# IT Security: Practicals

**Implementing and Attacking Cryptographic Primitives and Protocols**

Sebastian Ramacher,
Philipp Harb, Martin Mandl, Samuel Sprung, Stefan Steinegger

October 11, 2016

## Contents

# 1 Rules

You must perform all tasks in groups of two members.[1] We will provide a Git repository to coordinate both your teamwork and the submission. Using Git is mandatory (see Section 1.1 for details). At the end of the term mandatory group interviews (Abgabegespräche) conclude the exercise. Within the interviews you

- have to be able to explain the theory behind the tasks.
- have to be able to explain your practical work. (Every group member has to know the complete work of the group.)
- have to explain if your Git statistics look abnormal, e.g. only one of the team members committed regularly.
- have the opportunity to inspect and discuss your submission (Einsichtnahme).

The grading is done as follows: $\geq 50\% : 4, \geq 62.5\% : 3, \geq 75\% : 2, \geq 87.5\% : 1$. The percentage is calculated from the sum of the received points divided by all possible points multiplied by a factor depending on the performance at the group interviews. You need to pass the group interview to get a positive grade overall. You will get a grade as soon as you submit something to any of the tasks.

In case you are unable to acquire enough points to pass, you will have the chance to extend the submission to the task where you have submitted a solution but received the least points.

You are *not* allowed to use external code other than provided by the framework. If plagiarism is detected, all involved teams will fail the course.

The schedule (task presentation, submission deadlines) can be found in Table 1.

| **Date & Time** | **Remark** |
|---|---|
| 2016.10.03 16:00 | Registration. Task 0. |
| 2016.10.10 23:59 | Registration. Task 0 deadline. |
| 2016.10.10 16:00 | Presentation of Task 1. |
| 2016.10.31 23:59 | Task 1 submission deadline. |
| 2016.11.07 16:00 | Presentation of Task 2. |
| 2016.12.05 23:59 | Task 2 submission deadline. |
| 2016.12.12 16:00 | Presentation of Task 3. |
| 2017.01.16 23:59 | Task 3 submission deadline. |
| 2017.01.23–2017.01.27 | Group interviews |

Table 1: Course schedule.

All tasks ask you to answer questions for an assignment document. The goal of the assignment documents is to reinforce the knowledge on the particular topic. By properly

[1] If you are unable to find a partner, we will assign one to you.

answering the question, you will already have reached the educational objective. Therefore, we will not check the answers for correctness. We will only check if the assignment was done.

The questions from the assignment document will be part of the group interviews.

If the assignment document is missing for a task, you will receive no points for this particular task.

All submission deadlines are hard. Submissions after the deadline are not considered.

If you have any questions, consult one of the following sources for further information:

- Newsgroup: tu-graz.lv.it-sicherheit
- Via Email: itsec-team@iaik.tugraz.at
- (If needed) question times: about one week before the submission deadlines.

## 1.1 Task Submission

If you are not familiar with Git, please have a look at the vast amount of resources about Git online. A book called Pro Git is freely available online.

All tasks are submitted using your group's Git repository and need to be tagged. Each proper submission must be tagged with a Git *tag* called

- `submission-1` for Task 1.
- `submission-2` for Task 2.
- `submission-3` for Task 3.

To correctly tag the currently checked out commit and push the new tag to the server run:

```
% git tag submission-X
% git push origin master submission-X
```

You can easily verify your submission by cloning a fresh copy of your repository into a new directory and trying to check out the corresponding tag. The Git commands used by us to clone your submissions are semantically equivalent to:

```
% git clone git@teaching.student.iaik.tugraz.at:its16g0XX.git
% cd its16g0XX
% git checkout submission-X
```

☞

You might accidentally tag the wrong commit for submission. To re-submit another commit simply create a new tag named `submission-X-Y` where `Y` is an increasing number. Assuming that you want re-submit for the first time, run the following commands:

```
% git tag submission-X-1
% git push origin master submission-X-1
```

To re-submit again use `submission-X-2`, `submission-X-3` and so on.

The tags that qualify for submission, i.e. were created before the deadline, and reference a commit before the deadline, are sorted according to their name, and we will consider the tag that sorts last as your submission.

We will checkout your submission shortly after the deadline has passed, and we will publish a newsgroup posting in tu-graz.lv.it-sicherheitlisting the received Git commit IDs that are considered to be your final submissions for the tasks.

⚠

Any discrepancies between the commit IDs published by us in the newsgroup and the actual tags in your repositories will be resolved in favor of the published commit IDs - Attempts to (re-)tag your submissions after the deadline will be penalized.

⚠

Never force push to the repository.

The directory structure of your repository should look similar to:

```
assignment-document/ - your assignment documents for all tasks
tls/                 - framework and your solutions for Task 1-2
blockchain/          - framework and your solutions for Task 3
```

All your submissions have to include

- the full source code, and
- result files where applicable.

⚠

Keep your repository clean from build artifacts like object files or executables that were generated during the build process. The same applies for generated doxygen documentation and other by-products of the build process.

