Cryptography - Exercise 5

Stephanie Briere Americo / 22579627

January 15, 2019

1 CBC Malleability (Aufgabe 1)

The Cipher Block Chaining (CBC) mode of encryption is a technique that makes a XOR between the current plaintext block and the previous ciphertext block to generate the current ciphertext block, when encrypting. At decryption, it makes a XOR of the the current block passed through the decipher and the previous ciphertext block to get the current plaintext block being processed.

This and the other modes of operation on block ciphers were created to try to address some issues with the simple ECB mode which just pass a block through a cipher algorithm. All the common modes offers some advantages as well as some drawbacks.

At a CBC decryption, a plaintext is recovered from two adjacent ciphertext, meaning that a change on one ciphertext block will not cause the loss of all the information. This is used on this attack. As demonstrated in [1], an attacker in charge of a plaintext and its ciphertext can craft a valid ciphertext for another plaintext (with some limitations) of its own choice without knowing the key used.

The general idea is to manipulate two blocks of cipher and plaintexts with XOR operations, so we can create a ciphertext block at the position $i - 1$ that will produce a desired plaintext when used to decrypt the block at the position i. The drawback is that doing this corrupts the plaintext at the position i, but this is the "REASON" field at the transaction is this example, so it will not have a huge impact for us.

The `change_request.py` script reads a plaintext and ciphertext from standard input and do the operations to change the destination account, and outputs the base64 encoding of the new request. It has some commentaries about the operations, which are described in details in [1], on the description of attack section.

At `playing_with_hex` there are just some annotations of plaintexts, ciphertexts and hexadecimals from different stages of the encryption. I wrote this to help me understanding and calculating the steps to be done to achieve the goal.

The principle which was violated was *integrity*, since the receiver is not able to check if the message has been tampered with, besides being able to guarantee the confidentiality. To solve this, a mechanism such as HMAC (when correctly implemented) should be used, as it provides a way to check the message integrity. Besides that, I think a signing scheme which uses private and public keys could be used too, since it would also guarantee non-repudiation.

[1] `http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/`

[2] `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Block_Chaining_(CBC)`

2 HMAC Length Extension Attack (Aufgabe 2)

At [1] there is a very detailed explanation I found on Length Extension Attacks, and also a very complete tool to do this work.

The idea of the HMAC is that server and user share a secret key. When making a request to the server, we need to send the data and a Message Authentication Code, which should prove to the server that the entity doing the request is the user, because only the user (with the secret key) could be able to generate the correct MAC for the data. The HMAC is a MAC build up using hashes.

In this server the HMAC is constructed by concatenating the data to the secret key and then calculating the SHA256 digest of this string. Merkle-Damgard based hashes (as SHA256, MD5 and SHA1) have the following general way of work: it receives a data, apply a padding to fill the data up to a block size multiple, and then apply a block operation that transform the data in a specific way. This hashes starts with a predefined internal state values (like an initialization vector) and it do some operations in every block of fixed size. This operations receives the block and the internal state as input and combines it to a new internal state, and then go on to the next block.

This way of work leads to the Length Extension Attacks: if an attacker knows the hash used (in this case the question gave this information, but it can be guessed by e.g. the MAC size) and a pair (plain data, HMAC), then it could be easy enough to recreate the internal state of the hash function.

The padding scheme for SHA256 is: one bit '1' followed by as many '0' as necessary, followed by a 32-bit number, representing the number of bits of the original message and the padding, excluding this size number. This padding is added to the message to fill up the block, so the final digest is the same if we try to digest a string with the final padding or without it.

The final digest represent the internal state of the hash system when the input ended, so we can recreate it and then "continue" to hash, as if the message would not be ended yet. All we need to discover is the actual secret key length: the key length is what will determine (alongside the message (which is fixed in this attack)) what the padding will be, so we need to try it by brute-force: the idea is to append the padding relative to the key length x to the original data, then append our string and continue to has by starting with the internal state configured to the digest we have. Note that having this internal state, we don't need the secret, as the internal state already accomplishes the processing for "secret key + original data". This will give us the hash for this new message, so we can try sending it to the server. If the request fails, then the guessed key length was wrong, otherwise it was right and the server should update the message showed.

The tool available at [1] do exactly this: feeding it with the has, original data and digest, it generates all the possible messages and signatures for different key lengths. It generates the appropriate padding (for the hash you chose) and manipulate the internal state of the hash system to "continue" the processing.

The tool is at the `hash_extender` directory, and is compiled with just make. At the "cmd_line" file there is the exact command line used to retrieve the right message (appending "Stephanie" to it) and signature. The only difference is that when I was trying to discover it, I changed some options to retrieve the signature and the message separately.

