Project 1 – CCA2 Encryption

Due: Sunday, March 14th

Synopsis

In this assignment you are asked to build a public key cryptosystem using a ***key encapsulation mechanism***. The idea is that by using a hybrid encryption scheme (combining an asymmetric and symmetric system), we can produce a highly efficient public-key system, thus getting the best of both worlds.

Goals for the student

- Understand different security definitions for cryptosystems.
- Hands on experience programming with a variety of crypto building blocks (symmetric encryption, asymmetric encryption, hashing, MACs…).

Important Notes on Grading

I would like you all to collaborate on these projects in small teams. Teams should ideally have 4 people, but **no less than 3**. Details:

- Designate one member of your team to be the "leader" / repository owner. (They should set up a repository for the project and grant access to the rest of the team.)
- Every member should make commits to the leader's repository as they work. (Please don't make commits on behalf of another student – it helps me when grading to see who has done what via commit logs.)
- Team leaders – communicate to me (email is OK) the address of your repository. Please include "i4900" in the subject line.

The cryptosystem

Step 1: CCA2 symmetric encryption

First, we build CCA2 symmetric encryption from the weaker assumption of CPA encryption. Let f_k denote our symmetric encryption with key k, and let $h_{k'}$ denote our MAC with key k'. To encrypt a bit string m, we set $c = f_k(m)$, and set the ciphertext to the pair $(c, h_{k'}(c))$. Decryption of a pair (x, y) first makes sure that $h_{k'}(x) = y$; if this fails, output $\perp$, otherwise decrypt x and output the result.

Given that f_k is CPA secure and that $h_{k'}$ is pseudorandom, it is well known that this construction is CCA2 secure. The key idea is that the MAC makes the adversary's decryption queries useless: for any ciphertext which was not the output of the encryption oracle, the output will invariably be $\perp$: To find a valid ciphertext ***is*** to forge the MAC. Formal proof is left as an exercise (use any CCA2 adversary to build a CPA adversary with almost the same advantage by emulating a CCA2 ***challenger***).

Step 2: KEM to make it public-key

The idea is very simple: create a random key for the above scheme, encrypt the message you want to send, and then send it, along with a ***public-key encryption of the symmetric key***. The analysis is a little tricky though. To preserve the CCA2-ness, we can't just send a public-key encryption of the key – we need a ***key encapsulation mechanism*** which has some special properties. In particular, we need our KEM to have an analogous property to CCA2 for an encryption scheme: an adversary with access to a "decapsulation" oracle (a box that outputs the key from its encapsulation) cannot differentiate between valid encapsulations (where the key corresponds to the ciphertext), and random keys. Obviously the same CCA2 rule of "you can't decrypt the challenge" applies, but other than that, anything goes.

How to build such a thing? It turns out that all you need is a public key encryption (plain, deterministic RSA works!), a key derivation function (HMAC will do fine), and a hash function (we could use HMAC again, but we must make sure it is with a different key). Letting KDF denote the key derivation function, E_{pk} the encryption (with public key pk) and letting H denote the hash, then the KEM construction is as follows: select a random message x (needs at least as much entropy as your key!) and then let $C = (E_{pk}(x), H(x))$ be the encapsulation, while $KDF(x)$ is the key. The "decapsulation" algorithm on input $C = (C_0, C_1)$ simply computes $x = D_{pk}(C_0)$, and outputs $KDF(x)$ if $H(x) = C_1$; otherwise it outputs $\perp$. It isn't too hard to prove this has the property we need. (See Dent 2003 for the details.)

Why is the composition CCA2 secure?

There is a nice hybrid-style argument in (Cramer and Shoup 2003, chap. 7), but verifying all the details would take us a little off course. Here's the gist though: how different could the CCA2 game be if we swapped out the encapsulated key with a totally random key for the symmetric encryption? Not very! Even if we gave the adversary the ability to run decapsulation queries, he can't distinguish the cases (this is exactly our definition of CCA2 for a KEM). But now if the key is random, this is precisely the situation for which we've proved CCA2 security of the symmetric scheme. Voila.

Details

I've given you a skeleton in C, but you can write the program in other languages if you want, **as long as you follow the guidelines**. Look at the section on other languages for details.

Regarding the C skeleton

To facilitate the development, you can use GMP for the long integer arithmetic needed for RSA, and libressl or OpenSSL for various cryptographic primitives like hashing and symmetric encryption (they both provide a library `libcrypto` with these things).[1]

I've given you a skeleton, as well as some examples that you can draw upon. The stubs that you are supposed to fill out are labeled "TODO". Unless you have a super-compelling reason, I would recommend that you don't change the interface.

Building blocks:

- RSA for PKE. You will implement this yourself. Note that this is the naive, deterministic (and hence not even IND-CPA secure) version. But it will work fine for our KEM.

- AES for symmetric encryption. You can get this from `libcrypto`. We'll use it in counter mode for optimal speed during encryption.[2]
- HMAC for a MAC. Also available via `libcrypto`.

Be sure to read `man 4 random` at some point.

