

Common Developer Crypto Mistakes (with illustrations in Java)

Kevin W. Wall
Stir Trek, 2016/05/06

Copyright © 2016 – Kevin W. Wall – All Rights Reserved.
Released under Creative Commons Attribution-Noncommercial-
Share Alike 3.0 United States License
as specified at
<http://creativecommons.org/licenses/by-nc-sa/3.0/us/>

Obligatory “It's all about me” page



- 35+ years developer experience, 15+ yrs security experience
 - 17 yrs at (now Nokia) Bell Labs; left as DMTS
 - 3.5 yrs as independent contractor (C++ & Java)
 - 14 years AppSec & InfoSec experience at CenturyLink / Qwest
- Currently: Information Security Engineer at Wells Fargo on Secure Code Review team (3 yrs)
- OWASP ESAPI for Java
 - Project co-leader
 - Cryptography developer (since Aug 2009)
- New OWASP Dev Guide – Crypto chapter
- Blog: <http://off-the-wall-security.blogspot.com/>
- G+: <https://plus.google.com/+KevinWWall/>
- Email: <kevin.w.wall@gmail.com>
- Twitter: @KevinWWall
- CISSP, GIAC Web Application Defender (GWEB)



How I remain calm while public speaking



What I will cover

- Dev good news / bad news
- Mistakes in using the following:
 - Pseudo random number generators
 - Secure hashes
 - Symmetric encryption
 - Asymmetric encryption
- Miscellaneous topics (time permitting)
 - TLS issues
 - Key management
 - Transparent DB Encryption

Good News / Bad News

- **Good news:**
 - Devs no longer designing their own crypto
 - Devs rarely implementing standard algorithms
- **Bad news:**
 - ✗ Dev “expertise” from copy-&-paste from Stack Exchange, etc., so still get things wrong.
 - ✗ Confidentiality vs. authenticity
 - ✗ Confusion of cipher modes, padding schemes
 - ✗ Broken crypto for legacy applications
 - ✗ Even experts still get things wrong (e.g., OpenSSL, GPG, etc.).

Pseudo Random Number Generators (PRNG)

PRNG Weaknesses

- Having a good source of (pseudo) randomness is essential to good cryptography.
 - Poor randomness ==> broken crypto
 - Cryptographers demand a “cryptographically secure” PRNG (CSRNG)
 - `java.util.Random` is *not* a CSRNG
 - `java.security.SecureRandom` is a CSRNG
 - CSRNG must have unpredictable seed
 - Seed entropy must equal (and should exceed) the internal state of the CSRNG

PRNG Weaknesses: What to look for

- Using `java.util.Random` for *anything* related to crypto—this would include keys, IVs, nonces, etc.
- Seeding any CSRNG with insufficient entropy
 - If you initially require N-bits of randomness, then the entropy pool should have *at least* N-bits of randomness.
 - Generally not a problem with the default Oracle/Sun implementation of `SecureRandom` and `SHA1PRNG`.
 - Default `SecureRandom` CTOR uses `/dev/urandom` when available **BUT** may be a problem if lots of randomness is required at boot time or if no `/dev/urandom` or `/dev/random`

Example of correct use / seeding of SecureRandom

```
SecureRandom csrng =  
    SecureRandom.getInstance("SHA1PRNG",  
                             "BC");  
csrng.setSeed(  
    csrng.generateSeed( 160/8 )  
);
```

For JDK 8 and later, consider using
SecureRandom.getInstanceStrong()
instead of SecureRandom.getInstance().

Secure Cryptographic Hashing

Secure Hashing Weaknesses:

What to look for (1/4)

- Use of completely broken algorithms: MD2, MD4, MD5 or algorithms that are not true message digests such as CRCs.
- Use of mostly broken algorithms: SHA1 (may be okay for legacy use for backward compatibility and some CSRNG cases).

Secure Hashing Weaknesses:

What to look for (2/4)

- If concerned about *local* attacks...
 - Time-dependent comparison of hashes
 - E.g., Bad: `String.equals()` or `Arrays.equals()`
 - `MessageDigest.isEqual()` is okay *after* JDK 1.6.0_17
- Calling `MessageDigest.digest(byte[])` or `update(byte[])` methods on unbounded input under adversary's control. (DoS attack)

Secure Hashing Weaknesses:

What to look for (3/4)

- Misusing secure hash (MessageDigest) for message authentication codes (MAC):

- MAC is a *keyed* hash, where the key is a secret key generally shared out-of-band.

