# Essential Cryptography II

EECS 588: Computer and Network Security
January 14, 2010

## Today's Class

- Cipher Modes
- Building a Secure Channel
- Implementations

(BREAK)

- Diffie-Hellman Key Exchange
- RSA Encryption and Signing
- Establishing Trust

## Cipher Modes

How do we encrypt more than one block?

Some definitions:

- $P_i$ – $i$-th plaintext block
- $C_i$ – $i$-th ciphertext block
- E() – encryption function
- D() – decryption function
- $K$ – encryption key

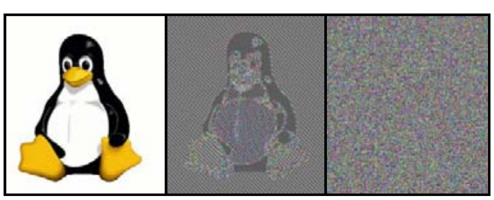
## Cipher Modes: ECB

"Electronic codebook" (ECB) mode

$C_i := E(K, P_i)$ for $i = 1, \ldots, k$

- Most "natural" construction
- Never use ECB

## What's Wrong with ECB?



ECB Other Modes

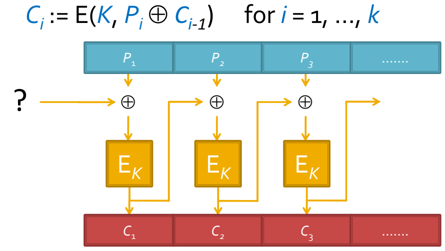
Same plaintext block always encrypts to same ciphertext block.

Don't use ECB mode.

## Cipher Modes: CBC

"Cipher-Block Chaining" (CBC) mode

$C_i := E(K, P_i \oplus C_{i-1})$ for $i = 1, \ldots, k$

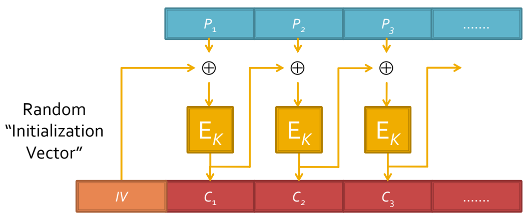


## Cipher Modes: CBC

Is CBC appropriate for encrypting an online movie?

"Cipher-Block Chaining" (CBC) mode

$C_i := E(K, P_i \oplus C_{i-1})$   for $i = 1, \ldots, k$




What if you reuse the IV? Bad.

## Cipher Modes: CTR

"Counter" (CTR) mode

$K_i := E(K, \mathit{Nonce} \,\|\, i)$   for $i = 1, \ldots, k$

$C_i := P_i \oplus K_i$

- Stream cipher construction – like OTP
- Plaintext never passes through *E*
- Don't need to pad the message
- Allows parallelization and seeking
- Never reuse same *K*+*Nonce* (like OTP)

## Building a Secure Channel

Session Key *K*
Never reuse

Session Key *K*
Never reuse

// Initialization (Both Parties)

Separate keys for each function

*KeySendEnc* := HMAC-SHA256(*K*, "Enc A-to-B")
*KeyRecvEnc* := HMAC-SHA256(*K*, "Enc B-to-A")
*KeySendAuth* := HMAC-SHA256(*K*, "Auth A-to-B")
*KeyRecvAuth* := HMAC-SHA256(*K*, "Auth B-to-A")
if Bob then
  swap(*KeySendEnc*, *KeyRecvEnc*)
  swap(*KeySendAuth*, *KeyRecvAuth*)
*MsgCntSend* := 0
*MsgCntRecv* := 0

## Building a Secure Channel

// Sender
*MsgCntSend* := *MsgCntSend* + 1
*i* := *MsgCntSend*
*a* := HMAC-SHA256(*KeySendAuth*,
  *i* || len(*m*) || *m*)
*t* := *a* || *m*
*K* := *KeySendEnc*
// to length of *t* :
*MsgKey* := E(*K*, *i* || 0) || E(*K*, *i* || 1) || …
Transmit(*i* || (*t* ⊕ *MsgKey*))

// Receiver
i || t := Receive()
*K* := *KeyRecvEnc*
// to length of *t* :
*MsgKey* := E(*K*, *i* || 0) || E(*K*, *i* || 1) || …
*a* || *m* := *t* ⊕ *MsgKey*
*a'* := HMAC-SHA256(*KeyRecvAuth*,
  *i* || len(*m*) || *m*)
Check(*a'* == *a*)
Check(*i* > *MsgCntRecv*)
*MsgCntRecv* := *i*

## Encrypt First or Auth First?

HMAC(E(*msg*)) or E(HMAC(*msg*)) ?

