

Foundations of Network and Computer Security

John Black

Lecture #11

Oct 4th 2005

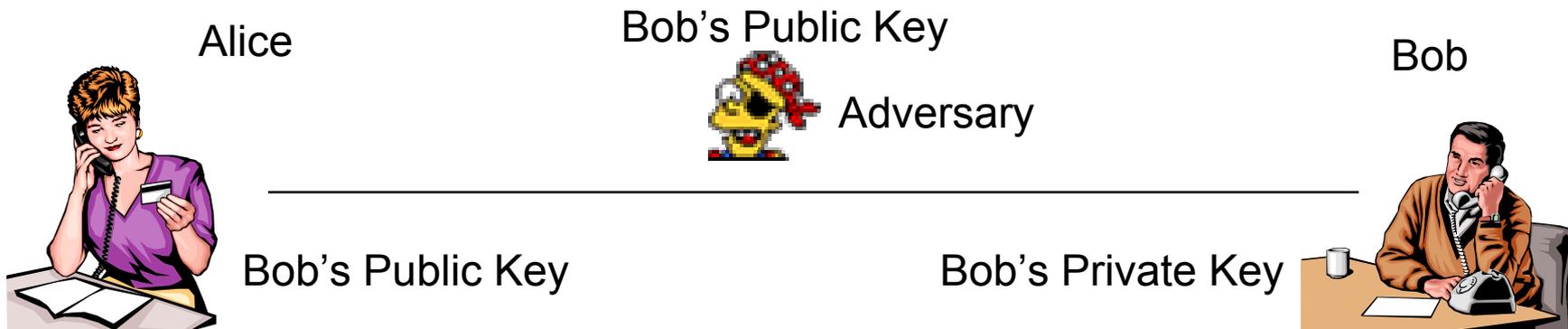
CSCI 6268/TLEN 5831, Fall 2005

Announcements

- Quiz #2 is a week from today, Nov 11th
 - Sorry, another Tuesday quiz!
 - We could have it this Thurs if you'd prefer??
- Project #0 will be assigned next time
 - Due Oct 18th (a Tuesday!)
 - Warm up for the main project in the course
- Don't forget to do the reading (RSA)

Basic RSA Cryptosystem

- Note that after Alice encrypts with pk , she cannot even decrypt what she encrypted
 - Only the holder of sk can decrypt
 - The adversary can have a copy of pk ; we don't care



Key Generation

- Bob generates his keys as follows
 - Choose two large distinct random primes p, q
 - Set $n = pq$ (in $\mathbb{Z} \dots$ no finite groups yet)
 - Compute $\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$
 - Choose some $e \in \mathbb{Z}_{\phi(n)}^*$
 - Compute $d = e^{-1}$ in $\mathbb{Z}_{\phi(n)}^*$
 - Set $pk = (e, n)$ and $sk = (d, n)$
 - Here (e, n) is the ordered pair (e, n) and does not mean gcd

Key Generation Notes

- Note that pk and sk share n
 - Ok, so only d is secret
- Note that d is the inverse in the group $Z_{\phi(n)}^*$ and not in Z_n^*
 - Kind of hard to grasp, but we'll see why
- Note that factoring n would leak d
- And knowing $\phi(n)$ would leak d
 - Bob has no further use for p , q , and $\phi(n)$ so he shouldn't leave them lying around

RSA Encryption

- For any message $M \in \mathbb{Z}_n^*$
 - Alice has $pk = (e, n)$
 - Alice computes $C = M^e \bmod n$
 - That's it
- To decrypt
 - Bob has $sk = (d, n)$
 - He computes $C^d \bmod n = M$
 - We need to prove this

RSA Example

- Let $p = 19$, $q = 23$
 - These aren't large primes, but they're primes!
 - $n = 437$
 - $\phi(n) = 396$
 - Clearly $5 \in \mathbb{Z}_{396}^*$, so set $e=5$
 - Then $d=317$
 - $ed = 5 \times 317 = 1585 = 1 + 4 \times 396$ ✓
 - $pk = (5, 437)$
 - $sk = (396, 437)$