- Incorrect, naïve use:

$\text{MAC}(\text{key}, \text{message}) := H(\text{key} || \text{message})$

Where '||' is bitwise concatenation.

Problem: Susceptible to “length extension attacks”.

- Correct use: Use an HMAC (RFC 2104)...

```
Mac hmac = Mac.getInstance("HmacSHA256", "SunJCE");  
hmac.init(key);
```

Secure Hashing Weaknesses:

What to look for (4/4)

- Misusing a secure hash to mask data where enumeration of all or most of the input space is feasible.
 - E.g., Use SHA-256(SSN) to store as key in database or to track in log file.
 - Problem: If adversary can observe hashes, she can enumerate SHA-256 hashes of all possible SSNs and compare these to stored hashes.

Is use of MD5 *ever* okay?

- Best collision attack against it is now about $O(2^{24.1})$, which takes at most 5 or 6 seconds on a modern desktop / laptop.
- But...okay in following cases:
 - Used as a PRNG when we only need something that is more or less unique and unpredictable; example IV generation used with CBC for symmetric ciphers.
 - Used as an HMAC construct as defined in RFC 2104
 - Bellare, Canetti & Krawczyk (1996): Proved HMAC security doesn't require that the underlying hash function be collision resistant, but only that it acts as a pseudo-random function.

Symmetric Encryption

Symmetric Encryption Weaknesses

- Inappropriate cipher algorithms
 - You aren't still using RC4, are you?
- Insufficient key size: ≥ 128 bits
 - Java: DESede defaults to 2-key TDES (112-bit) unless the JCE Unlimited Strength Jurisdiction Policy files are installed.
- “ASCII” generated keys
- Inappropriate use of cipher modes
 - Related: IV abuses
- Assuming confidentiality implies data integrity.

ASCII Keys

- Keys generated from passwords or passphrases. E.g.,

```
String key = "#s0meSeCR3tK3y!!"; // Or from prop  
SecretKeySpec skey =  
    new SecretKeySpec( key.getBytes(), "AES");  
Cipher cipher =  
    Cipher.getInstance("AES/CBC/PKCS5Padding");  
cipher.init(Cipher.ENCRYPT_MODE, skey);  
...
```

Inappropriate use of cipher modes

Question: `Cipher.getInstance("AES")`

... what's the default cipher mode?

- Block modes and stream modes
 - Block modes: ECB and CBC
 - Stream modes: pretty much everything else
- All modes except for ECB require an IV.
- Streaming modes: Must **not** reuse the same key / IV pair... ***EVER!***
- Streaming modes do not require padding.

Inappropriate use of cipher modes: ECB

- ECB is the raw application of the cipher algorithm.
- Reasons why it is the most commonly misused:
 - First (and sometimes only) example in textbooks
 - Simplest to implement (no need to bother with IVs)
- Weaknesses:
 - Same plaintext blocks always encrypt to same ciphertext
 - Block replay attacks are possible

What's Wrong with ECB Mode?

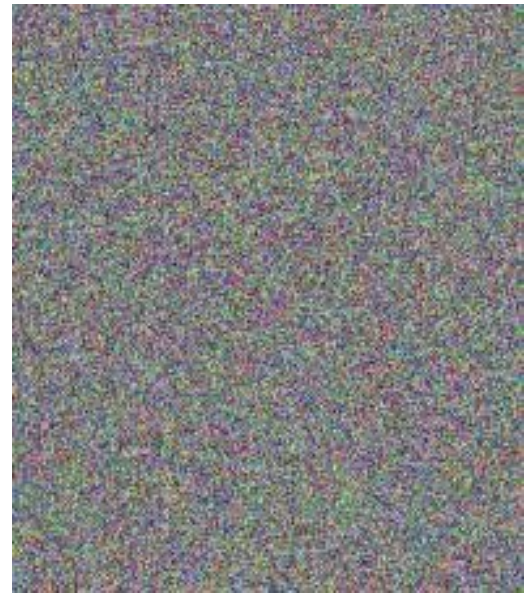
**Original
Tux image**



**Tux image
encrypted
with ECB
mode**



**Tux image
encrypted
with any
other cipher
mode**



From: http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

ECB: Block Replay Attack (1/6)

- Adversary can modify encrypted message without knowing the key or even encryption algorithm.
 - Can mangle message beyond recognition.
 - Remove, duplicate, and/or interchange blocks
 - Can usurp meaning of message if structure known. Consider the following scenario...

