Data Protection

Lab work 2: Hash functions and MAC forgery attacks

Students

Alejandro Capella del Solar

Cristian Fernández Jiménez

Professor

Jorge Luis Villar Santos

November 13, 2022

Contents

List of Figures

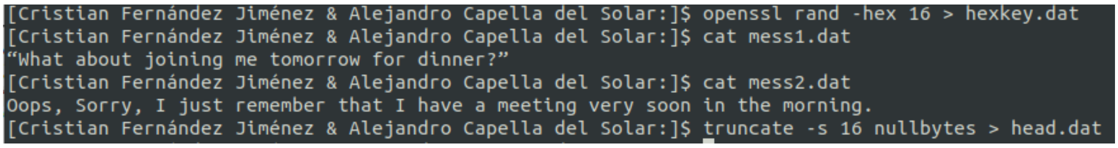
Chapter 1

Recreation of known MAC forgery attacks

1.1 CBC-MAC concatenation attack

CBC-MAC is insecure if used with variable length messages, we will try then to perform a MAC forgery attack against it. We have recreated the attack following these steps:

1. Create a random AES-128 key (random string of 16 bytes), and choose two arbitrary messages. We used the ones suggested and saved them in two files. Also we generated a random key and a 16 bytes null header, in order to append it to the messages:



```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ openssl rand -hex 16 > hexkey.dat
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat mess1.dat
“What about joining me tomorrow for dinner?”
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat mess2.dat
Oops, Sorry, I just remember that I have a meeting very soon in the morning.
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ truncate -s 16 nullbytes > head.dat
```

Figure 1.1: Creation of random key, message files and 16 bytes null header

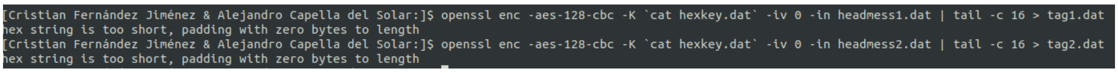
2. For simplicity, we assumed that the system adds a header to the messages consisting of 16 zero bytes, in order to create the tag of each message. Then, previously to CBC-MAC generation, was neccessary to create two files with both header and message.

   We sticked both messages with the header with the simple following commands:

```
    cat head.dat mess1.dat > headmess1.dat
    cat head.dat mess2.dat > headmess2.dat
```

Listing 1.1: concatenating messages with headers

3. Next up, generating the corresponding AES-128-CBC-MACs for the two messages with headers and store them in the files tag1.dat and tag2.dat.



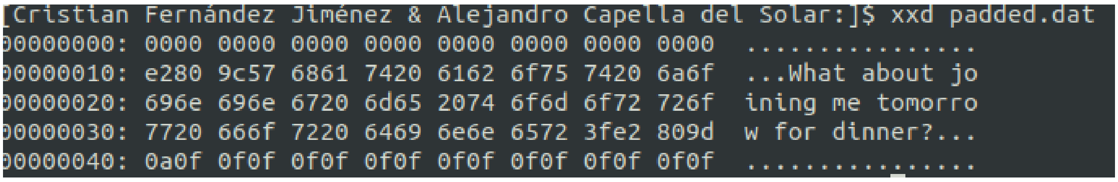
```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ openssl enc -aes-128-cbc -K `cat hexkey.dat` -iv 0 -in headmess1.dat | tail -c 16 > tag1.dat
hex string is too short, padding with zero bytes to length
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ openssl enc -aes-128-cbc -K `cat hexkey.dat` -iv 0 -in headmess2.dat | tail -c 16 > tag2.dat
hex string is too short, padding with zero bytes to length
```

Figure 1.2: Tag generation for the two messages (header included)

4. In order to investigate the padding that AES-128-CBC introduces in the last incomplete block, we encrypted a message with AES-128-CBC, and then decrypted the result with the option -nopad, recovering the padded version of the first message (that will be used to generate the forgery). The encryption/decryption lines to investigate the padding would be something like

```
openssl enc -aes-128-cbc -K $key -iv 0 -in message.dat -out
    cipher.dat

openssl enc -d -aes-128-cbc -K $key -iv 0 -nopad -in cipher.dat
    -out padded.dat xxd padded.dat
```