RSA Example (cont)

- Suppose $M = 100$ is Alice's message
 - Ensure $(100, 437) = 1$ ✓
 - Compute $C = 100^5 \bmod 437 = 85$
 - Send 85 to Bob
- Bob receives $C = 85$
 - Computes $85^{317} \bmod 437 = 100$ ✓
- We'll discuss implementation issues later

RSA Proof

- Need to show that for any $M \in \mathbb{Z}_n^*$, $M^{\text{ed}} = M \pmod n$
 - $ed = 1 \pmod{\phi(n)}$ [by def of d]
 - So $ed = k\phi(n) + 1$ [by def of modulus]
 - So working in \mathbb{Z}_n^* , $M^{\text{ed}} = M^{k\phi(n) + 1} = M^{k\phi(n)} M^1 = (M^{\phi(n)})^k M = 1^k M = M$
 - Do you see LaGrange's Theorem there?
- This doesn't say anything about the security of RSA, just that we can decrypt

Security of RSA

- Clearly if we can factor efficiently, RSA breaks
 - It's unknown if breaking RSA implies we can factor
- *Basic* RSA is not good encryption
 - There are problems with using RSA as I've just described; don't do it
 - Use a method like OAEP
 - We won't go into this

Factoring Technology

- Factoring Algorithms
 - Try everything up to \sqrt{n}
 - Good if n is small
 - Sieving
 - Ditto
 - Quadratic Sieve, Elliptic Curves, Pollard's Rho Algorithm
 - Good up to about 40 bits
 - Number Field Sieve
 - State of the Art for large composites

The Number Field Sieve

- Running time is estimated as

$$e^{(1.526+o(1))}(\log n)^{1/3}(\log \log n)^{2/3}$$

- This is super-polynomial, but sub-exponential
 - It's unknown what the complexity of this problem is, but it's thought that it lies between P and NPC, assuming $P \neq NP$

NFS (cont)

- How it works (sort of)
 - The first step is called “sieving” and it can be widely distributed
 - The second step builds and solves a system of equations in a large matrix and must be done on a large computer
 - Massive memory requirements
 - Usually done on a large supercomputer

The Record

- In Dec, 2003, RSA-576 was factored
 - That's 576 bits, 174 decimal digits
 - The next number is RSA-640 which is

31074182404900437213507500358885679300373460228427
27545720161948823206440518081504556346829671723286
78243791627283803341547107310850191954852900733772
4822783525742386454014691736602477652346609

- Anyone delivering the two factors gets an immediate A in the class (and 10,000 USD)

On the Forefront

- Other methods in the offing
 - Bernstein's Integer Factoring Circuits
 - TWIRL and TWINKLE
 - Using lights and mirrors
 - Shamir and Tromer's methods
 - They estimate that factoring a 1024 bit RSA modulus would take 10M USD to build and one year to run
 - Some skepticism has been expressed
 - And the beat goes on...
 - I wonder what the NSA knows

Implementation Notes

- We didn't say anything about how to *implement* RSA
 - What were the hard steps?!
 - Key generation:
 - Two large primes
 - Finding inverses mod $\phi(n)$
 - Encryption
 - Computing $M^e \bmod n$ for large M, e, n
 - All this can be done reasonably efficiently

Implementation Notes (cont)

- Finding inverses
 - Linear time with Euclid's Extended Algorithm
- Modular exponentiation
 - Use repeated squaring and reduce by the modulus to keep things manageable
- Primality Testing
 - Sieve first, use pseudo-prime test, then Rabin-Miller if you want to be sure
 - Primality testing is the slowest part of all this
 - Ever generate keys for PGP, GPG, OpenSSL, etc?