ECB: Block Replay Attack (2/6)

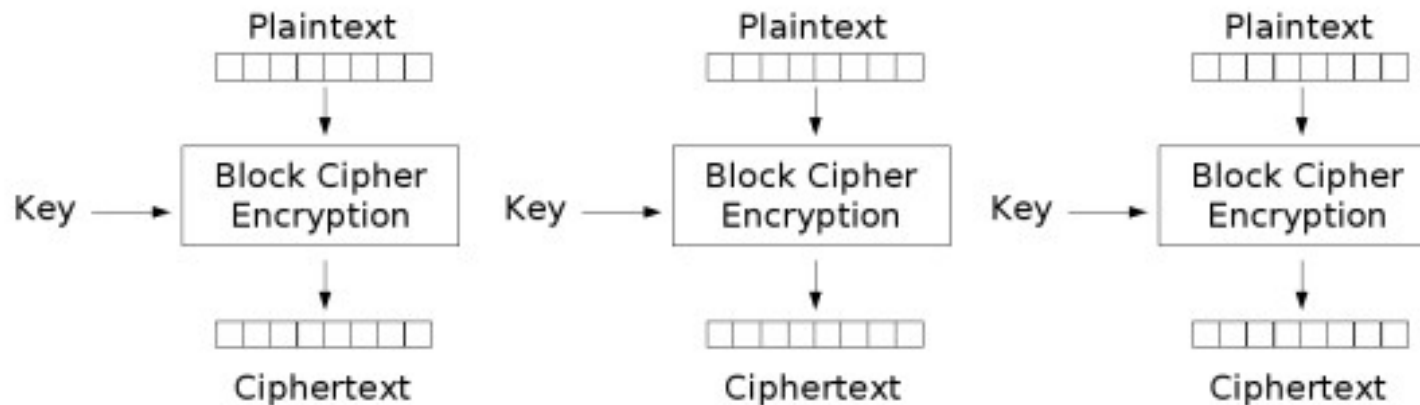
[Example from Schneier, *Applied Cryptography*]*

- Assume 8-byte encryption block size.
- Money transfer system to move \$ btw banks
- Assume bank's standard message format is:

Bank 1: Sending	1.5 blocks
Bank 2: Receiving	1.5 blocks
Depositor's Name	6 blocks
Depositor's Acct #	2 blocks
Deposit Amount	1 block

* First discussed by C. Campell, *IEEE Computer*, 1978

ECB: Block Replay Attack (3/6)



Electronic Codebook (ECB) mode encryption

Each block is encrypted (and decrypted) independently

ECB: Block Replay Attack (4/6)

- Mallory is MITM agent, listening to comm channel between Bank of Alice and Bank of Bob.
- Mallory sets up accounts in both banks and deposits seed money in Bank of Alice.
- Mallory transfers some fixed amount of the seed money to Bank of Bob and records transaction.
- Repeats later, and looks for identical blocks; eventually isolates acct transfer authorization.

ECB: Block Replay Attack (5/6)

- Mallory can now insert those message blocks into communication channel at will. Each time, that fixed amount will be deposited in Mallory's account at the Bank of Bob.
- Two banks will notice by close of business when accts are reconciled. By that time, Mallory has already skipped town.

ECB: Block Replay Attack (6/6)

- Can *not* be defeated by simply prepending date/time stamp to bank transfer authorization message. Mallory can replay individual blocks that lie on whole block boundaries (e.g., in this case the Depositor's Name and account #).
- *Can* be defeated by adding secure *keyed* hash to entire message (or using another cipher mode).

ECB: What to look for

- No cipher mode specified at all. E.g.,
`Cipher cipher = Cipher.getInstance("AES");`
In Java, this is the same as:
`Cipher cipher =
 Cipher.getInstance("AES/ECB/PKCS5Padding");`
- No evidence that an IV is used
 - In Java, look for *absence* of both
 `IVParameterSpec` and `Cipher.getIV()`
 - Check lengths of resulting encryption
 - Generally IV is prepended to the raw ciphertext.
(Exception might be where IV is fixed (bad) or
determined algorithmically; discussed later.)

