**650.445/600.454: Practical Cryptographic Systems** February 24, 2014

# Assignment 1

*Instructor: Matthew Green* *Due: 3:00pm, March 12*

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 are free to use any reasonable programming language, although I recommend C, Python or Java. You can obtain an implementation of the AES cipher ("ECB mode") and the SHA1 hash function from a cryptographic library, *e.g.,* OpenSSL, PyCrypto or BouncyCastle. Your code must compile on one of the standard graduate lab machines (MSSI, UGrad, or Masters lab). **However you must implement both CBC mode and HMAC yourself.**[1]

*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. First, apply the HMAC-SHA1 algorithm on input $(k_{mac}, M)$ to obtain a 20-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.*

[1] For a description of the HMAC construction, see FIPS 198 or even Wikipedia.
[2] For $n = 9$ this would produce the padding string: 0x 07 07 07 07 07 07 07

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 20-byte HMAC-SHA1 tag.

4. Apply the HMAC-SHA1 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).

You have been asked to review a web service that uses a CAPTCHA to prevent automated login attempts. For your evaluation you have been given access to a proof-of-concept implementation at the following URL:

`http://cs.jhu.edu/~cgarman/PracticalCrypto.html`

You should visit the site to determine how the mechanism works. Your goal in this problem is to develop a tool that programmatically solves the CAPTCHA — *i.e.,* can log into the site efficiently, and without human interaction. Moreover, you should *not* accomplish this by developing an algorithm to solve the CAPTCHA itself. It is expected that your tool will use the web server as an "oracle", interacting with it using basic HTTP communication techniques.

As a deliverable, you must provide the source code for your tool.

*Note 1:* The web server is a shared resource. To avoid denial of service, please throttle your access by adding a 15ms delay between repeated connections.

*Note 2:* The purpose of this assignment is to use a cryptographic technique to decrypt cookies. Please do not attempt to find non-cryptographic vulnerabilities in the web server (*e.g.,* software exploits, password guessing, SQL injection)!

**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!)