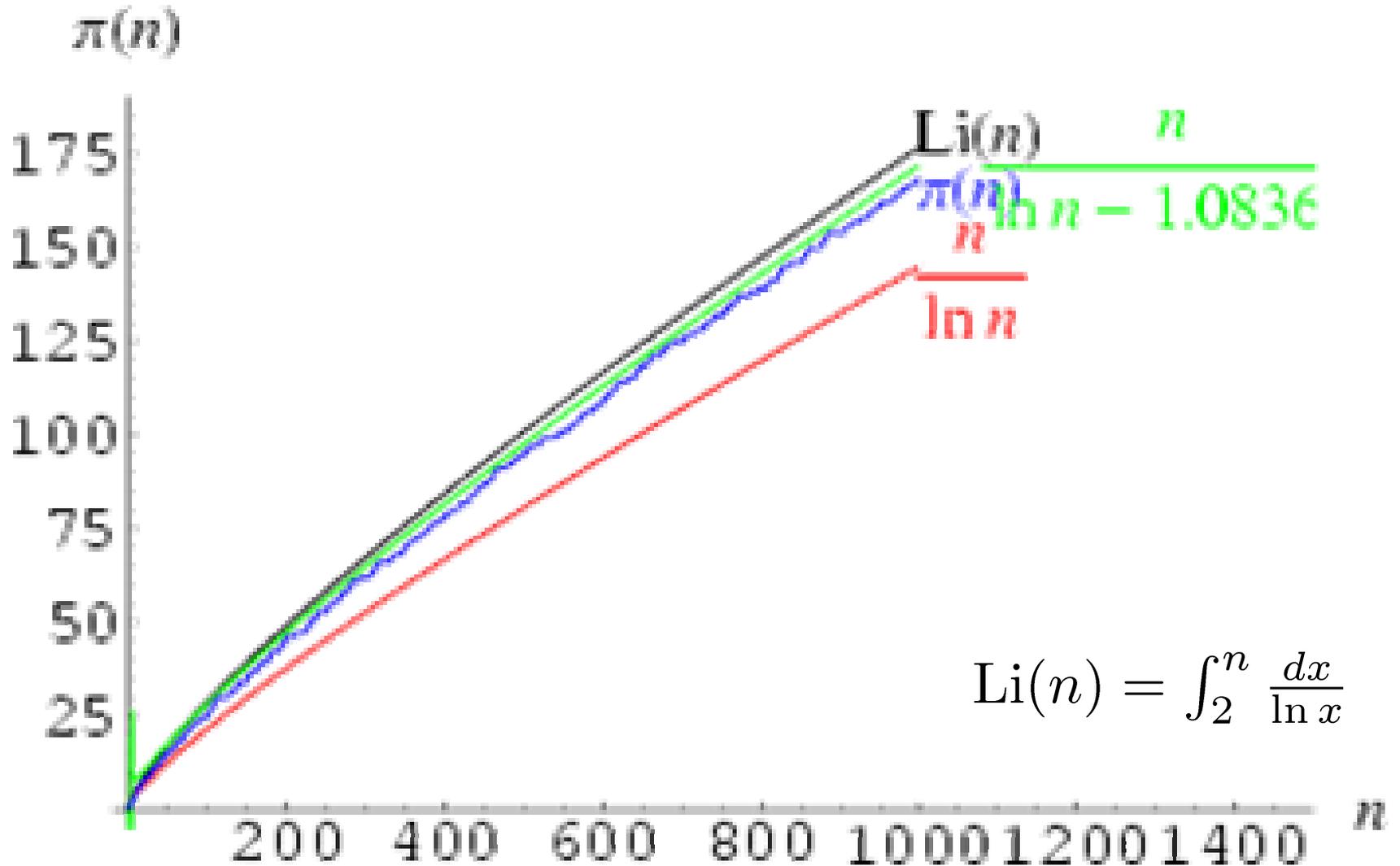
Note on Primality Testing

- Primality testing is *different* from factoring
 - Kind of interesting that we can tell something is composite without being able to actually factor it
- Recent result from IIT trio
 - Recently it was shown that deterministic primality testing could be done in polynomial time
 - Complexity was like $O(n^{12})$, though it's been slightly reduced since then
 - One of our faculty thought this meant RSA was broken!
- Randomized algorithms like Rabin-Miller are far more efficient than the IIT algorithm, so we'll keep using those