## 1.2 First Steps in the Git Repository

In your Git repository, perform the following steps to get started:

1. Update author information:

```
% git config --global user.name "John Doe"
% git config --global user.email "jd@student.tugraz.at"
```

2. Ignore build artifacts:

```
% echo build/ >> .gitignore
% git add .gitignore
% git commit -m "Ignore build artifacts"
```

3. From `https://seafile.iaik.tugraz.at/d/446befa18a/` download all patches. Then apply them with `git am`:

```
% git am 0001-Import-initial-framework.patch
```

4. In the top-most `CMakeLists.txt` change the value of `GROUP_NUMBER` according to your groups name from STicS[2].

# 2 Task 0: Registration

⚠

**HARD SUBMISSION DEADLINE: 2016.10.10 23:59**

In order to fully sign up for this course, each group has to register in STicS. The first member registers a group and invites the second member to the group. Make sure to select KU as topic.

[2]Please use a leading zero if your group number is below 10.

# 3 Task 1: Implementing parts of TLS

In this task we will look at different aspects of a TLS implementation. We will primarily focus on the symmetric primitives that are involved in the data transmission step of the protocol. In particular, we will integrate the authenticated encryption cipher Ascon and look at TLS pseudorandom function family used to derive different secrets during the protocol.

## 3.1 Ascon

Ascon is an Authenticated Encryption with Associated Data (AEAD) or Authenticated Encryption (AE) cipher. AEAD schemes are designed in a way such that they can provide integrity and authenticity in addition to confidentiality. AEAD schemes add the ability to check the integrity and authenticity of the provided Associated Data which is not encrypted. Ascon [2] is one of fifteen candidates for the third round of the ongoing CAESAR competition.

Other AEAD schemes include AES in Galois/Counter Mode (AES-GCM) and AES in Counter and CBC MAC Mode (AES-CCM), and of course the other candidates of the CAESAR competition.

The reference implementations of the submissions to the competition use the following interface:

```
int crypto_aead_encrypt(
  unsigned char *c, unsigned long long *clen,
  const unsigned char *m, unsigned long long mlen,
  const unsigned char *ad, unsigned long long adlen,
  const unsigned char *nsec,
  const unsigned char *npub,
  const unsigned char *k);

int crypto_aead_decrypt(
  unsigned char *m, unsigned long long *mlen,
  unsigned char *nsec,
  const unsigned char *c, unsigned long long clen,
  const unsigned char *ad, unsigned long long adlen,
  const unsigned char *npub,
  const unsigned char *k);
```

In our case, `nsec` is unused, `npub` is the nonce, `k` the secret key, and `ad` the associated data.

## 3.2 HMAC

An HMAC is a keyed message authentication code built from a cryptographic hash function. It allows to verify data integrity and authentication of a message simultaneously [3].

Given a cryptographic hash function $h$, the HMAC $H$ is defined as:

$$H(K, m) = h\left((K \oplus O) \| h\left((K \oplus I) \| m\right)\right)$$

where $m$ is the message, $K$ the secret key padded to the right with extra zeros to the block size of $H$, or the hash of the original key if it is longer than the block size. $O$ and $I$ are the outer and inner padding which are defined to be $b$ copies of `0x5c` respectively `0x36`.

## 3.3 TLS pseudorandom function family

In multiple parts of the TLS, random data is generated from specific values. TLS defines a pseudorandom function family (PRF) [1, Section 5] for this purpose. It is defined in the following way: let $h$ be a cryptographically secure hash function and $H_h$ an HMAC based on $h$. First let $p_h$ be defined as:

$$p_h(s, d) = H_h(s, A_1 \| d) \| H_h(s, A_2 \| d) \| H_h(s, A_3 \| d) \| ...$$

where $A_i$ is defined recursively as

$$A_0 = d, A_i = H_h(s, A_{i-1}).$$

$H_h$ in $p_h$ can be iterated as many times as necessary to produce the required quantity of data. The PRF $P$ is then defined as:

$$P(s, l, d) = p_h(s, l \| d)$$

## 3.4 Generating initial secrets

The TLS PRF still takes a random secret as input. Hence a TLS implementation requires some way to gather enough randomness to produce these master secrets. The kernels of all major operating systems provide APIs respectively devices to fetch random data from pools which are filled from different entropy sources.



This subtask is primarliy meant as research task.

## 3.5 Partially implicit nonces

As Ascon (and other AE ciphers) require nonces for every encryption, TLS specifies partially implicit nonces that can be used to generate a new nonce for every encryption [4, Section 3]. The idea to construct the nonce as a fixed random part and a counter, which is incremented for every encryption. Hence, when viewed as an integer value in network byte order, the nonces look like a counter with fixed high bits. This construction ensures that as long as the counter does not overflow, every nonce is unique. Therefore, the size of the counter part determines the number of encryptions that can be performed using the same fixed part.

```
struct nonce
{
  unsigned char fixed[fixed_size];
  unsigned char counter[counter_size];
};
```

Instead of including the full nonce in every transmitted fragment, it is possible to split the fixed part in an implicit and an explicit part. The implicit part is derived from shared information known to both participants and the explicit part (including the counter) is transmitted.

