IT Security

Group G41

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

Theoretical Questions for Task 1

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:

   - Confidentiality: Nobody except the communication partners are able the read/understand the data. Is achieved by encrypting the data.
   - Authenticity: It should be verifiable that the data was sent by the sender and not some other entity (attacker).
   - Integrity: 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 $F : S \times D \rightarrow R$ be a family of functions (indexed by S) and let Γ be the set of all functions $D \rightarrow R$. F is a pseudorandom function (family) if it is efficiently computable and it is (computationally) indistinguishable from a random $f \in \Gamma$. 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.

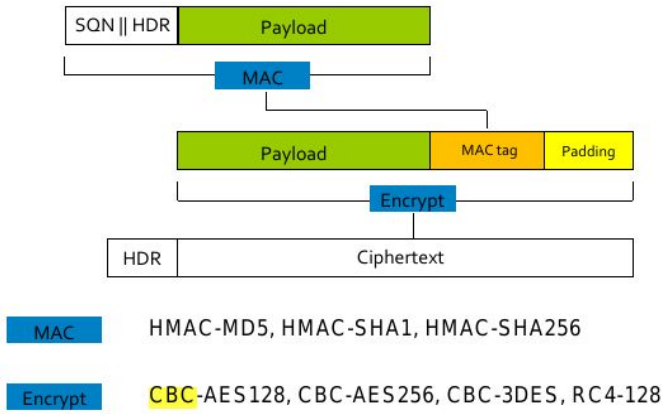
   For non-security related tasks, usually the *rand()* function is used to generate pseudo-random numbers. C++11 also introduced a separate <*random*> header with different functions to generate random numbers. This sequence is not random, because it is determined by a small set of initial values, called 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.

Theoretical Questions for Task 2

1. **What purpose fulfill AES-CBC and HMAC in TLS?**

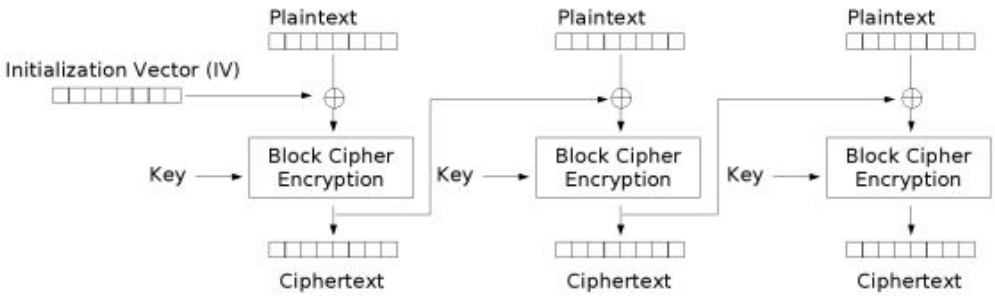
   The block cipher mode AES-CBC and the HMAC are used to encrypt TLS records (“record protocol”) and therefore provides confidentiality and authenticity of the data. The used keys are established in the handshake protocol.

   As shown in the image below, first the HMAC algorithm is used to generate a MAC of the payload and header, which is then appended to the payload together with a padding to extend everything to a multiple of the used block cipher (AES-CBS -> 16 bytes). After that everything is encrypted using AES-CBC and the resulting ciphertext together with the header is sent to the other party.



2. **Explain how ciphersuites using AES-CBC and HMAC work in principle.**

   AES-CBC:



   You have an initialization vector (IV) which you XOR with the first (128 bit) block of the plaintext. Then this block of plaintext is encrypted. The next block of plaintext is XORed with the previous encrypted block and the resulting block will again be encrypted via AES to get the next ciphertext block and so on.