Prime Number Theorem

- Are there enough primes?
 - There are plenty, as exhibited by the PNT:
 - PNT: $\pi(n) \sim n/\ln(n)$ where $\pi(n)$ is the number of primes smaller than n
 - In other words, $\lim_{n \rightarrow \infty} \pi(n) \ln(n)/n = 1$
 - What does this mean?
 - Primes get sparser as we go to the right on the number line

$\pi(n)$ versus $n/\ln(n)$



Sample Calculation

- Let's say we're generating an RSA modulus and we need two 512-bit primes
 - This will give us a 1024-bit modulus n
- Let's generate the first prime, p
 - Question: if I start at some random 512-bit odd candidate c , what is the probability that c is prime?
 - Ans: about $1/\ln(c) \approx 1/350$
 - Question: what is the expected number of candidates I have to test before I find a prime, assuming I try every odd starting from c ?
 - Ans: each number has a $1/350$ chance, but I'm testing only odd numbers, so my chance is $1/175$; I therefore expect to test 175 numbers on average before I find a prime
 - Of course I could do more sieving (eliminate multiples of 3, 5, etc)

Digital Signatures

- Digital Signatures are authentication in the asymmetric key model
 - MAC was in the symmetric key model
- Once again, Alice wants to send an authenticated message to Bob
 - This time they don't share a key
 - The security definition is the same
 - ACMA model

We Can Use RSA to Sign

- RSA gives us a signing primitive as well
 - Alice generates her RSA keys
 - Signing key $sk = (d, n)$
 - Verification key $vk = (e, n)$
 - Distributes verification key to the world
 - Keeps signing key private
 - To sign message $M \in \mathbb{Z}_n^*$
 - Alice computes $sig = M^d \bmod n$
 - Alice sends (M, sig) to Bob
 - To verify (M', sig')
 - Bob checks to ensure $M' = sig'^e \bmod n$
 - If not, he rejects
- Once again, don't do this; use PSS or similar

Efficiency

- Why is this inefficient?
 - Signature is same size as message!
 - For MACs, our tag was small... that was good
- Hash-then-sign
 - We normally use a cryptographic hash function on the message, then sign the hash
 - This produces a much smaller signature
 - 2nd-preimage resistance is key here
 - Without 2nd-preimage resistance, forgeries would be possible by attacking the hash function

Let's Sum Up

- Symmetric Key Model
 - Encryption
 - ECB (bad), CBC, CTR
 - All these are modes of operation built on a blockcipher
 - Authentication (MACs)
 - CBC MAC, XCBC, UMAC, HMAC
- Asymmetric Key Model
 - Encryption
 - RSA-OAEP
 - Assumes factoring product of large primes is hard
 - Authentication
 - RSA signatures
 - Usually hash-then-sign

Next Up: SSL

- Next we'll look at how to put all this together to form a network security protocol
- We will use SSL/TLS as our model since it's ubiquitous
- But first, we'll digress to talk about OpenSSL, and our first part of the project (a warm-up)

OpenSSL

- Was SSLeay
- Open Source
- Has everything we've talked about and a lot more
- Most everything can be done on the command line
- Ungainly, awkward, inconsistent
 - Mostly because of history
 - Have fun, it's the only game in town
- <http://www.openssl.org/>

Brief Tutorial

- This is a grad class; you can figure it out from the man page, but...
 - Syntax is
 - % openssl <cmd> <parms>
 - cmd can be ‘enc’, ‘rsautl’, ‘x509’, and more
 - We’ll start with the ‘enc’ command (symmetric encryption)
 - Let’s look at the enc command in more detail

