participant to both of the participants and generate a random number that he wants.

Protocol Steps

1. Both participants choose a random number.
2. They hash this number with a CRF hash function $H(x)$.
3. Both participants reveal these hashes and commit to them.
4. Afterwards, they reveal their random numbers.
5. Both of the participants can check that the random number of the other participant hashes to the commited value using the function $H(x)$.
6. The random numbers are XOR-ed together to form a new randomly generated number.

1.4.2 b)

Properties

- The participants should not be able to change the hand-signs they want to play based on the other participant's hand-sign.
  - My protocol achieves this by forcing the players to commit to the hashes of their hand-signs padded with a random number. Thus the participants are not able to change their hand-signs or random numbers after they reveal their hashes because they would not hash to the same hash (we use a collision-resistant function for hashing). Furthermore, they are not able to deduce what

hand-sign the other participant played before they commit to their hand-sign because the hashing function is one-way and the other participant used an unknown random number to pad the value of their hand-sign before hashing.

Requirements

- The participants have agreed on a CRF hash function $H(x)$ beforehand which they can use to verify that the other participant's hash corresponds to their number.
- The participants have agreed on how they are going to perform the padding.
- The participants are able to hash the number they generated with $H(x)$ and are able to verify other players' hashes.

Assumptions

- The game is played on a secure channel so that an attacker cannot change players' hand-signs.
- The hand-signs do not need to be secret so that if an attacker is eavesdropping it has no unwanted consequences.

Protocol Steps

1. Each participant chooses what hand-sign they are going to play.
2. They add some padding to the value of the hand-sign using a randomly generated number.
3. They hash the padded value with a CRF $H(x)$ and reveal the hash - commiting to it.
4. Afterwards, they reveal their randomly generated number and their hand-sign - commiting to it.
5. After that, the players are able to verify each other's hand-sign by padding the hand-sign with the randomly generated number and hashing it using $H(x)$.
6. After that, all the hand-signs are verified and we can play the round.