IT Security - Task 1

Group G41

Name	E-Mail
LORENZ Peter	peter.lorenz@student.tugraz.at
STRAMER Jorrit	jstramer@student.tugraz.at

Theoretical Questions

1. **Explain the idea behind authenticated encryption ciphers and list their properties.**

   Authenticated encryption (AE/AEAD) ciphers combine encryption and message authentication codes, which means that a developer does not have to combine these two security mechanisms on his own, which often led to errors in the past. An AE cipher takes a plaintext as input and generates a ciphertext as well as an authentication code, which can be used by the receiver to check if the data has not been changed in transit. Authenticated encryption ciphers therefore have the following properties:

   - <u>Confidentiality:</u> Nobody except the communication partners are able the read/understand the data. Is achieved by encrypting the data.
   - <u>Authenticity:</u> It should be verifiable that the data was sent by the sender and not some other entity (attacker).
   - <u>Integrity:</u> The message cannot be modified while it is being transmitted (e.g. by a man-in-the-middle attack). Usually done using Message Authentication Codes (MACs). Sender sends the data and the generated MAC value of the data to the receiver, which also generates the MAC value of the received data and checks if the generated value matches the received value.

   Additionally, authenticated encryption with associated data (AEAD) ciphers also take additional data as input, which is only authenticated and not encrypted.

2. **Explain how cipher suites using authenticated encryption ciphers are integrated in TLS 1.2.**

   TLS 1.2 only supports authenticated encryption ciphers with additional data like Galois/Counter Mode (GCM) or Counter with CBC-MAC (CCM). The cipher that should be used is specified in the handshake process, as always, and AEAD ciphers are only chosen, if both server and client support it.

   Every packet consists of a header (type, version and length of fragment) as well as the fragment itself. In case of AEAD ciphers, the fragment consists of an explicit nonce and the encrypted data. AEAD ciphers use a secret, the nonce, the plaintext as well as the additional data as input and generate a ciphertext. The additional data is only used for authentication and not encrypted itself and consists of the sequence number, the type, the TLS version and the length of the ciphertext (even though in our TLS ascon implementation we used the length of the plaintext). Every AEAD cipher has to specify the format of the nonce and how it is build.

   For the decryption and the verification, AEAD ciphers use the secret, the nonce that was used for the encryption (build using the common part and the explicit part of the record), the additional data and the ciphertext as input.

```
struct
{
  ContentType type;
  ProtocolVersion version:
  uint16 length;
  switch (CipherSpec.cipher_type) {
    case block: GenericBlockCipher;
    case aead: GenericAEADCipher;
    case ...
  } fragment;
} TLSCiphertext;
```

```
struct
{
  opaque explicit_nonce[CipherSpec.explicit_iv_leng
  ciphered struct
  {
    opaque content[PlainText.length];
  };
} GenericAEADCipher;
```

3. **Why is it important that if a key is reused a fresh nonce is used for the encryption?**

   A nonce (number used only once) is a pseudo-random value that is used as a countermeasure against replay attacks, where an attacker gets hold of an older response and tries to use that to gain information/higher privileges by sending it again (e.g. ClientHello of TLS handshake). Obviously, if the same nonce and the same key are used, the main purpose of nonces to prevent replay attacks does not work anymore. Also if everything is encrypted with the same key and same nonce, cryptanalysis might be possible by using techniques like known plaintext attack, pattern analysis etc.

4. **More formally a PRF is defined in the following way: Let F : S × D → R be a family of functions (indexed by S) and let Γ be the set of all functions D → R. F is a pseudorandom function (family) if it is efficiently computable and it is (computationally) indistinguishable from a random f ∈ Γ. Compare this definition of a PRF with the PRF used in TLS. Does the TLS PRF fulfill the definition?**

   - Efficiently-Computable: Obviously, the PRF used in TLS has to be efficiently computable, otherwise it would not be suitable to send data over the internet, where usually a low latency is required. The most expensive parts are calculating the different HMACs, but ciphersuites in TLS 1.2 should use HMACs based on SHA-256 or stronger hash functions, which are also usually efficient to compute.

   - Indistinguishable: Using the same input twice for a pseudo-random function results in the same output. A PRF is (computationally) indistinguishable from a random function, if no algorithm exists that can determine if a values was created using a PRF or randomly chosen. Without knowing the secret, hash values of strong hash functions (e.g. SHA-256) cannot be connected to the plaintext that was used to create the hash value (or at least it is computationally unfeasible). Due to that property and because the PRF used in TLS 1.2 uses HMACs, it is therefore indistinguishable from a random function, because the output cannot be predicted from the input.

5. **Explain how you can fetch random data as initial secret when you only have the C/C++ standard library and the Linux kernel available. Why is the random number generator from the C/C++ standard library not suitable for this task?**

Unix systems have to special files that can be used as a source for random numbers, namely */dev/random* and */dev/urandom*. Both files serve as pseudo-random number generators (PRNGs) which use environmental noise for the generation. The difference between the two files is that */dev/random* will block when the entropy pool (the collected noise) is empty until more noise has been collected to generate new numbers. On the other hand, */dev/urandom* does not block and will generate new numbers using an algorithm when the pool is empty. Therefore it is theoretically vulnerable to cryptanalysis, but there is no known attack that always works, which is why */dev/urandom* is more often used for security tasks due to the efficiency.

For non-security related tasks, usually the *rand()* function is used to generate pseudo-random numbers. C++11 also introduced a separate <*random*> header with different functions to generate random numbers. This sequence is not random, because it is determined by a small set of initial values, called the seed. A PRNG only generates random numbers if also the seed has been randomly chosen. The problem with PRNGs of the C/C++ standard library is, that often the current time is used as a seed and if an attacker can guess the initialization time or even set it himself, this sequence can be predicted, which obviously is not secure.