OpenSSL enc command

- `openssl enc -ciphername [-in filename] [-out filename] [-pass arg] [-e] [-d] [-a] [-K key] [-iv IV] [-p] [-P]`
- `-ciphername` can be
 - `des-ecb` (yuk!), `des-cbc` (hmm), `des` (same as `des-cbc`), `des-ede3-cbc`, `des3` (same), `aes-128-cbc`, `bf`, `cast`, `idea`, `rc5`
 - Can omit the 'enc' command if specifying these... kind of hokey
- If you don't specify filenames, reads from and writes to `stdin/stdout`
 - Looks like garbage, of course
- If you don't specify a password on the command line, it prompts you for one
 - Why are command-line passwords bad?
 - You can use environment variables but this is bad too
 - You can point to a file on disk... less bad
- What does the password do?
 - Password is converted to produce IV and blockcipher key

enc (cont)

```
% openssl aes-128-cbc -P
```

```
salt=39A9CF66C733597E  
key=FB7D6E2490318E5CFC113751C10402A4  
iv =6ED946AD35158A2BD3E7B5BAFC9A83EA
```

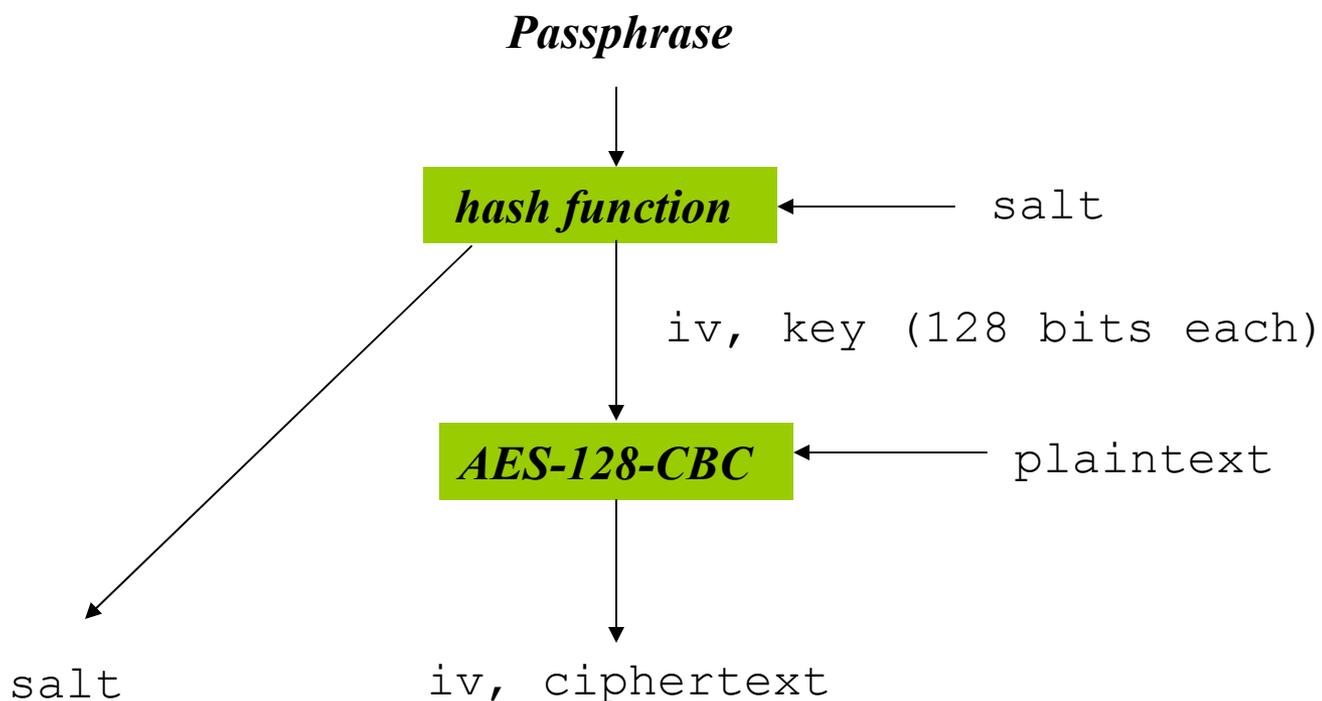
- salt is a random number generated for each encryption in order to make the key and iv different even with the same password
 - Begins to get confusing... didn't we just change the IV before?
 - Use this mode only when deriving a new key for each encryption
 - Eg, when encrypting a file on disk for our own use
 - If key is fixed, we specify it and the iv explicitly

```
% openssl aes-128-cbc -K FB7D6E2490318E5CFC113751C10402A4 -iv  
6ED946AD35158A2BD3E7B5BAFC9A83EA
```