HMAC:

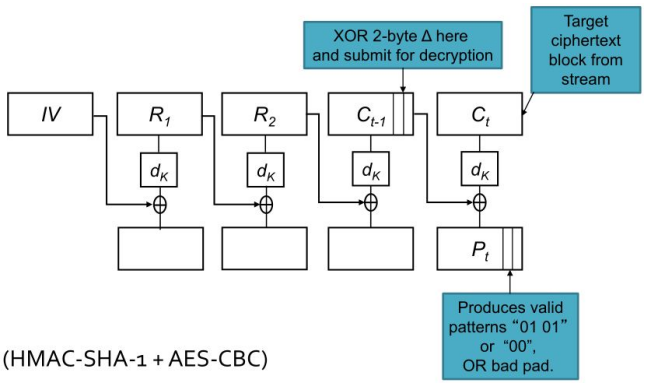
- Computed on SQN || HDR || Payload
- Adding >9 bytes of padding and iteration of hash compression function, e.g. MD5, SHA-1, SHA-256
- Running time of HMAC depends on the byte length L of SQN || HDR || Payload
    - L <= 55 bytes: 4 compression functions
    - 56 <= L <= 119: 5 compression functions
    - 120 <= L <= ?: 6 compression functions

To compute HMAC over the data we perform: H(K XOR opad, H(K XOR ipad, data))[1]:

1. Append zeros to K to create a string with B (= blocksize) bytes
2. XOR the B byte string computed in step (1) with ipad (inner padding)
3. Append the data to the B byte string resulting from step (2)
4. Apply the HMAC function H to the B byte string generated in step (3)
5. XOR the B byte string computed in step (1) with opad (outer padding)
6. Append the HMAC result from step (4) to the B byte string resulting from step (5) and HMAC everything again

3. Explain the principles of the attack.

Lucky Thirteen: The name comes from the fact that for the computation of the TLS HMAC, additionally to the data, 13 more bytes are used (5 bytes of TLS header and 8 bytes of TLS sequence number).



Given is an encrypted message. First the message is split into multiple 16 byte blocks (because AES has a blocksize of 128 bits = 16 bytes). To recover one plaintext block, two consecutive ciphertext blocks are needed. According to the paper we use 4 block of 16 bytes each (C1 || C2 || C3 || C4) for the attack, whereby C3 and C4 are the two consecutive ciphertext blocks and C1 and C2 are irrelevant (we used all 0's) and the attack tries to recover P4.

As shown in the picture above, C3 is XORed with the decrypted C4 block, which results in the plaintext block P4. Therefore if an attacker changes C3, P4 will also be changed accordingly.

[1] Copied from https://www.ietf.org/rfc/rfc2104.txt

To recover the last two bytes, an attacker tries to guess two deltas, which are XORed with the last two bytes of C3, such that the last two bytes of P4 are 0x01 || 0x01 aka. a valid padding. This is done by bruteforcing the deltas and measuring the time it took for the decryption of the blocks (using a decryption oracle). If a valid padding is used, the decryption is faster than otherwise (or rather the checking of the MAC). The bytes of the plaintext can then be calculated by simply XORing the padding (0x01) with the found delta.

To recover the following bytes, first the attacker has to change all previous deltas to again generate a valid padding (just flip a few bits or XOR the original plaintext bytes with the needed padding value, e.g. 0x02). Then the attacker tries again to find a delta which results in a valid padding by measuring the time.

Due to noise (network etc.) the attacker has to try every possible delta a few times to be able to extract the correct delta. According to the paper 2^7 times per delta should be enough, which leads to an overall complexity of $2^7 * 2^{16}$ (last 2 bytes) + $2^7 * 2^8 * 14$ (remaining 14 bytes).

4. **Why is it important to perform all checks during the decryption in constant time? If constant-time decryption was not possible, would random delays prevent the attack?**

   We assume that constant time means that the decryption of every possible value takes the same time to process, independent of the used padding etc. If TLS-CBC would need constant time for every decryption, a side-channel timing attack such as Lucky13 would not be possible, because every timing would be the same and no cases could be distinguished.
   According to the paper, adding random delays does not prevent the attack, it only lowers the success rate to 90% and more samples per delta are needed.

5. **While using the same ciphersuite, is it possible to remove the timing side-channel?**

   Yes, according to the paper[2] and the lecture it is difficult and rather complex to get constant timings, but not impossible. The paper also recommends to use a dedicated authenticated encryption algorithm such as AES-GCM instead of the CBC mode.

6. **Compare the use of the block cipher construction from this Task to the use of Authenticated Encryption in Task 1. Is it possible to model the Encrypt-Then-MAC paradigm as Authenticated Encryption scheme?**

   Yes, Encrypt-Then-MAC (EtM) is one approach for authenticated encryption schemes and is e.g. used in Internet Protocol Security (IPsec).