If we inspect the file now with *xxd*, we can check how it behaves:



```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd padded.dat
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000020: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000030: 7720 666f 7220 6469 6e6e 6572 3fe2 809d  w for dinner?...
00000040: 0a0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f  ................
```

Figure 1.3: Binary representation of message 1

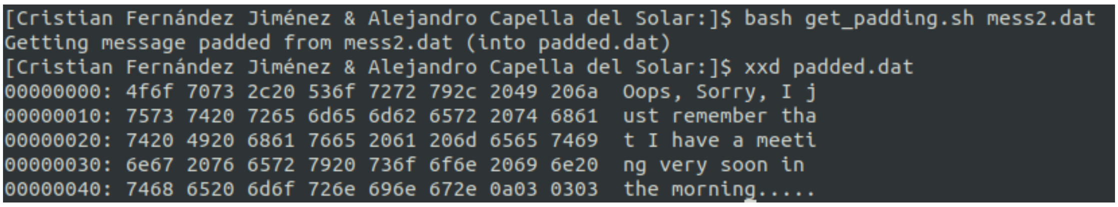
As we see, the last block incomplete is terminated with the value of the bytes missing, in this case 15 (0f). Which is, if there are n missing bytes in the block, then the padding is n bytes all with the value n.

In a sort of practical attack, and following the previous instructions, we also came up with a script to generate a padded message, without having the key: We decrypt with the option nopad set up.

```
#! /usr/bin/bash

if [ "$#" -ne "1" ]
then
```

```
  echo 'Introduce file with message to pad'
  echo "$0 file"
    exit 1

else

  echo "Getting message padded from $1 (into padded.dat)"
  total_bytes=`wc -c $1 | cut -f1 -d" "`
  padding=$(( 16 - ($total_bytes%16) ))

  if [[ $padding -eq 0 ]] # FACT 1, IF IV=01FFxx
  then
    padding=16
  fi

  cat $1 > padded.dat

  for (( i=0; i<$padding; i++ ))
  do
    printf "\x$(printf "%02x" "$padding")" >> padded.dat
  done
fi
```

Listing 1.2: Bash script to get AES-128-CBC padding from a message file



```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ bash get_padding.sh mess2.dat
Getting message padded from mess2.dat (into padded.dat)
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd padded.dat
00000000: 4f6f 7073 2c20 536f 7272 792c 2049 206a  Oops, Sorry, I j
00000010: 7573 7420 7265 6d65 6d62 6572 2074 6861  ust remember tha
00000020: 7420 4920 6861 7665 2061 206d 6565 7469  t I have a meeti
00000030: 6e67 2076 6572 7920 736f 6f6e 2069 6e20  ng very soon in 
00000040: 7468 6520 6d6f 726e 696e 672e 0a03 0303  the morning.....
```

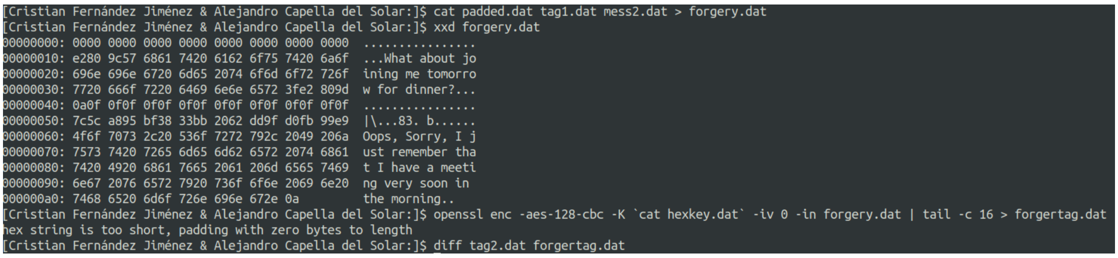
Figure 1.4: Padded message by adding missing bytes until end of block

As we perceive, 3 bytes are appended, as those are the missing ones to complete the block.