ECB: Is it ever okay?

- Yes, when:
 - Encrypting plaintext with a less than 1 cipher block and ciphertext attacks not feasible:
 - Blowfish and DES (and hence DESede) block size: 64 bits
 - AES block size (and most other AES candidates): 128 bits
 - OR when encrypting *random* data
 - E.g., nonces, session IDs, *random* secret keys; maybe passwords if strong passwords enforced (LOL!).
- AND padding is used when appropriate (random data)
- AND block replay attacks are not an issue
- OR, using it for asymmetric encryption (only applicable mode!)

If use of ECB *seems* okay...

- Make sure it is not used in a scenario where a block replay attack is possible.
- Ask yourself:
 - Are multiple blocks of ciphertext encrypted with ECB used?
 - Are these multiple ciphertext blocks exposed to an “adversary”?
 - Will block re-ordering ever fail to be detected in any cases? (I.e., are there cases where data integrity not always ensured?)
- If answer to these is “yes” for all questions, block replay is probably possible.

Key / IV reuse in streaming mode (1/9)

- Stream ciphers and block ciphers operating in streaming modes create a cipher bit stream that is XOR'd with the plaintext stream.
- For a given key / IV pair, the same cipher bit stream is generated each time. Let's call this cipher bit stream, $C(K, IV)$.
- Let the encryption function for such a streaming mode be designated as $E(K, IV, msg)$.
 - Then $E(K, IV, msg) = msg \text{ XOR } C(K, IV)$

Key / IV reuse in streaming mode (2/9)

- Let's see what happens if we encrypt 2 different plaintext messages, A and B, this way

$$E(K, IV, A) = A \text{ XOR } C(K, IV)$$

$$E(K, IV, B) = B \text{ XOR } C(K, IV)$$

- If an adversary intercepted both of these ciphertext results, they can compute the XOR of them, which is

$$E(K, IV, A) \text{ XOR } E(K, IV, B) =$$

$$A \text{ XOR } C(K, IV) \text{ XOR } B \text{ XOR } C(K, IV)$$

which, since XOR is commutative, is:

$$A \text{ XOR } B \text{ XOR } C(K, IV) \text{ XOR } C(K, IV) = A \text{ XOR } B$$

That is, the XOR of the 2 plaintext messages, A and B.

Key / IV reuse in streaming mode (3/9)

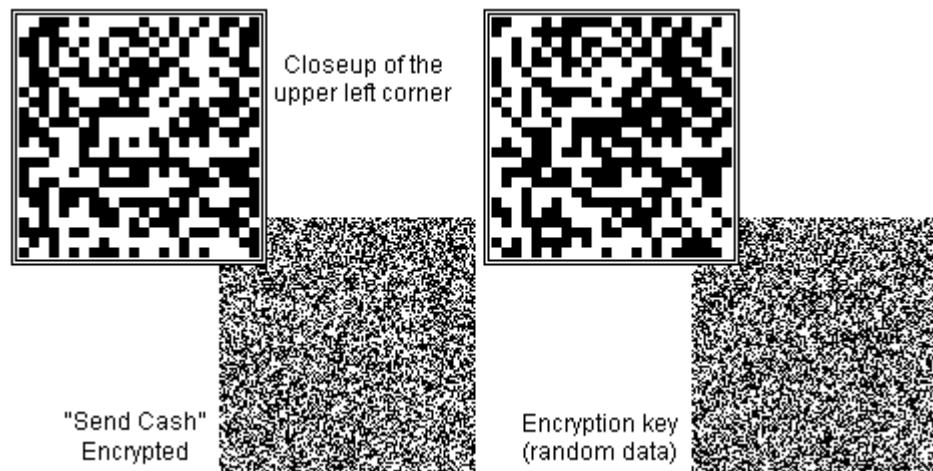
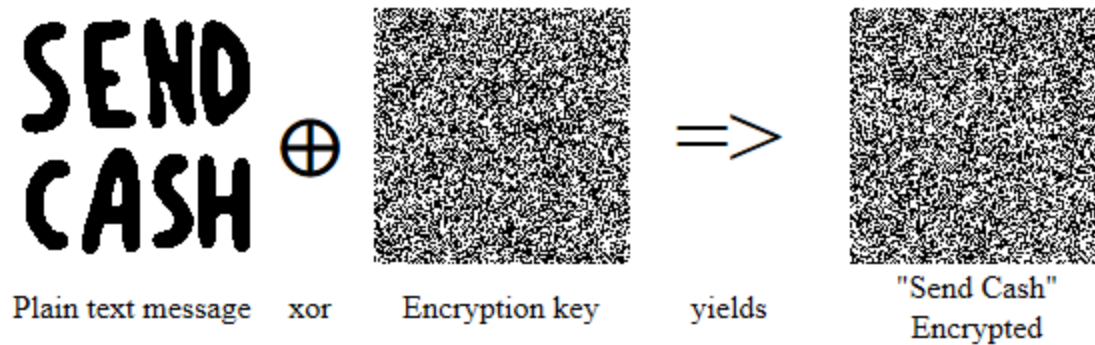
- So what do we do with the XOR of 2 plaintext messages, A and B?
- If messages A and B are both written in some normal language (or character set, like ASCII), we can make that as a guess and use frequency distribution of some anticipated language (or format, such as C#s, etc.) and guess likely plaintext bits (characters). If the result resembles something intelligible (e.g., ASCII letter), guess was probably right.
- Modest computers can crack this in matter of few minutes for modest length messages.