[2] http://www.isg.rhul.ac.uk/tls/TLStiming.pdf

7. **Explain the difference between Encrypt-then-MAC and MAC-then-Encrypt. Would MAC-then-Encrypt prevent the attack?**

   Encrypt-then-MAC (EtM) = Encrypt the plaintext, calculate MAC based on ciphertext and send both together. Properties:
   - Provides integrity of ciphertext, as long as MAC shared secret has not been compromised (Ciphertext -> MAC).
   - Provides integrity of plaintext (Plaintext -> Ciphertext -> MAC).
   - The MAC does not provide any information on the plaintext.

   MAC-then-Encrypt (MtE) = Generate MAC based on plaintext, append MAC to the plaintext and encrypt it all. Ciphertext and MAC are again sent together.
   - Does not provide integrity of ciphertext (unknown until decrypted)
   - Provides integrity of plaintext (Plaintext -> MAC).
   - The MAC does not provide any information on the plaintext (since it is encrypted).

   No, MtE does not prevent the attack, because MAC-then-Encrypt (or MAC-Encode-Encrypt) is already used in TLS and is also the same method used for the Lucky13 attack in this exercise.

8. **POSIX.1-2008 defines multiple clocks which measure different notions of "time". It specifies system wide clocks to measure real time, per-process or per-thread CPU-time clocks, and others. Compare the clocks specified in POSIX.1-2008 and compare to the timing information retrieved using the timestamp counter. Can the timestamp counter be replaced with any of the standardized clocks?**

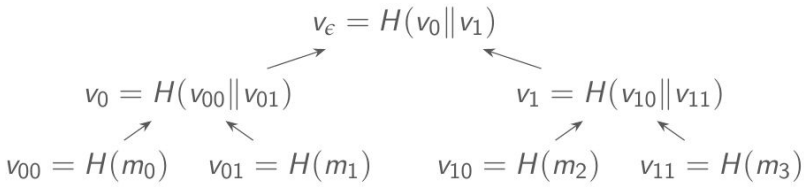
Clock	Description
CLOCK_MONOTONIC	Time since some unspecified starting point (e.g. system boot).
CLOCK_PROCESS_CPUTIME_ID	Per-process clock that uses the TimeStampCounter (TSC) register. Clock is only increased if process is active.
CLOCK_REALTIME	System-wide real-time (wall-time) clock.
CLOCK_THREAD_CPUTIME_ID	Similar to the per-process clock, but every thread has a unique counter. Clock is only increased if thread is active.

   The per-process and the per-thread clock are not suitable to replace the timestamp counter, because if e.g. the decryption oracle is in another process or thread, the current process/thread will be blocked and the counter will not be increased. Either way to replace the timestamp counter for our attack, a clock with a high enough precision is needed (at least microseconds, but nanoseconds are better).

Theoretical Questions for Task 3

1. **Explain the use of Merkle trees in blockchain.**

   Each block in the block chain consists of 1 reward transaction, multiple other transactions, a hash pointer to the previous block, the seed to solve the hash puzzle and the root hash of the merkle tree. The merkle tree is a binary hash tree where the leaves are the hashes of the transactions (including reward transaction). Each parent node is calculated by hashing the two children, i.e. H(left child || right child). The root of the tree is called the root hash.



   Therefore, the merkle tree or the root hash in the blockchain is used to check the integrity of the block that should be added.

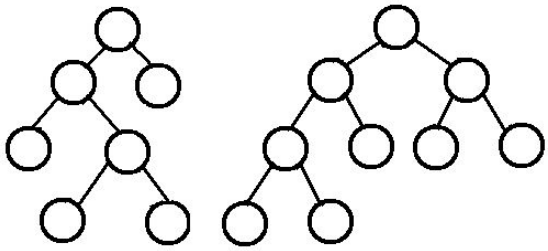
2. **Merkle trees and the blockchain both use SHA-256 in KUcoin. Which properties of the hash functions are required to protect the Merkle Tree and the blockchain respectively from tampering?**

   Properties of hash functions:
   - *Pre-image resistance:* Given the hash value it should be infeasible to find the corresponding data that was used to create that value.
   - *Second pre-image resistance:* Given one input it should be infeasible to find a second input with the same hash value.
   - Collision resistance: It should be infeasible to find two different inputs with the same hash values.