5. The last step would be create a forgery message, from the known messages and their tags. In this case, we would concatenate as follows:

$$HEADER + MESS1 + PADDING + TAG1 + MESS2$$

The creation of the forgery and its tags are shown in these commands, in addition with *diff* we verify there are no differences between the actual tag and the forged one. With these steps we have proved that is possible to come with a forged message/tag pair using CBC-MAC.



```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat padded.dat tag1.dat mess2.dat > forgery.dat
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd forgery.dat
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000020: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000030: 7720 666f 7220 6469 6e6e 6572 3fe2 809d  w for dinner?...
00000040: 0a0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f  ................
00000050: 7c5c a895 bf38 33bb 2062 dd9f d0fb 99e9  |\...83. b......
00000060: 4f6f 7073 2c20 536f 7272 792c 2049 206a  Oops, Sorry, I j
00000070: 7573 7420 7265 6d65 6d62 6572 2074 6861  ust remember tha
00000080: 7420 4920 6861 7665 2061 206d 6565 7469  t I have a meeti
00000090: 6e67 2076 6572 7920 736f 6f6e 2069 6e20  ng very soon in
000000a0: 7468 6520 6d6f 726e 696e 672e 0a        the morning..
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ openssl enc -aes-128-cbc -K `cat hexkey.dat` -iv 0 -in forgery.dat | tail -c 16 > forgertag.dat
hex string is too short, padding with zero bytes to length
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ diff tag2.dat forgertag.dat
```

Figure 1.5: Forged message creation and inspection, and tag generation and verification

1.2 One-pass HMAC length extension attack

In a similar way, we can forge a pass message without knowing the key. Knowing a message and its corresponding tag, an attacker can perform a length extension procedure with the only limitation that the concatenated string must start wihth a particular padding to the first message.

The structure that we are going to follow on this exercise is the following, in order to simulate an actual attack:

Preparation phase: Simulation of the victim, who would generate a message and a random 16-byte key. Then, obtain the digest of their concatenation, by using the MD5 option of OpenSSL.

```
cat key.dat message.dat | openssl dgst -md5 -binary > tag.dat
```

We assume this done and does not require any further methods.

Attack phase: We start by knowing message and tag. The goal is to concatenate a message and receive a valid tag that makes it authentic. The procedure will consist of a padding generation of the initial message to *glue* the second one and make MD5 to generate a valid tag. This will be achieved by tweaking the code in the OpenSSL library, so we can start a concatenation of data from a valid state (end of the initial padded message).

Validation phase: Once the tag has been generated, we may be able to validate it by comparing the one that we could generate if we knew the key. The forged tag and the legit one must be equal.

1.2.1 Length extension attack

As it was said before, the message must be padded if we plan to *glue* an additional one. Then the tag would be processed only if the forgery is correct and the state is initialized correctly.

1.2.1.1 Adding a padding

For the MD5 hash function, there is a block size of 512 bits. Padding takes the form `0x80 0x00 ... 0x00 n0 n1 ... n7`, where the last 8 bytes encode the length in bits of the unpadded message (little-endian) and the 0s are optional. This means that padding requires 65 mandatory bits, or as we have assumed, 72 bits (bytes will not be split as we are ussing plain text messages).