Key / IV reuse in streaming mode (4/9)

- The more ciphertexts created using the same key / IV pair and observed by an adversary, the better.
- Fixed message formats / structures (e.g., knowing you have all numeric fields such as SSN or credit card #) make it even more trivial.
- Eventually, both plaintexts (or shortest part if different lengths) get revealed.

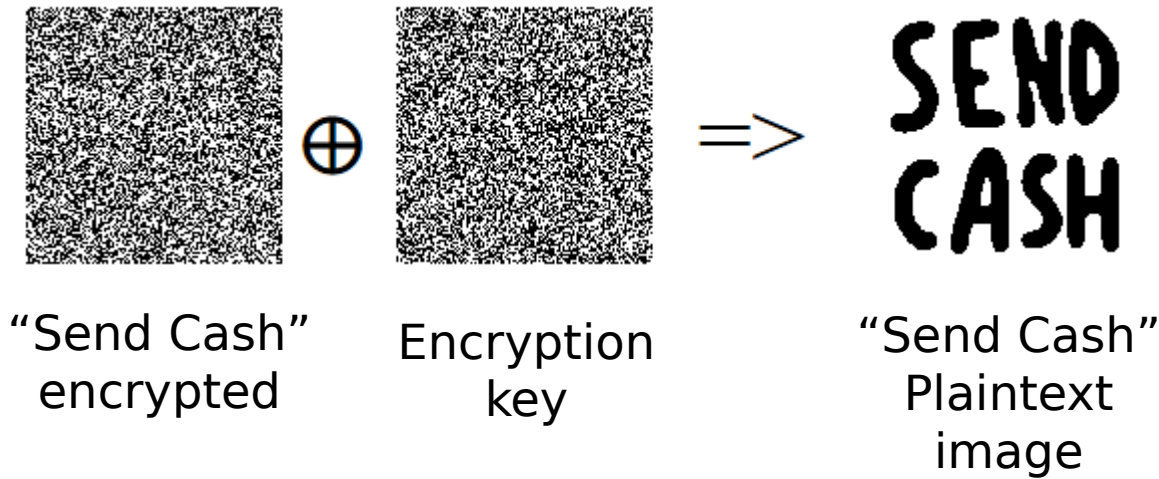
Key / IV reuse in streaming mode (5/9)

