Assignment 3 writeup and relevant notes

ANOTHER UPDATE: You can now create your own ciphertexts via

```
echo -n "this is a plaintext" | encrypt1a
```

and likewise for `encrypt1b`, `encrypt2a`, and `encrypt2b`. This might be useful for testing your code...

(Note this code is setuid; you'll need to ensure that the current working directory is writeable by all, else you'll probably get error messages when it fails to create some files.)

UPDATE: I just recompile the oracles with a larger delay for the BADMAC (reflected in the updated `/srv/a3/constants.h`), as it was requiring quite a few repetitions to reliably distinguish between BADMAC and BADPAD. With the change, my code now consistently works without running obscenely slow.

For this assignment, your task is to implement (four variants of) a classic padding oracle attack.

In class we learned about several notions of secrecy for encryption schemes; namely, *indistinguishability under chosen plaintext attacks* (IND-CPA security) and *indistinguishability under chosen ciphertext attacks* (IND-CCA security). We formalized both of these using the notion of indistinguishability games. The padding oracle attack is a dramatic (and real, and fun to implement) example of (i) why IND-CPA security is often insufficient, (ii) why IND-CCA security is useful, and (iii) how easy it is for seemingly minor issues with crypto implementations to completely nullify theoretical security guarantees.

IND-CPA security game

To review, in the IND-CPA security game, the defender (D) samples a symmetric key k using the underlying encryption scheme's Gen algorithm and it flips an unbiased coin to obtain uniform random bit $\mathrm{b} \in \{0, 1\}$. From here, the attacker (A) is allowed to submit an arbitrary number of plaintexts, say $\mathrm{M}_1, \mathrm{M}_2, \ldots, \mathrm{M}_q$ to D and, for each submitted message, it obtains an encryption of that message under k (i.e., it receives $\mathrm{Enc}_k(\mathrm{M}_1), \mathrm{Enc}_k(\mathrm{M}_2), \ldots, \mathrm{Enc}_k(\mathrm{M}_q)$).

Note that $\mathrm{A}'\mathrm{s}$ is accessing D as a so-called *blackbox oracle*: It submits arbitrary plaintexts and receives honestly generated ciphertexts in return, but it cannot "peer inside" its oracle to learn anything about k or b that is not implied by these outputs. With that said, A does know precisely what the oracle is doing--like what algorithms it uses for Gen and Enc--and it knows that that oracle will never lie about its answers. Finally, note that A gets to choose the messages M_i adaptively; that is, it can strategically choose what messages it wants to submit next based on its observations about how the oracle responded to earlier queries. Finally, note that there is no hard limit on how many messages q the attacker A gets to submit; however, because A is assumed to by a polynomial-time algorithm, there is an implicit polynomial upper bound on how many messages it can possibly submit.

At some point, A submits an ordered pair of messages $(\mathrm{m}_0, \mathrm{m}_1)$ and receives back $\mathrm{Enc}_k(\mathrm{m}_b)$, where the subscript b is determined by the outcome of D's coin toss. When the game is written down, this challenge typically happens right at the end. However, it is technically allowed to happen at any time; that is, we allow A to keep submitting adaptive queries to D after this challenge, if doing so will help it figure out which of the two messages it sent were encrypted.

Finally, A outputs a guess b' for D's bit; it wins if and only if $\mathrm{b}' \overset{?}{=} \mathrm{b}$. Loosely speaking, we say that the encryption scheme $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ is *IND-CPA secure* if no PPT algorithm A can win this game with a probability that is noticeably better than 0.5. Intuitively, this means that even if the attacker gets to see all but one of the messages ever encrypted with a specific key, and even if there were a whole lot of such messages that were encrypted, and even if the universe was conspiring to ensure that the messages encrypted just-so-happen to be the messages that maximally benefit the attacker, and even if the attacker knows that the one message it didn't get to see must have been just one of two possibilities, and even if, keeping with the cosmic conspiracy, those two possibilities were maximally beneficial to the attacker, and so on and so forth...that one message that attacker didn't get to see is, for all intents to purposes, still completely hidden from the attacker. It's a mighty strong definition, indeed.