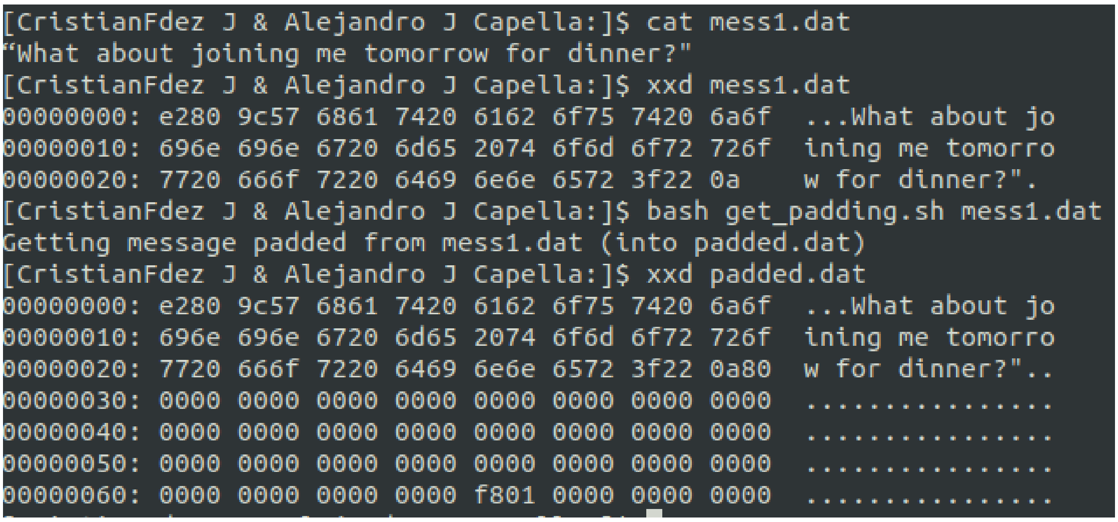
We have implemented a bash script to achieve it. This script will accept the message (without the key)[1] and will generate the neccesary padding depending on its size.

```bash
if [[ $lastblock_bits -eq 0 || $lastblock_bits -le $(( $block_size - $mandatory_bits )) ]] #LAST BLOCK IS COMPLETED (64 BYTES) or FITS MANDATORY PADDING (9 BYTES)
then

  necessary_zeros=$(( $bytes_available - $mandatory_bytes ))

  printf "\x$(printf "%02x" "128")" >> padded.dat #APPEND 0x80
  for run in $( seq 1 $necessary_zeros ); do printf "\\x$(printf "%x" "0")"; done >> padded.dat #APPEND NECESSARY 0x00
  echo 00: `printf "%016x" $total_bits` | xxd -r | xxd -g 8 -e | xxd -r >> padded.dat #APPEND SIZE in LITTLE-ENDIAN (8 BYTES)

elif [[ $lastblock_bits -gt $(( $block_size - $mandatory_bits )) && $lastblock_bits -lt $block_size ]] #LAST BLOCK DOESN'T FIT MANDATORY PADDING
then

  necessary_zeros=$(( $bytes_available - 1 ))

  printf "\x$(printf "%02x" "128")" >> padded.dat #APPEND 0x80
  for run in $( seq 1 $necessary_zeros ); do printf "\\x$(printf "%x" "0")"; done >> padded.dat  #APPEND NECESSARY 0x00 TO FINISH BLOCK

   #--BLOCK FINISHED--
```

[1] **Important**: Note that 16 bytes are missing, as we don't know the key. This is done mainly to act fully as an attacker, the bits will be counted back in the forgery script.

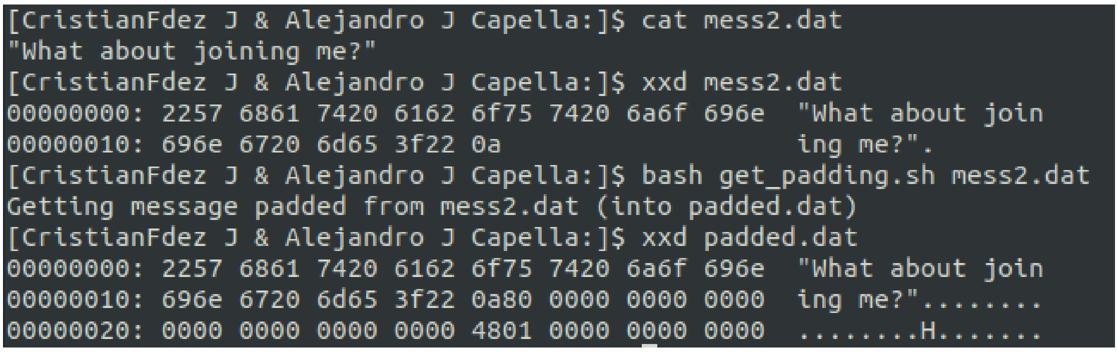
```

    for run in {1..56}; do printf "\\x$(printf "%x" "0")"; done >> padded.dat  #APPEND 56 BYTES OF 0x00 TO FINISH BLOCK
    echo 00: `printf "%016x" $total_bits` | xxd -r | xxd -g 8 -e | xxd -r >> padded.dat #APPEND SIZE in LITTLE-ENDIAN (8 BYTES)


  fi
```

Listing 1.3: Script building the message padding



```
[CristianFdez J & Alejandro J Capella:]$ cat mess1.dat
“What about joining me tomorrow for dinner?"
[CristianFdez J & Alejandro J Capella:]$ xxd mess1.dat
00000000: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000010: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000020: 7720 666f 7220 6469 6e6e 6572 3f22 0a    w for dinner?".
[CristianFdez J & Alejandro J Capella:]$ bash get_padding.sh mess1.dat
Getting message padded from mess1.dat (into padded.dat)
[CristianFdez J & Alejandro J Capella:]$ xxd padded.dat
00000000: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000010: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000020: 7720 666f 7220 6469 6e6e 6572 3f22 0a80  w for dinner?"..
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 f801 0000 0000 0000  ................
```