Next 4 slides from Dr. Rick Smith, Univ of St. Thomas, MN

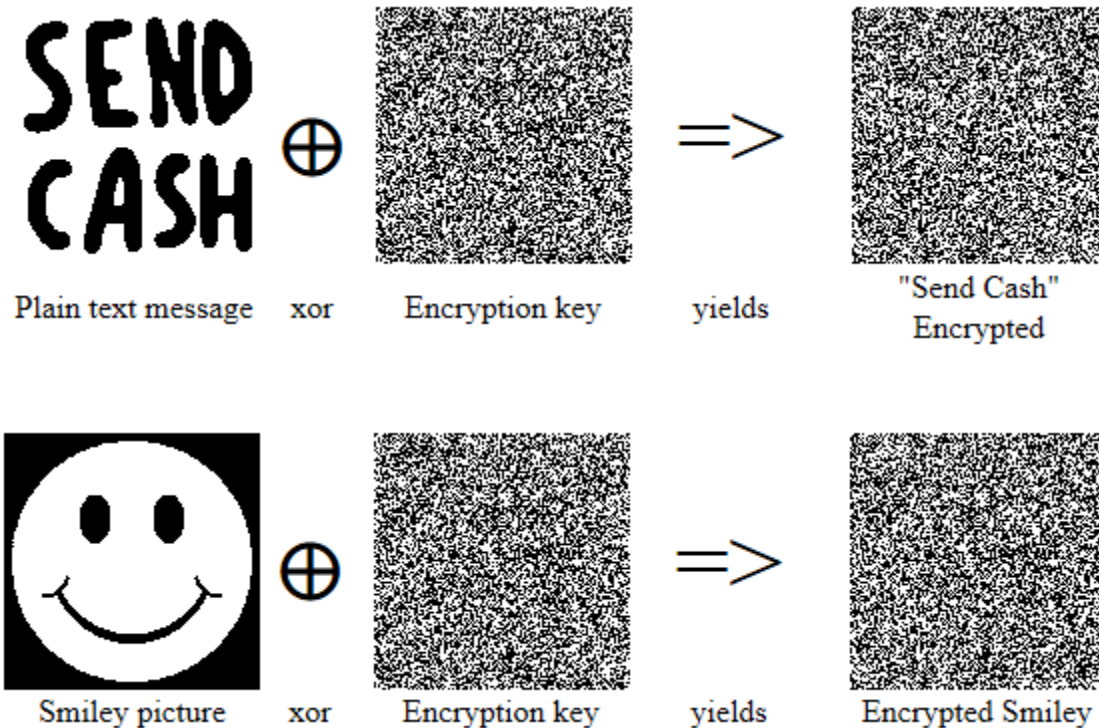


Key / IV reuse in streaming mode (6/9)

- To recover the original message (image), we XOR the encrypted “Send Cash” image with the encryption key again:



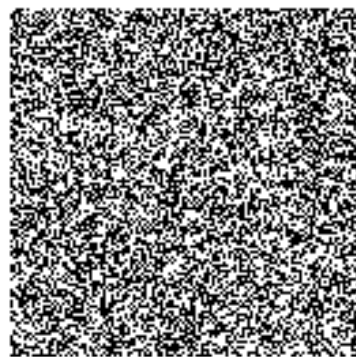
Key / IV reuse in streaming mode (7/9)



Note that we have the **same** encryption key XOR'ing both images.

Key / IV reuse in streaming mode (8/9)

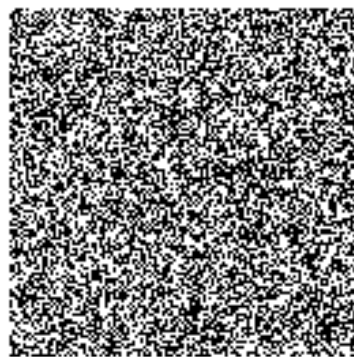
Here's what happens when we XOR the 2 images that both used the same encryption key together:



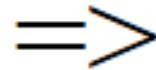
"Send Cash"
Encrypted



xor



Smiley Encrypted



yields



Both messages overlaid

Key / IV reuse in streaming mode (9/9)

- *But wait!* It gets worse. If an application is doing this and an adversary can decrypt a message, they may be able to use a MITM attack to actually *alter* the ciphertext.
- Wikipedia example (Stream_cipher_attack):

$$(C(K) \text{ xor } "\$1000.00") \text{ xor } (" \$1000.00" \text{ xor } "\$9500.00") = C(K) \text{ xor } "\$1000.00" \text{ xor } "\$1000.00" \text{ xor } "\$9500.00" = C(K) \text{ xor } "\$9500.00"$$

Detour: Authenticated Encryption

- Encryption provides confidentiality, not integrity. (Integrity aka authenticity)
- Approaches to authenticated encryption
 - **Encrypt-then-MAC (EtM)**: Encrypt, then apply MAC over IV+ciphertext and append the MAC.
 - **Encrypt-and-MAC (E&M)**: Encrypt the plaintext and append a MAC of the plaintext.
 - **MAC-then-Encrypt (MtE)**: Append a MAC of the plaintext and encrypt them both together.
- Decryption operation applied in reverse order.
- **EtM** built into some cipher modes such as CCM, GCM, EAX, etc.