With the question hint about messages starting with the letter "H", I searched first for the results with this condition and the low key length numbers, and found that the key length is 20.

Regarding the question about letter "H", it is possible to check this by using `hexdump -C` at the new, extended message: After the original data (which have 373 characters), we can see the padding with '1' and many '0's, before 2 hexadecimal numbers ("0C 48") and then our injected message. This last hexadecimal numbers convert to 3144 bits or 393 bytes, which matches up to $373 + 20$ bytes of padding prior to the 32-bit size field. It happens that the hexadecimal "48" translates the letter "H" on the ASCII-table, instead to the others, non-ASCII characters that are

ignored by the server. Because the original message and key are always the same, so the padding will be, and then all the messages will ”receive” an extra ”H” at the beginning of the appended string.

[1] `https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks`

[2] `https://en.wikipedia.org/wiki/Length_extension_attack`

3 RSA Small Exponent (Aufgabe 3)

Using `openssl rsa -text -pubin < key_file` I was able to discover some things about the 3 public keys. All the 3 exponents are 3. The beginning of the 3 keys are the same (as for a same header, defined in the protocol), and also the very end of the keys are identical (as for the exponent). It is also possible to see the modulus of the keys. This way I extracted the modulus to the respective files modulus.{1,2,3}.

As described in [1] and [2], if the same message encrypted with different keys that have the same small public exponent ϵ (in this case $\epsilon = 3$) have no padding at all or even no random padding (in this case, the question gave us a hint that no padding was applied), than we can recover the plain text by calculating a ciphertext C (with $C = C_1 mod N_1$, $C = C_2 mod N_2$ and $C = C_3 mod N_3$) using the Chinese Remainder Theorem [3], and then calculating the cubic root of C.

The implementation from [4] were used to decrypt the message (file ”hastads.py”) (it turned out to be very simple and similar to what I was doing by myself). By passing the 3 ciphertexts and 3 key modulus as hexadecimals files, it returned the phrase ”The answer to life the universe and everything = 42”.

The general idea of the Chinese Remainder Theorem is that if the modulus are pairwise co-prime, then there is a unique solution C that satisfies the $C = C_i mod N_i$ condition for every i. The algorithm uses the Extended Euclidean algorithm to find the co-primes integers that should be multiplied to the modulus and ciphertext. Doing this for all 3 ciphertexts, we can take the cubic root to get the final answer (”canonical” answer). At [5] there are another explanation which made it more easy to me to understand the theorem, along with codes in many languages to solve the theorem problem.

[1] `https://en.wikipedia.org/wiki/Coppersmith%27s_attack`

[2] `https://www.coursera.org/lecture/number-theory-cryptography/hastads-broadcast-attack-fyPIB`

[3] `https://en.wikipedia.org/wiki/Chinese_remainder_theorem`

[4] `https://github.com/aaossa/Computer-Security-Algorithms/commit/1834ba974dd4804767d48d426a0bfc2c0071847c`

[5] `https://rosettacode.org/wiki/Chinese_remainder_theorem`

4 ECDSA Fixed K (Aufgabe 4)

Using openssl (command line at ”cmd_line”) with the public key it is possible to discover the curve used (NIST Prime 256 v1), and then its predefined parameters. The parameters were used

directly from the ”python ecdsa” library that was recommended, but at [1] it is possible to check the parameters of many elliptic-curves in different formats.

By using `hexdump -C` we are able to see that more than the first half of the signatures are the same. Reading about how ECDSA signatures are generated, I found out that there are two integers, r and s, and that the integer r was almost surely the one shared by the two signatures. Looking for problem with ECDSA with low entropy, I found that the reuse of the nonce k used in the signature calculation surely leads to signatures with the same number r. Because of the design of the algorithms, if an attacker knows 2 messages which share the r part of its signature and the underlying hash used, it is sufficient to the attacker to retrieve the private key used to sign the messages.

At [2], on the signature generation algorithm, it is showed that with this information we can find the numbers z based on the hash digest of the plaintext and then we can calculate the nonce k and finally the private key used.

The `get_key.py` script is responsible for reading the two messages and signatures provided and to outputting the PEM encoded private key, which was stored at `private_key`. The key was used to sign the message at ”my_message.txt”, and the same command line provided in the materials could be used to verify the signature (as in the shell script ”sign_and_verify.sh”). [3] and the source code of ”python ecdsa” [4] were of great help to get the script working and simple.

[1] `https://safecurves.cr.yp.to/`

[2] `https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm`

[3] `https://bitcoin.stackexchange.com/a/58859`

[4] `https://github.com/warner/python-ecdsa`