## 3.6 TLS Application Data

The TLS record layer used to send application data is simple: every packet consists of header which includes the type, a version, the length of the data fragment and the data fragment itself[3][4]:

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

The content type is a single byte and the version consists of two bytes (the major and the minor protocol version). It is possible for `fragment` to be different kinds of fragments. The type of the fragment is fully determined by the cipher suite. We will only deal with Ascon and AES-CBC-HMAC based ciphersuites, so `fragment` will always be a `GenericAEADCipher` or `GenericBlockCipher`[5] in our case.

A `GenericAEADCipher` looks like the following[6]:

```
struct
{
  opaque explicit_nonce[CipherSpec.explicit_iv_length];
  ciphered struct
  {
```

[3] Note that whenever looking at the packet data, consider that integers are always in network byte-order, which is big endian.

[4] We follow the language from RFC 5246 [1]. The code is not valid C.

[5] AES-CBC-HMAC and `GenericBlockCipher` will be used in TLS part 2.

[6] `opaque` is the type used for single-byte entitities containing uninterpreted data.

```
    opaque content[PlainText.length];
  };
} GenericAEADCipher;
```

`explicit_nonce` contains the explicit part of the nonce as specified in Section 3.5. The remaing part is the encryption with an AEAD cipher using the given nonce. The associated data consists of the sequence number (`uint64_t`), the content type (`uint8_t`), the version (`uint16_t`) of the record, and the length of the plaintext (`uint16_t`).

A `GenericBlockCipher` looks like the following:

```
struct
{
  opaque IV[CipherSpec.block_length];
  block-ciphered struct
  {
    opaque content[PlainText.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
  };
} GenericBlockCipher;
```

So in short it consists of an IV of appropriate length, the plaintext, the MAC and a padding to make sure that the size is a multiple of the block cipher's block size. This structure (excluding the IV) is then encrypted using the selected block cipher.

The MAC is computed from multiple parts of the message: the sequence number (`uint64_t`), the content type (`uint8_t`), and version (`uint16_t`) of the record, the length (`uint16_t`) of the plaintext and the plaintext itself.

The padding length is chosen between 0 and 255 such that the size is a multiple of the block size. The padding length is also copied to each byte of the padding. For example, if the block length is 8 bytes, the content length is 61 bytes, and the MAC length is 20 bytes, then the length before padding is 82 bytes. Thus, the padding length modulo 8 must be equal to 6 in order to make the total length an even multiple of 8 bytes (the block length). The padding length can be 6, 14, 22, and so on, through 254. If the padding length were the minimum necessary, 6, the padding would be 6 bytes, each containing the value 6. Thus, the last 8 bytes of the `GenericBlockCipher` before block encryption would be `xx 06 06 06 06 06 06 06`, where `xx` is the last byte of the MAC.

## 3.7 Framework

The framework is written in C++11 and comes with a check suite for your submissions. The framework makes use of the CMake build system. For details on the use of the framework please check `FRAMEWORK-README.txt`. For a quick start, run

```
% ./run_cmake.sh build Debug
```

and check your submissions by running

```
% make check
```

in the `build` folder.

## 3.8 Your Tasks

⚠

**HARD SUBMISSION DEADLINE: 2016.10.31 23:59**

Your tasks for this part of the practicals are:

1. Answer the following questions for the assignment document:
   a) Explain the idea behind authenticated encryption ciphers and list their properties.
   b) Explain how cipher suites using authenticated encryption ciphers are integrated in TLS 1.2.
   c) Why is it important that if a key is re-used a fresh nonce is used for the encryption?
   d) More formally a PRF is defined in the following way: Let $F : \mathcal{S} \times D \to R$ be a family of functions (indexed by $\mathcal{S}$) and let $\Gamma$ be the set of all functions $D \to R$. $F$ 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?
   e) 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?
2. *(2 points)* In `tls/hmac-sha256.cpp` implement the class `hmac_sha256`, computing HMACs based on SHA256.
3. *(4 points)* In `tls/prf.cpp` implement the TLS PRF using HMAC-SHA256 according to [1, Section 5].
4. *(2 points)* In `tls/random.cpp` implement fetching of the initial secrets.
5. *(2 points)* In `tls/ascon128.cpp` implement the class `ascon128` performing encryption and decrpytion using Ascon using the reference implementation.
6. *(4 points)* In `tls/counter.cpp` implement the partially implicit nonce counter.
7. *(6 points)* In `tls/tls-ascon.cpp` implement record layer encryption and decryption using an Ascon based ciphersuite.

# References

[1] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465.

[2] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1.2.

[3] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997. Updated by RFC 6151.

[4] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116 (Proposed Standard), Jan. 2008.