Horton Principle

- David Wagner and Bruce Schneier
- Relevant when considering what to data to include in a MAC
- Semantic authentication: “Authenticate what is meant, not what is said”
 - Avoid unauthenticated data: either don’t send / rely on it, or include it in the MAC
 - Relevant in message formats and protocols
- E.g., Alice sends: “metadata||IV|| ciphertext||MAC”

Symmetric Encryption

Weaknesses: CBC

- Overall, CBC probably most robust mode when used correctly.
- Use correctly means:
 - Random key and random IV with padding
 - HMAC over the IV+ciphertext applied as “encrypt-then-MAC” approach.
- Common mistakes:
 - Fixed IV or predictable IV (e.g., counter, time, etc.)
 - Failure to MAC correctly (e.g., no MAC at all, encrypt-and-MAC, or MAC-then-encrypt)

Why is AE needed?

- When ciphertext's authenticity is in doubt, certain cryptographic attacks are possible that will either divulge the plaintext (or portions thereof) or possibly even reveal the secret key.
- Padding oracle attack, Serge Vaudenay, 2002
 - Originally discussed as deficiency in IPSec and SSL
 - Dismissed as being impractical until Rizzo and Duong research and POET software in 2010

Symmetric Encryption Weaknesses:

Assuming confidentiality implies data integrity

- Only true if one is using an AE cipher mode such as CCM or GCM (the only 2 AE modes that are NIST approved) or using a correctly implemented EtM approach.
- If confidentiality is not required, better (and faster) to just use an HMAC.
- Look for cases where plaintext is already known to attacker and encryption is used to prevent tampering.

Asymmetric Cryptography: Encryption

Asymmetric Ciphers and Chosen Plaintext Attacks (1/3)

- All asymmetric ciphers are prone to chosen plaintext attacks (CPA).
 - CPA is a cryptanalytic attack where an attacker can choose which plaintext to encrypt and then observe the resulting ciphertext.
 - CPA is always possible with asymmetric ciphers because we assume the algorithm details is known as well as the *public* key.

Asymmetric Ciphers and Chosen Plaintext Attacks (2/3)

- Why might this be a problem?
 - Normally it's not because we usually are encrypting highly unpredictable plaintext that is too large to be enumerated.
 - E.g., symmetric session keys, cryptographic hash values
 - It becomes a problem when the is highly regular or short enough to enumerate all possible values

Asymmetric Ciphers and Chosen Plaintext Attacks (3/3)

- Real-life (bad) example
 - Application uses RSA algorithm to encrypt credit-card #s and store the resulting ciphertexts in application DB.
 - Consider inside attacker with access to DB records (e.g., DBA, developer, tester) as well as the *public* key.
 - Attacker encrypts all possible credit card #s with public key and saves mapping of plaintext / ciphertext pairs.
 - Lookup into application DB records via CC# ciphertext allows discovery of credit card holder as well as revealing plaintext CC#.

Miscellaneous Topics

- Key Management
- Database Encryption
- TLS/SSL issues

Key Management: Re-keying Frequency

- PCI DSS 2.0 and later says that you *must* change symmetric crypto keys *at least* yearly? Is that enough?
- Steve Bellovin says in <http://osdir.com/ml/encryption.general/2005-02/msg00005.html>:
 - For 3DES in CBC mode, re-key at least every $2^{32} * 64$ -bits of plaintext
 - For AES in CBC mode, every $2^{64} * 128$ -bits
 - General: every $2^{N/2} * \text{cipher_block_size}$ bits, where N is key size in bits.

Key Management: Secure Key Storage

So where do you store your keys?

- Ideally: an HSM or a TPM
- FAIL: If hard-coded in source code or put into properties file.
 - Both situations usually under version control!
- Ok: Config file, locked down & controlled by ops staff and unavailable to all others.
- Better: For .NET, DPAPI, WebLogic Encryption Services, Java Key Store
- NEVER put encryption key in *same* file with data that's being encrypted.

Encrypting Data in a DB

Three ways to encrypt data for a database:

1. DB Engine itself does it via (mostly) Transparent Data Encryption (TDE)
2. Done via a proxy; e.g., MIT's CryptDB
3. Done via application code

From application perspective, TDE approach is simplest.