## Implementations: OpenSSL

- Try not to implement crypto functions. Use OpenSSL libraries if possible.
  - Open source implementation
  - SSL protocol plus general crypto functions
  - Very fast hand-tunes assembly language

## OpenSSL on the Command Line

- Hashing (a.k.a. "message digest")
```
$ openssl dgst -sha256 myfile
```
- Encryption and decryption
```
$ openssl enc -aes-256-cbc \
         -in myfile -out myfile.enc
$ openssl enc -d -aes-256-cbc \
         -in myfile.enc -out myfile
```
- Performance tests
```
$ openssl speed sha
$ openssl speed aes
```

See man page

## OpenSSL in C – Authentication

```
#include <openssl/hmac.h>
#include <openssl/sha.h>
#include <openssl/evp.h>

  unsigned char mac[SHA256_DIGEST_LENGTH];
  mac = HMAC(
     EVP_sha256(), // use SHA-256 hash function
     (unsigned char*) key,
     (unsigned long ) keyNumBytes,
     (unsigned char*) data,
     (unsigned long ) dataNumBytes,
     NULL, NULL
  );
```

## OpenSSL in C – Encryption

```
#include <openssl/evp.h>

// 256-bit AES in CBC mode with padding
void AesEncrypt(unsigned char key[32], unsigned char iv[16])
{
   unsigned char inData[16], outData[16];
   Int inLen, outLen;
   EVP_CIPHER_CTX ctx;

   EVP_CIPHER_CTX_init(&ctx);
   EVP_EncryptInit_ex(&ctx, EVP_aes_256_cbc(), NULL,
       (unsigned char *)key, (unsigned char *)iv);

   while ((inLen = fread(inData, 1, 16, stdin)) > 0) {
       EVP_EncryptUpdate(&ctx, outData, &outLen, inData, inLen);
       fwrite(outData, 1, outLen, stdout);
   }

   EVP_EncryptFinal_ex(&ctx, outData, &outLen);
   fwrite(outData, 1, outLen, stdout);
   EVP_CIPHER_CTX_cleanup(&ctx);  // zeroize the key
}
```

## Try OpenSSL at Home

- Install OpenSSL or use try it on a cluster
  - Sign and encrypt a message
  - Compare the speed of various functions
  - Think… How does the AES implementation compare to the speed of your Internet connection? You hard disk? You RAM?
- Use C, Python, or Perl and the OpenSSL library to implement our secure message passing protocol

## Summary of Practical Advice

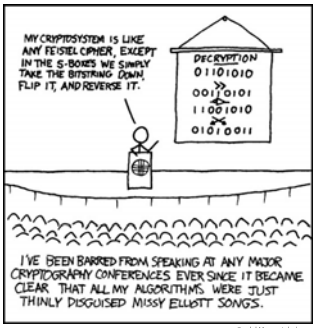
- Don't use MD5; avoid hash function pitfalls
- Don't use DES; avoid ECB mode
- Don't use `rand()` and its ilk

- For a hash/MAC, use **HMAC-SHA256**
- For a block cipher, use **AES-256**
- For randomness, use the **OS's CPRNG**
- For implementations, use **OpenSSL**

## Related Research Problems

- *Cryptanalysis:* Ongoing work to break crypto functions… rapid progress on hash collisions
- *Cryptographic function design:* We desperately need better hash functions… NIST competition now to replace SHA
- *Attacks:* Only beginning to understand implications of MD5 breaks – likely enables many major attacks

## 5 Minute Break




Randall Munroe / xkcd.com

## Public-Key Cryptography

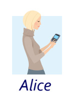
- **Problem:** With symmetric ciphers, every sender-receiver pair must share a secret key
- **Question:** What if we could use *different keys* for encryption and decryption?