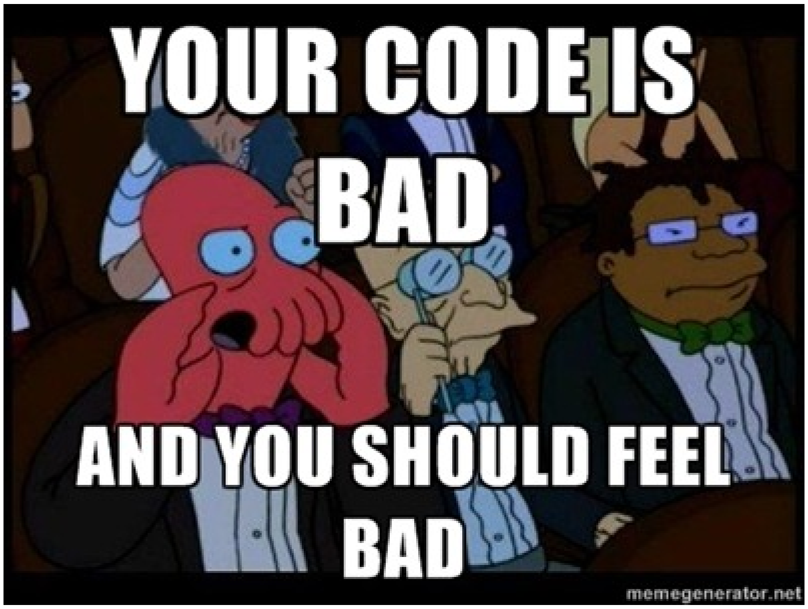
Hints / even more details

What to do, and when

I'd attack this in the following order:

1. RSA
2. SKE (only on buffers)
3. SKE that works on files
4. KEM (shouldn't be too challenging once you have the other pieces)

There are some basic tests for RSA and the memory buffer version of SKE (`ske_encrypt` / `ske_decrypt`) in the `tests/` directory, so those are good to start with. Once you have that working, implement the versions which operate on files. ***Hint:*** For this, I would recommend `mmap`. Then you can just hand off the pointers from `mmap` to the simple versions and let the kernel do all the buffering work for you. (Nice, right?) Or if you are lazy, you can also just read the entire file contents into a (potentially huge) buffer. But Zoidberg will be mad at you.



Extra notes on the KDF for symmetric encryption

Note: for the KEM scheme, both the KDF and the hash function are public. To ensure "orthogonality" of the two, one is implemented via HMAC, but the key is public (it is hard-coded into `ske.c` – see `KDF_KEY`). Note that the KDF should be handled inside of this function:

```
int ske_keyGen(SKE_KEY* K, unsigned char* entropy, size_t entLen);
```

If the `entropy` buffer is supplied, the KDF should be applied to it to derive the key. Thus when implementing `kem_encrypt`, you can take the encapsulated key `x` and supply that as `entropy`. Maybe something like this:

```
unsigned char* x = malloc(len);
/* ...fill x with random bytes (which fit in an RSA plaintext)... */
SKE_KEY SK;
ske_keyGen(&SK,x,len);
/* ...now encrypt with SK... */
```

Basic usage (command line interface)

This is documented via the usage string (as well as by looking at the test script), but here are some examples.

Generate a 2048 bit key, and save to /tmp/testkey{,.pub}:

```
./kem-enc -b 2048 -g /tmp/testkey
```

Encrypt `file` with the public key and write ciphertext to `ct`:

```
./kem-enc -e -i file -o ct -k /tmp/testkey.pub
```

Decrypt `ct` with the private key and write plaintext to `file0`:

```
./kem-enc -d -i ct -o file0 -k /tmp/testkey
```

Compiling, testing, debugging

As mentioned, there are some test programs in `tests/` for the RSA and SKE components. (You can build these via `make tests`.) For the hybrid KEM scheme, there's a `kem-test.sh` script. Fill the `tests/data/` directory with some files, and it will check if encrypt and decrypt at least compose to be the identity on those inputs. Also, there is a make target called `debug` to add a few helpful compiler flags. (Run `make -B debug` to recompile with debugging flags enabled.)

Other languages

If you want to do this in another language (or without the skeleton code), feel free to do so. Keep in mind that your code should speak the same language as the one described in the skeleton. That is,

- The binary file formats (for keys and ciphertext) should be the same.
- Your program should understand the same command line arguments.

Moreover, your code cannot assume additional cryptographic functionality beyond what is outlined above. In particular, **you must implement RSA from long integers**. You're welcome to get your hash functions and AES from somewhere other than `libcrypto`, but you can't rely on things that trivialize any part of the project.

Lastly, please provide a Makefile along with any instructions you think would help if you don't use the skeleton.

References

Cramer, Ronald, and Victor Shoup. 2003. "Design and Analysis of Practical Public-Key Encryption Schemes Secure Against Adaptive Chosen Ciphertext Attack." ***SIAM Journal on Computing*** 33

(1): 167–226.

Dent, Alex. 2003. “A Designer’s Guide to KEMs.” ***Cryptography and Coding***, 133–51.

---

1. Note that libressl/OpenSSL also contain implementations of RSA, but I want you to write this part yourself – it is more educational, and actually quite simple since “plain” RSA suffices for our application.↩

2. CBC mode encryption usually slower than CBC decryption. Can you guess why?↩