601.445/600.454: Practical Cryptographic Systems	September 26, 2018
Assignment 2	
Instructor: Matthew Green	*Due: 11:59pm, October 10*

Name: ____

The assignment should be completed individually. You are permitted to use the Internet and any printed references.

Please submit the completed assignment via Blackboard.

Problem 1: Warm up: implementing a cryptographic specification (30 points)

Implement the following specification for encrypting and decrypting messages. You must implement your project in the Go language. As a building block you must use an implementation of the "raw" AES cipher (meaning, AES encrypting single blocks in ECB mode) from the `crypto/aes` package and `crypto/cipher.Block`. You may obtain the SHA256 hash function from `crypto/sha256`. You may also use random numbers from the `crypto/rand` package. **You must implement both CBC mode and HMAC yourself. Do not use existing Go packages or online code to implement anything beyond the raw AES cipher.**[1]

Submission and grading: In order to allow for grading to be efficient, there will be strict submission guidelines. Failure to follow will result in a grade penalty. Make sure that your program takes input exactly as the specification indicates. Similarly, ensure that programs output exactly what is specified. We will be providing a testing harness that will mock the expected inputs and outputs so that everyone can validate their code before submitting. Additionally, please make sure that the code submission for each part of the assignment is a self-contained .go file. We know that organizing code into packages is good practice, but we have decided it is preferable to have a consistent build pattern for all submissions.

Interface: Your code must use the following command-line interface:

```
encrypt-auth <mode> -k <32-byte key in hexadecimal> -i <input file> -o <output file>
```

Where `<mode>` is one of either `encrypt` or `decrypt`, and the input/output files contain raw binary data. You should parse the first 16 bytes of the key as the encryption key k_{enc}, and the second 16 bytes as the MAC key k_{mac}.

Notation. We denote concatenation by $||$. By $|M|$ we denote the length of an octet-string M in *bytes*. Recall that AES has a fixed block size of 16 bytes.

Encrypt(k_{enc}, k_{mac}, M). Given a 16-byte secret key k_{enc}, a 16-byte secret key k_{mac}, and a variable-length octet string M, encrypt M as follows:

[1] For a description of the HMAC construction, see FIPS 198 or even Wikipedia.

1. First, apply the HMAC-SHA256 algorithm (see RFC 4634) on input (k_{mac}, M) to obtain a 32-byte MAC tag T.
2. Compute $M' = M||T$.
3. Compute $M'' = M'||PS$ where PS is a padding string computed using the method of PKCS #5 as follows: first let $n = |M'|\ mod\ 16$. Now:
   (a) If $n \neq 0$, then set PS to be a string of $16 - n$ bytes, with each byte set to the value $(16 - n)$.[2]
   (b) If $n = 0$ then set PS to be a string consisting of 16 bytes, where each byte is set to 16 (0x10).
4. Finally, select a random 16-byte Initialization Vector IV and encrypt the padded message M'' using AES-128 in CBC mode under key k_{enc}:

   $$C' = \text{AES-CBC-ENC}(k_{enc}, IV, M'')$$

   Note: you must implement the CBC mode of operation in your own code, although you are free to use a library implementation of the AES cipher.

The output of the encryption algorithm is the ciphertext $C = (IV||C')$.

Decrypt(k_{enc}, k_{mac}, C). Given a 16-byte key k_{enc}, a 16-byte key k_{mac} and a ciphertext C, decryption is conducted as follows:

1. First, parse $C = (IV||C')$ and decrypt using AES-128 in CBC mode to obtain M'':

   $$M'' = \text{AES-CBC-DEC}(k_{enc}, IV, C')$$

   Note: you must implement the CBC mode of operation in your own code, although you are free to use a library implementation of the AES cipher.
2. Next, validate that the message padding is correctly structured. Let n be the value of the last byte in M''. Ensure that each of the final n bytes in M'' is equal to the value n.