Figure 1.6: Padded message by adding an extra necessary block



```
[CristianFdez J & Alejandro J Capella:]$ cat mess2.dat
"What about joining me?"
[CristianFdez J & Alejandro J Capella:]$ xxd mess2.dat
00000000: 2257 6861 7420 6162 6f75 7420 6a6f 696e  "What about join
00000010: 696e 6720 6d65 3f22 0a                   ing me?".
[CristianFdez J & Alejandro J Capella:]$ bash get_padding.sh mess2.dat
Getting message padded from mess2.dat (into padded.dat)
[CristianFdez J & Alejandro J Capella:]$ xxd padded.dat
00000000: 2257 6861 7420 6162 6f75 7420 6a6f 696e  "What about join
00000010: 696e 6720 6d65 3f22 0a80 0000 0000 0000  ing me?"........
00000020: 0000 0000 0000 0000 4801 0000 0000 0000  ........H.......
```

Figure 1.7: Padded message by finishing last block

1.2.1.2 Forge a message

To come with the length extension attack, it's not as trivial as the previous attack, as we need to tweak the code in the OpenSSL library in order to establish a valid current state to append the new data.

The computation of a hash function in OpenSSL consists of three steps: initialize, update and finalize. All the internal state of the computation is placed into a context object. Seen this, we need to implement an intermediate step, for the purpose of initialize the context with on the *glueing* point.

The following function is intended to be called just after `MD5_Init(MD5_CTX *)`:

```
void set_ctx(MD5_CTX *pctx, const char *digest, unsigned long nblocks) {
    pctx->A = gethexword32(digest);
    pctx->B = gethexword32(digest+4);
    pctx->C = gethexword32(digest+8);
    pctx->D = gethexword32(digest+12);
    nblocks <<= 9; // converting into bits
    pctx->Nh = nblocks>>32;
    pctx->Nl = nblocks&0xFFFFFFFFul;
}
```

Note that shifting is being reduced to the half, as digest is read as plain text from a file. In this way we read the 16 bytes properly.

In addition, `MD5_LONG gethexword32(const char *digest)` needs to be implemented. It reads a 4 byte integer from its hexadecimal representation, interpreting the byted in little-endian order.

```
MD5_LONG gethexword32(const unsigned char *digest){

    MD5_LONG var =(unsigned) digest[0]|((unsigned)digest[1]<<8)|((unsigned)digest[2]<<16)|((unsigned)digest[3]<<24); //Little endian to Int

    return var;
}
```

From this point, it is straightforward to reconstruct the last internal state of the computation and obtain a forged tag.
We propose the following function, which accepts a padded message file, its tag and the new data to append and returns a valid new tag:

```
unsigned char *file2md5(const char *filename, const char *digest_file,
const char * newdata_file)
```

.

Inside this function, we are going to follow the same procedure[2] that OpenSSL would do to generate the digest, but with an intermediate point to establish the wanted state.

