# Challenge Sheet 2 (10 Points)

**Date due: December 5th 2022**

## Submission Instructions

Solutions to this challenge sheet must be submitted by **December 5th 2022** *before 11:59 am* to https://cysec1.de/. Please upload **only** the files *diffiehellman.py*, *md5.py* and *hle.py* to the corresponding submission fields! **Make sure that your solution contains your name and matriculation number at the top of the files!** It is not allowed to submit these solutions in a group, as you do with the regular exercise sheet solutions. In the .py files, you may add as many additional methods as you want, but please do not rename the existing methods or add code outside of methods. How many points you get for the tasks is determined by tests on our system.

### 1 Diffie-Hellman Key Exchange

This week's lecture taught us about the Diffie-Hellman Key Exchange. Now we want to implement such a key exchange using Python. For this purpose, we modeled our two entities Alice and Bob as classes.

(4 points) (a) In **diffiehellman.py** please implement all TODOs such that **exchangeKey(...)** simulates a key exchange and returns the key computed by Alice and the key computed by Bob as a tuple.

### 2 Hash Length Extension Attacks

Now you're getting hungry. Unfortunately, the best pizzeria in Saarland does not deliver to the campus. Good thing we're skilled hackers, so let's take a look at their infrastructure. Maybe there's a way around that.
When an order is placed, the customer sends a Get request to the pizzeria's server, which contains the parameters userid, name of the pizza they ordered, and the location they ordered to. It looks something like this:

```
de/order?userid=12117379&pizza=4Cheese&lat=49.31639559634483&long=6.877010186218736
```

We have already noticed that there is a problem with duplicate parameters. If one of the parameters occurs more than once, the server always takes the last value specified. For a request like:

```
at=49.31639559634483&long=6.8770101862187 36&long=10.76514&long=7.812123
```

The server would send the pizza to the location with the longitude 7.812123. With this knowledge, our plan is simple. Interrupt an order, relocate it to the front of the Cispa building, enjoy free pizza.

There is only one problem left. For each request, a checksum is sent along with it in the form of a Md5 hash. This means that the checksum will no longer match after our location is attached to the request. We are also unable to compute our own new checksum because a shared secret (key) is prepended to the request before hashing. We only know that this key has a length of 32 bytes, so we don't have to guess the length anymore. Fortunately for us, Md5 is vulnerable to so-called hash length extension attacks (attacking naive MACs). This is where you come into play.

(2 points) (a) You may already have heard a lot about hashes. But like many of us, you probably only know the theory behind it and have no idea how such a hash actually works. Now we want to change that. For this we have created an example **md5.py** implementation. This program is not complete and some bits and pieces are still missing (indicated by TODOs). Your task is to research how Md5 works on the Internet and insert the missing pieces of code. Don't worry, you don't need to understand every single line of code used, but it might be helpful to take a closer look at how the padding works in our implementation.

(2 points) (b) Next, let's prepare our attack. In the lecture, you learned that in order to perform a hash length extension attack, we need to be able to calculate the correct padding for a hash. During the last part you may have seen that the padding of Md5 contains the length of the message. This leads to a problem because during the attack we have the hash of the original request, and we need to append our new part of the request as well as a proper padding that takes into account what we already have and what we add. For this purpose we need you to implement **addPadding(...)** in **hle.py**. The length of the key (<*secret*>) is accessible via *self.KEYLENGTH* and the parameter *length* specifies the length of the original message/request + the length of the padding already present. The *message*/*message_copy* parameter is equivalent to *attacker msg* from the lecture and is the part we want to attach the padding to. This gives you the three lengths you need to successfully calculate the padding.

(2 points) (c) **This task can only be solved if you have already completed a) and b)!**
Now it is time to create the actual attack in **attack(...)** of **hle.py**. Currently, this function only forwards the customer's request to the pizzeria's server. Your task is to manipulate this request, append the content of *data_to_add*, calculate the correct padding and compute the correct hash. To do this, you can use *loadState(...)* and *runHashAlgo(...)* of *md5.py* to use the current hash as the base for hashing the added part (already padded) and get the new hash. You can run *hletest.py* to test your implementation.
*Note: Don't forget that the length parameter of addPadding should also account for the length of the existing padding. Maybe addPadding with length = 0 is useful for something :D*