   Pre-image resistance is not really needed to prevent someone from tampering, because just knowing the input is not enough, but the other two properties are required.

3. **We define the Merkle tree in terms of a full binary tree. Assume we would now define it using a complete binary tree. Would this change the affect any properties of the Merkle tree?**

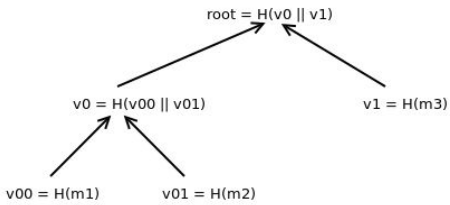
   - A full binary tree is a tree in which every node other than the leaves has either zero or two children (in our case we even have a perfect binary tree, i.e. every node has exactly two children except the leaves).
   - A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

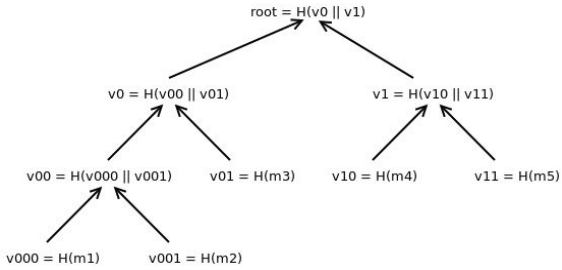


full tree complete tree

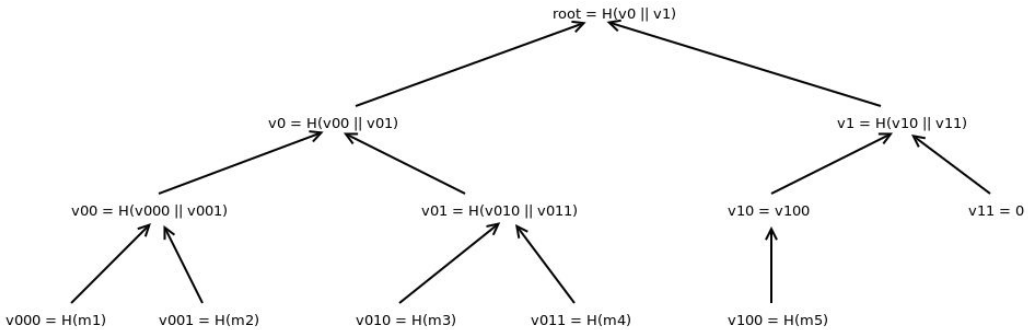
Two possibilities how the change could be meant:

- All leaves are hashes of the transactions, in which case it would actually change nothing except the structure. Examples:





- Only the last level contains the leaves/hashes of the transactions, in which case some nodes do not have any leaves and one has to define how the hash is calculated if children are missing (e.g. if node has only 1 child, copy the hash, if node has no childs, it has a hash of 0). Example:



4. Research an alternative to randomly sampling ephemeral keys in ECDSA.

Every signature generated with the ECDSA algorithms requires some random or unpredictable as input, otherwise an attacker could figure out the private key.

RFC 6979 introduced a deterministic method to the random sampling of ephemeral keys. It uses the HMAC_DRBG pseudorandom number generator. For every signature, an instance of HMAC-DRBG is generated. The private key is used as the source of entropy and the hash of the message as the nonce. The key is generated from the output of this HMAC-DRBG instance. This makes the key deterministic, given the message and the private key, but still impossible for an attacker to guess.

Allows easy verifications of ECDSA implementations using fixed test vectors, while regular implementations cannot use test vectors, because the signatures are random.

5. **In the hash puzzle the hash function is applied twice. Would applying it only once make a difference?**

   It should not make a difference because one property of a hash function is that the same input always results in the same output. Therefore obviously if the first hash value is already the same (e.g. on collision), the second will also be identical.

6. **Is there a better strategy to solve the hash puzzle than guessing seeds?**

   No, otherwise it would be easier and faster to generate KUcoins.