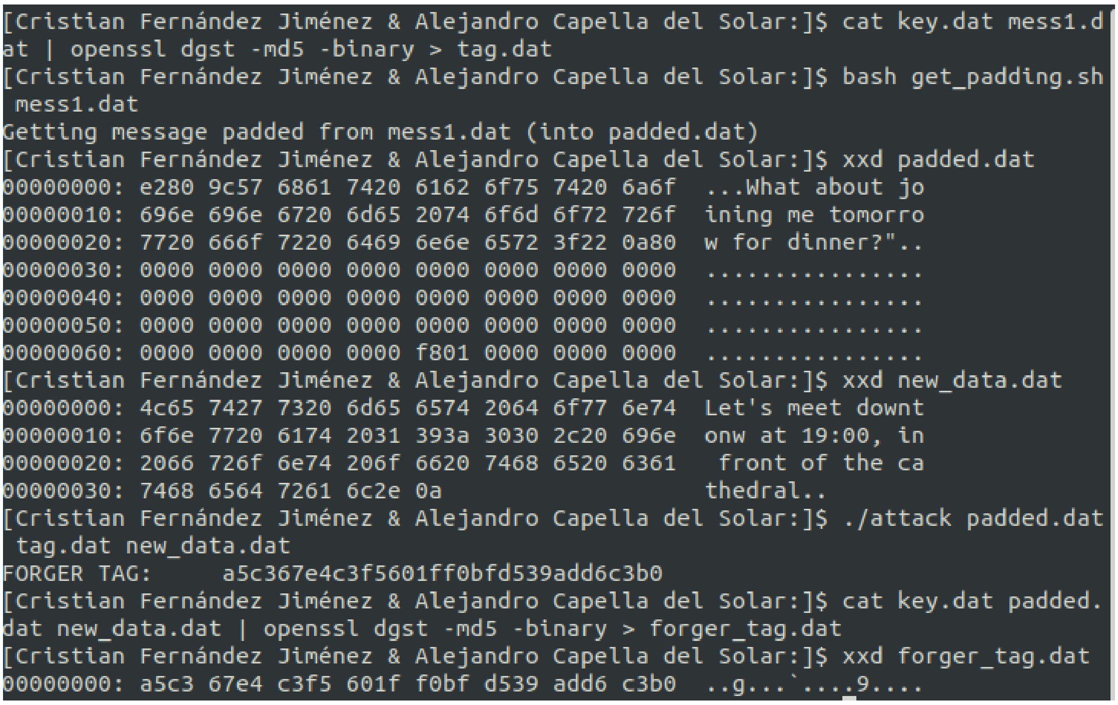
```
unsigned char *file2md5(const char *filename, const char *digest_file, const char * newdata_file) {
    char * msg = 0;
    long length;
    long total_bytes;
    unsigned long nblocks;

    \\... Read message file into msg

    total_bytes=length+16; //Bytes of padded message + key size
    nblocks=total_bytes/64;
    MD5_CTX c;
    unsigned char * digest=0;

    \\... Read tag file into digest

    MD5_Init(&c);

    set_ctx(&c,digest,nblocks); //Set context according to padded message and passing the tag

    char * new_data=0;

    \\... Read new message file into new_data

    //Call update for each new block of data
    while (length > 0) {

        if (length > 64) {
            MD5_Update(&c, new_data, 64);
        } else {
            MD5_Update(&c, new_data, length);
        }
        length -= 64;
        new_data += 64; //Shift to next block of data
    }

    unsigned char new_digest[MD5_DIGEST_LENGTH];

    MD5_Final(new_digest, &c); //Get the new forged tag

    \\... Return new digest
```

[2] Note that 16 bytes are counted back, as we omitted them in the padding gen.

```
    }
```

1.2.2 Attack verification

Once we have achieved to forge a tag for an extended message, we must put it alongside a legitimate one, that we would generate using the real key.



```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat key.dat mess1.d
at | openssl dgst -md5 -binary > tag.dat
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ bash get_padding.sh
 mess1.dat
Getting message padded from mess1.dat (into padded.dat)
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd padded.dat
00000000: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000010: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000020: 7720 666f 7220 6469 6e6e 6572 3f22 0a80  w for dinner?"..
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 f801 0000 0000 0000  ................
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd new_data.dat
00000000: 4c65 7427 7320 6d65 6574 2064 6f77 6e74  Let's meet downt
00000010: 6f6e 7720 6174 2031 393a 3030 2c20 696e  onw at 19:00, in
00000020: 2066 726f 6e74 206f 6620 7468 6520 6361   front of the ca
00000030: 7468 6564 7261 6c2e 0a                   thedral..
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ ./attack padded.dat
 tag.dat new_data.dat
FORGER TAG:     a5c367e4c3f5601ff0bfd539add6c3b0
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat key.dat padded.
dat new_data.dat | openssl dgst -md5 -binary > forger_tag.dat
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd forger_tag.dat
00000000: a5c3 67e4 c3f5 601f f0bf d539 add6 c3b0  ..g...`....9....
```

Figure 1.8: Complete attack sequence and tag verification

We created a message file `mess1.dat` and hashed it on `tag.dat`. As attackers, calling the script to get the padding would only need the message. Once received, we generated a new message to forge. With these files, padded message, tag and new message we are able to call the attack script, getting a brand new tag.
We may validate this by generating a digest of the concatenation of key, padded initial message and new message. And as we can see, `forger_tag.dat` contains the same digest as the one generated by our script.

With these steps we have proved that is possible to come with a forged message/tag pair using HMAC.