Understanding Passwords vs. a Specified IV and Key

- So there are two modes you can use with enc
 - 1) Specify the key and IV yourself
 - This means YOU are in charge of ensuring the IV doesn't repeat
 - Use a good random number source or
 - Use a counter (which you have to maintain... headache!)
 - 2) Use a passphrase
 - OpenSSL uses randomness for you by generating a salt along with the IV and AES key
 - Passphrases are less secure (more guessable) in general
- Either way, we get non-deterministic encryption

Passphrase-Based enc



Things to think about:

- How to decrypt?
- Is the passphrase safe even though the salt and iv are known?

So How to Encrypt

- Let's encrypt the file 'test'

```
% cat test
```

```
hi there
```

```
% openssl aes-128-cbc -in test
```

```
enter aes-128-cbc encryption password:
```

```
Verifying - enter aes-128-cbc encryption password:
```

```
Salted__mTR&Qi|1K-¿Óàg&5&kE
```

- What's up with the garbage?
 - Of course the AES outputs aren't ASCII!
 - Use `–base64` option

base64

- This is an encoding scheme (not cryptographic)
 - Translates each set of 6 bits into a subset of ASCII which is printable
 - Makes ‘garbage’ binary into printable ASCII
 - Kind of like uuencode
 - Of course this mapping is invertible
 - For encryption we want to do this after we encrypt
 - For decryption, we undo this before we decrypt
 - This is the `-a` flag for ‘enc’ but `-base64` works as well and is preferable

Example: base64

- Let's encrypt file 'test' again, but output readable ciphertext

```
% openssl aes-128-cbc -in test -base64  
enter aes-128-cbc encryption password:  
Verifying - enter aes-128-cbc encryption password:  
U2FsdGVkX1/tdjfZnPrD+mSjBB07InU8Mo4ttzTk8eY=
```

- We'll always use this option when dealing with portability issues
 - Like sending ciphertext over email

Decrypting

- The command to decrypt is once again ‘enc’
 - This makes no sense; get used to it
 - Use the `-d` flag to tell enc to decrypt
 - Let’s decrypt the string

```
U2FsdGVkX1/tdj fZnPrD+mSjBB07InU8Mo4ttzTk8eY=
```

which I’ve placed into a file called ‘test.enc’

```
% openssl enc -d -in test.enc
```

```
U2FsdGVkX18FZEN0ZFZdYvLoqPdpRTgZw2CZIQs6bMQ=
```

Hunh?

- It just gave back the ciphertext?!
 - We didn't specify an encryption algorithm
 - Default is the identity map (get used to it)
 - Let's try again

```
% openssl aes-128-cbc -d -in test.enc  
enter aes-128-cbc decryption password:  
bad magic number
```

- Ok, now what's wrong?

Error messages not useful

- We forgot to undo the `-base64`
 - The error msg didn't tell us that (get used to it)
 - One more try:

```
% openssl aes-128-cbc -d -in test.enc -base64
enter aes-128-cbc decryption password:
hi there
```
 - It was all worth it, right?
 - Now it's your turn

Project #0

- I'll give you a ciphertext, you find the password
 - Password is a three-letter lowercase alpha string
 - Main purpose is to get you to figure out where openssl lives on your computer(s)
 - Don't do it by hand
 - Full description on our web page
 - Due Oct 5th, in class