   If this check fails, output the distinguished error message "INVALID PADDING" and stop. Otherwise, strip the last n bytes from M'' to obtain M'.
3. Parse M' as $M||T$ where T is a 32-byte HMAC-SHA256 tag.
4. Apply the HMAC-SHA256 algorithm on input (k_{mac}, M) to obtain T'. If $T \neq T'$ output the distinguished error message "INVALID MAC" and stop. Otherwise, output the decrypted message M.

Problem 2: Active Attacks (60 points)

In the first part of this assignment you were asked to implement a cryptographic specification. This resulted in a utility for encrypting and decrypting using a symmetric key K.

[2] For $n = 9$ this would produce the padding string: 0x 07 07 07 07 07 07 07

In this part of the assignment you will develop a tool that programmatically decrypts any ciphertext produced by your encryption utility from Part 1. Your tool will not have access to the decryption key. It will instead call a second program that attempts to decrypt the ciphertext using the decryption key, and returns an error on failure.

The command line profile for your tool will be as follows:

```
decrypt-attack -i <ciphertext file>
```

This program will take as input a ciphertext encrypted with the key K. When it completes it should output the decryption of the ciphertext. This program will call a second program called `decrypt-test` that *has the key* K *hardcoded into it.* The profile for `decrypt-test` will be as follows:

```
decrypt-test -i <ciphertext file>
```

The utility will have a hard-coded decryption key. It *will not return the decrypted ciphertext*, but instead only a single one of the following three response messages:

1. "SUCCESS"
2. "INVALID PADDING"
3. "INVALID MAC"

You are expected to implement the tool `decrypt-test` using your own code from Part 1 of this assignment, though you do not have to turn it in. You only need to turn in `decrypt-attack`, as we will provide our own implementation of `decrypt-test` for grading.

For test purposes you should also generate your own key K (to hard-code into `decrypt-test`) and generate a test ciphertext based on some plaintext of input size at least 256 bytes.

Problem 3: Padding oracles in practice (10 points).

TLS and Datagram TLS (DTLS) each use a form of encryption that's nearly identical to the scheme you implemented in Problem 1. The following partial code listing is responsible for processing a received DTLS packet (source: `d1_pkt.c` in OpenSSL version 1.0.0e).

Note that this listing calls two subroutines that are *not* shown here:

1. `s->method->ssl3_enc->enc(...)` performs decryption and check the padding. This returns -1 if the padding is invalid.
2. `s->method->ssl3_enc->mac(...)` computes the MAC.

If you want to look at those subroutines you can find them in the OpenSSL codebase at openssl.org. But you shouldn't need them to answer the questions below.

```
static int
dtls1_process_record(SSL *s)
{
...
```

```
/* decrypt in place in 'rr->input' */
rr->data=rr->input;

enc_err = s->method->ssl3_enc->enc(s,0);  /* <<<--- decryption/padding check */
if (enc_err <= 0)
{
/* decryption failed, discard message */
if (enc_err < 0)
{
rr->length = 0;
s->packet_length = 0;
}
goto err;
}

...

/* r->length is now the compressed data plus mac */
if ( (sess == NULL) ||
(s->enc_read_ctx == NULL) ||
(s->read_hash == NULL))
clear=1;

if (!clear)
{
/* !clear => s->read_hash != NULL => mac_size != -1 */
int t;
t=EVP_MD_CTX_size(s->read_hash);
OPENSSL_assert(t >= 0);
mac_size=t;

if (rr->length > SSL3_RT_MAX_COMPRESSED_LENGTH+mac_size)
{
...
}
/* check the MAC for rr->input (it's in mac_size bytes at the tail) */
if (rr->length < mac_size)
{
...
}
rr->length-=mac_size;
i=s->method->ssl3_enc->mac(s,md,0);  /* <<<--- MAC computation */
if (i < 0 || memcmp(md,&(rr->data[rr->length]),mac_size) != 0)
{
goto err;
```

```
}
}

...

f_err:
ssl3_send_alert(s,SSL3_AL_FATAL,al);
err:
return(0);
}
```

1. Assume that the attacker is sending packets to the server for decryption, and that he receives some notification from the server when the routine outputs an error (*i.e.,* returns 0). How might he execute a padding oracle attack? (Hint: this is trickier than it looks!)