- Transparent to the application.
- Available for Oracle and Microsoft SQL Server
- Probably satisfies “letter of the law” for PCI DSS compliance (not verified).

30k' view of TDE

- Offers encryption at the column, table, and tablespace levels.
- Limited ciphersuite available; e.g., AES & 3DES
- Key management: usually 2 keys involved:
 - DB “master” key – a key encryption key, secured w/ password
 - Table / column / tablespace keys, encrypted by DB master key
- Usually CBC mode used, with usually with same IV for all encryptions
 - Same IV required for deterministic encryption so indexing works as expected
 - “Salt” allows non-deterministic encryption

WIYTM? Why TDE fails

- If any application has that DB table / column open, then any other application with access to that table / column has access to encrypted data!
 - Not problem if data properly partitioned via “views”.
 - Backups, depending on how done, can be in plaintext!
- Usually the data we are encrypting in DB is:
 - Less than 20 bytes
 - Has particular format
 - Limited possible values

Result: Patterns may allow enumeration of values.

SSLSocket & Server AuthN

- SSLSocket (or subclass) created by SSLSocketFactory does not do host name verification or cert pinning by default. Hence, MITM attacks are possible.
 - Must implement your own. 2 approaches:
 - Subclass SSLSocket; see <http://www.velocityreviews.com/forums/t958287-adding-hostname-verification-to-sslsocket.html>
 - Create an SSLContext that does host name verification; see <http://stackoverflow.com/questions/8545685/writing-a-ssl-checker-using-java>

Specifying JCE Providers

- Java has a concept of security providers.
 - Statically added via:
 - JRE: `$JAVA_HOME/lib/security/java.security`
 - JDK: `$JAVA_HOME/jre/lib/security/java.security`
 - Dynamically added via:
 - `Security.addProvider(Provider provider)`
 - `Security.insertProviderAt(Provider provider, int pos)`
 - Various `getInstance()` methods take `Provider` as 2nd arg
- Determined by position; defaults to what is in `java.security`.
- This concept extends to crypto providers

What could possibly go wrong?

```
import org.bouncycastle.jce.provider.*;  
...  
int pos = Security.addProvider(  
    new BouncyCastleProvider() );
```

Static setting in java.security

- Default list of providers ordered by preference:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
```

...

```
security.provider.9=sun.security.smartcardio.SunPCSC
security.provider.10=sun.security.mscapi.SunMSCAPI
security.provider.11=org.bouncycastle.jce.provider.BouncyCastleProvider
```

How about this?

```
import org.bouncycastle.jce.provider.*;  
...  
Security.insertProviderAt(  
    new BouncyCastleProvider(), 1 );
```

Equivalent static setting in java.security

- Equivalent as if we did this:

security.provider.1=org.bouncycastle.jce.provider.BouncyCastleProvider

security.provider.2=sun.security.provider.Sun

security.provider.3=sun.security.rsa.SunRsaSign

security.provider.4=sun.security.ec.SunEC

...

security.provider.10=sun.security.smartcardio.SunPCSC

security.provider.11=sun.security.mscapi.SunMSCAPI

What could possibly go wrong?

- Consider this in `Logger.getLogger()` method in *rogue* copy of `log4j.jar` someone downloaded:

```
...  
Security.insertProviderAt(  
new MyEvilProvider(), 1 );  
...
```

How do we address this?

- Specify the Provider instance as part of the `getInstance()` methods; e.g.,
`Cipher.getInstance("AES/CBC/PKCS5Padding",
 new BouncyCastleProvider());`

OR

- Use a Java Security Manager and restrict what classes may call `Security.addProvider()` and `Security.insertProviderAt()`

What to look for

- Calls to either
 `Security.addProvider()`

OR

`Security.insertProviderAt()`
without the use of a Java Security
Manager (JSM)

Caveat: Java Security Manager is rarely used and if it is used, usage of a properly restrictive security policy is hardly ever set. Also, if the jars are not signed and validated before use, using the JSM matters little.

Additional References

- New OWASP Dev Guide, chapter 11 (Cryptography) [still a work in progress]
 - <https://github.com/OWASP/DevGuide/blob/master/03-Build/0x11-Cryptography.md>
 - And those references therein

Questions?

(Now, over lunch, or email me at
kevin.w.wall@gmail.com)