## Diffie-Hellman Key Exchange

- Whitfield Diffie and Martin Hellman, 1976


- Lets Alice and Bob establish a shared secret even if Eve is listening in

## Diffie-Hellman Key Exchange




Agree on a large prime $p$ and primitive element $g$ that generates the group $Z_p^*$ ($p$, $g$ can be public)


Alice: Chooses random $x < p$ (secret)

$a := g^x \bmod p$ →

Bob: Chooses random $y < p$ (secret)

← $b := g^y \bmod p$

Alice: Calculates $k := b^x \bmod p$

Bob: Calculates $k := a^y \bmod p$

Why this works:
$b^x = (g^y)^x = (g^x)^y = a^y \pmod{p}$

## Difficulty?

- Diffie-Hellman (DH) problem:
  Compute $g^{xy}$ given $g^x$ and $g^y$ (mod $p$)
- Best *known* approach: Compute $x$ from $g^x$
  - Called the discrete logarithm (DL) problem
  - No *known* efficient algorithm
- Modular exponentiation believed to be a one-way function
  - Easy to compute
  - Hard to invert

## Attacking Diffie-Hellman


Alice: Chooses random $x < p$ — $g^x$ →

Mallory: Chooses random $v < p$ — $g^v$ →

Bob: Chooses random $y < p$ — ← $g^y$

Mallory: Chooses random $w < p$ — ← $g^w$

Alice: $k := (g^w)^x$

Mallory: $k := (g^w)^x$, $k' := (g^v)^y$

Bob: $k' := (g^v)^y$

## RSA

- Ron Rivest, Adi Shamir, Len Adleman (1977)
- Used for encryption and signatures
- Based on a trapdoor function
  - Easy to compute
  - Hard to invert without special information
- Based on apparent difficulty of factoring large numbers

## RSA in One Slide

| | |
|---|---|
| $p$, $q$ | large random primes |
| $n := pq$ | modulus |
| $t := (p-1)(q-1)$ | ensures $x^t = 1 \pmod{n}$ |
| $e :=$ [small odd value] | public exponent |
| $d := 1/e \bmod t$ | private exponent |

Public key: $(n, e)$
Private key: any of $p$, $q$, $t$, $d$

Encryption: $c := m^e \bmod n$
Decryption: $m := c^d \bmod n$

*Why?* $(m^e)^d = m^{ed} = m^{kt+1} = (m^t)^k m = 1^k m = m \pmod{n}$

## RSA for Encryption

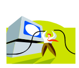
- Publish: $(n, e)$
  Store secretly: $d$
- Encryption of $m$
  Choose random $k$ same size as $n$
  $c := k^e \bmod n$
  Send $c$, encrypt $m$ with AES using $k$
- Decryption
  $k := c^d \bmod n$; decrypt $m$ with AES using $k$

Why don't we use RSA to directly encrypt the message?

## RSA for Signatures

- Publish: $(n, e)$
  Store secretly: $d$
- Signing $m$
  Seed a CPRNG with $m$ and calculate pseudorandom string $s$ same size as $n$
  $\sigma := s^d \bmod n$
- Verifying a signature on $m$
  Recalculate $s$ from $m$
  Check $s = \sigma^e \bmod n$

## D-H with Authentication




Alice: Chooses random $x < p$ (secret)

Alice → Bob: $a := g^x \bmod p$ $\text{Sign}_{Alice}(a)$

Bob: Chooses random $y < p$ (secret)

Bob → Alice: $b := g^y \bmod p$ $\text{Sign}_{Bob}(a,b)$

Alice: Verifies signature
Calculates $k := b^x \bmod p$

Bob: Verifies signature
Calculates $k := a^y \bmod p$

## Establishing Trust

How do Alice and Bob learn each others' signature verification keys?

- Web of Trust
  - Transitive trust among associates (e.g. PGP)
- Public Key Infrastructure (PKI)
  - Trusted third-party Certificate Authority (CA) binds keys-identities (e.g. SSL)

## Tuesday: Crypto Attacks (I)

- Optional Background Reading
  - Introducing SSL and Certificates using SSLeay
    Hirsch. WWW Journal, Summer 1997.
- Required Reading – **Responses Due Before Class**
  - MD5 To Be Considered Harmful Someday
    Kaminsky. 2004.
  - MD5 Considered Harmful Today
    Sotirov, Stevens, Appelbaum, Lenstra, Molnar, Osvik, and Weger. CCC 2008.

## Paper Responses

- Brief written response to each required paper (should be < 350 words/paper):
  (1) state the problem the paper is trying to solve
  (2) summarize its main contributions
  (3) evaluate its strengths and weaknesses*
  (4) suggest at least two interesting open problems on related topics*
  (5) tell me if anything was too difficult to understand*
- Due by email before class
  - Put "[reading588]" in subject line

## Security Reading Group

- Thursdays 12-1:30pm
- Read 1 paper, get free lunch
- Get on the mailing list, http://wiki.eecs.umich.edu/secrit/