IND-CCA security game

The IND-CCA security game is very similar, but with one (and a half) important difference(s): in addition to asking D for encryptions of arbitrary messages, A can also ask for *deryptions* of arbitrary ciphertexts. The one exception (i.e., the half change) is that D will refuse to decrypt the ciphertext that it returns in the challenge phase; i.e., once A submits the ordered pair $(\mathrm{m}_0, \mathrm{m}_1)$ and receives $\mathrm{c} = \mathrm{Enc}_{\mathrm{k}}(\mathrm{m}_{\mathrm{b}})$, the defender D with thereafter refuse to decrypt this particular c. To be sure, even if A flips a single bit of c, or truncates it, or concatenates some bytes to the end, or any other such change, then D will happily (attempt to) decrypt the result. But if A submits c without any modifications, D will refuse to decrypt.

Clearly this notion of secrecy is stronger than IND-CPA: the A in this game gets to do everything that A was allowed to do in the IND-CPA game, and a whole lot more, in order to inform its guess b'.

Padding oracle assignment

The attack you are expected to implement for the assignment is a stark illustration of just how much stronger IND-CCA security is compared with IND-CPA security. In particular, the encryption I used to generate your challenge ciphertexts is IND-CPA security, and it "attempts" to be IND-CCA secure; however, due to implementation sloppiness it fails to be the latter.

Instead of getting access to an encryption oracle as in IND-CPA, you get a single ciphertext. All you know about it is 1) that it was encrypted using AES-256, 2) what mode of operation was used in the encryption, 3) that underlying plaintext is valid English (indeed, `ciphertext1a` encrypts a quote by Frank Zappa; `ciphertext1b` encrypts a quote from Nineteen Eight-Four; `ciphertext2a` encrypts a quote from A Clockwork Orange; and `ciphertext2b` encrypts a quote from The Trial.)

Likewise, instead of getting access to a decryption oracle as in IND-CCA, you get access to an oracle that decrypt any given ciphertext, and provide feedback about whether the padding of the ciphertext was well-formed and, if so, whether the underlying plaintext was *preceded* by a valid MAC. Two of the oracles (`oracle1a` and `oracle2a`) will explicitly tell you whether the padding and/or MAC were valid, while the other two (`oracles1b` and `oracle2b`) implicitly leak this information via a timing side channel. (Note that I moved the MAC to the front of the message, which is pretty uncommon, so that you would be able to recover the entire plaintext. If I had not done so, it would be impossible to recover the first 32 bytes of the plaintext, as the resulting ciphertext would have been too short to contain a MAC and would be summarily rejected by the oracle.)

Specifically,

- `oracle1a` and `oracle1b` decrypt `ciphertext1a` and `ciphertext1b`, respectively, using AES-256 in CTR mode.
- `oracle2a` and `oracle2b` decrypt `ciphertext2a` and `ciphertext2b`, respectively, using AES-256 in CBC mode.

(Note that the four oracles use different encryption keys, so ciphertexts intended for one oracle will no produce sensible results if submitted to another oracle.)

In all cases, the provided ciphertexts have the following form:

$$[16-\mathrm{byteIV}][\mathrm{Enc}_{\mathrm{k,IV}}(32-\mathrm{byteMAC}\|\mathrm{message}\|\mathrm{padding})].$$

The padding follows the PKCS#5 standard: Padding is in whole bytes, and there is always at least one byte of padding. The value of each added byte is the number of bytes that are added, i.e. N bytes, each of value N are added. The number of bytes added is the smallest number of bytes necessary to pad the plaintext to the *next* multiple of 16 bytes.

What you need to do: Write four programs that interact with the four oracles to recover the four underlying plaintexts by interacting with the oracles. I have provided `/srv/a3/client.cpp` and an associated `Makefile` to get you started, but you are free to write your code in any language of your choosing. All that I care about is that it works. (If you want me to install any interpreters, packages, compilers, etc. on mocha, just ask.)

You will receive 30 marks for writing a program that can recover the plaintext associated with `ciphertext1a`; 10 marks for a program that can recover the plaintext associated with `ciphertext2a`; and 5 marks each for programs that can recover the plaintexts associated with `ciphertext1b` and `ciphertext2b`. For grading purposes, your code will but run against the same oracles *but with different ciphertexts*.

I will post information later in the week about how to submit your code for (auto)grading.
#pin