Analyzing (Java) Source Code for Cryptographic Weaknesses

Kevin W. Wall
Columbus OWASP, 2015/12/03
Obligatory “It's all about me” page

- 35+ years developer experience, 15+ yrs security experience
  - 17 yrs at (now Alcatel-Lucent) 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
- OWASP ESAPI for Java
  - Project co-leader
  - Cryptography developer (since Aug 2009)
- New OWSAP 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
How I remain calm while public speaking

Geeks, in their “encrypted” underwear
(because you never want to seek geeks in their real underwear!)
What I will cover

• Basic crypto rules of thumb
• Weaknesses in using the following:
  – Pseudo random number generators
  – Secure hashes
  – Symmetric encryption
  – Asymmetric cryptography
  • Encryption
  • Digital signatures
• A more complete data flow analysis
• Various crypto gotchas
Major rules of thumb

• Do not design your own cryptographic algorithms...ever (unless you are a professional cryptographer)
  – You will get it wrong
  – Without expert peer review it likely will still be wrong
• Do not even *implement* your own cryptographic algorithms
  – Do you know what test vectors are?
  – Do you know what side-channel attacks are?
• Avoid shiny!
  – Algorithms usually take a few years of peer review to mature.
• But don’t get trapped by obsolete technology
  – Especially a concern for embedded software
• Beware providing too many options
• Schneier and Wagner’s “Horton Principle” (covered with authenticated encryption)
A simple example

```java
public SecretKey generateKey(String alg, int keySizeInBits, SecureRandom prng) {
    KeyGenerator keyGen = KeyGenerator.getInstance(alg);
    keyGen.init(keySizeInBits, prng);
    return keyGen.generateKey();
}

public byte[] encrypt(SecretKey key, String plain) throws EncryptionException {
    byte[] plaintext = plain.getBytes("UTF8");
    int keySize = key.getEncoded().length * 8; // Convert to # bits
    SecretKeySpec encKey = new SecretKeySpec(key, "AES"); // Correct?
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    IvParameterSpec ivSpec =
        new IvParameterSpec( prng.nextBytes( cipher.getBlockSize() ) );
    cipher.init(Cipher.ENCRYPT_MODE, encKey, ivSpec /* , prng */);
    byte[] raw = cipher.doFinal(plaintext.getBytes("UTF8"));
    return bitStringConcat(iv, raw);
```
Data Flow for Symmetric Encryption (Java)

/\dev/\random or
/\dev/\urandom

\?

PRNG seed

(CS)PRNG

(via \KeyGenerator)

SecretKey / SecretKeySpec

Key size, cipher alg (hard-coded or from config file)

Cipher transformation - "cipherAlg/\mode/\padding"

Typically hard-coded or from properties / config file.

Is an AE mode used?

Cipher instance

uses

skey, IV

Is key / IV pair reused (if streaming mode)?

IVParameterSpec

Is SecretKey properly protected (if persisted)?
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 SecurRandom CTOR uses /dev/urandom when available **BUT** may 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 Hashes: Required properties

To be cryptographically useful, a hash function must have the following 3 properties:

• One-way function (AKA, pre-image resistance)
• Weak collision resistance (AKA, 2nd pre-image resistance)
• Strong collision resistance (AKA, collision resistance)
Secure Hashes: Pre-image Resistance

• For essentially all pre-specified outputs, it is *computationally infeasible* to find any input (i.e., the pre-image) that hashes to that output.

That is:

• For a given hash function \( H(x) \) and some output \( y \) such that \( y = H(x) \), it is *computationally infeasible* to find any input (pre-image) \( x \).
Secure Hashes: Weak Collision Resistance

- It is *computationally infeasible* to find a second input that has the same output as any specified input. That is:

- Given input $x$, it is *computationally infeasible* to find an $x' \neq x$ such that $H(x) = H(x')$. 
Secure Hashes: Strong Collision Resistance

• It is *computationally infeasible* to find any *two* distinct inputs $x$ and $x'$ that hash to the same output.

• That is:
  It’s *computationally infeasible* to find any $x$ and $x'$ such that $H(x) = H(x')$.
  – Note: Unlike weak collision resistance, here there is a free choice of an adversary selecting *both* inputs.
Secure Hashing Weaknesses

• Recall the “computationally infeasible” in the preceding slides.
• Some algorithms are “broken”.
  – What does that mean?
    • Any of these 3 conditions are violated in a work factor better than a brute-force attack. Usually only focuses on [strong] collision resistance.
  – Degrees of brokenness
    • If a hash is n-bits, the best theoretical brute-force collision attack is $O(2^{n/2})$ (i.e., the “birthday attack”). If collisions can be found in less effort than this, the algorithm is technically considered “broken” even though the attack (currently) may be completely impractical.
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).
Secure Hashing Weaknesses: What to look for (1/3)

• 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 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:

    \[
    \text{MAC(key, message)} := H(\text{key} \mathbin{||} \text{message})
    \]

    Where ‘||’ is bitwise concatenation.

    Problem: Susceptible to “length extension attacks”.

  – Correct use: Use an HMAC.
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 PC.
• 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
• Failure to apply proper padding.
• 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 cipher algorithms

• Check your corporate InfoSec policies
• Stick with NIST approved algorithms:
  – FIPS 140-2 Annex A:
• Quick spot check: symmetric cipher that is not AES (or maybe TDES for legacy applications) should be considered potentially suspect.
Failure to apply proper padding

• What is padding and why is it needed?
• What happens if padding is omitted?
• Popular padding schemes
What is padding and why is it needed? (1/2)

• Why is padding needed?
  – Because some cipher modes (notably ECB and CBC) are “block mode” operations and can only operate on a \textit{full} cipher block at a time.

• What is padding?
  – It’s additional data added ([almost?] always appended) to the \texttt{plaintext} before encryption and removed immediately after decryption.
What is padding and why is it needed? (2/2)

• When padding is specified it is always applied.
• Padding increases overhead of ciphertext by 1 cipher block size (which is significant when encrypting short plaintext messages).
What happens if padding is omitted?

• That’s the $64,000 question.
• The answer seems to be implementation specific. Possible approaches:
  – Refuse to encrypt plaintext not an integral multiple of the cipher’s block size (this is the JCE approach in Java, where an IllegalBlockSizeException will be thrown).
  – Silently do some kludgy internal implementation-specific padding.
  – Silently truncate excessive plaintext and do not encrypt it, but leave it just as plaintext.
Popular padding schemes

• For symmetric ciphers:
  – PKCS#7 & PKCS#5 (.NET uses PKCS7, Java uses PKCS5; technically PKCS5 is only defined for ciphers whose block size is 64 bits so Java is wrong!)
  – ISO 10126 (used in W3C’s XML Encryption)

• For asymmetric ciphers:
  – PKCS#1 padding
  – OAEP (Optimal Asymmetric Encryption Padding)
    • In Java: OAEPPWith<digest>AndMGF1Padding, where <digest> is MD5, SHA-1, SHA-256.

• NoPadding is appropriate for streaming modes.
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.
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 CC#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.
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
http://courseweb.stthomas.edu/resmith/c/csec/streamattack.html
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:

```
“Send Cash” encrypted
```

```
Encryption key
```

```
“Send Cash” Plaintext image
```
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. It 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):

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

Where, \( P_{n-1} \) is the last full plaintext block
\( P_n \) is the final, short, ciphertext block
\( C_{n-1} \) is the last full ciphertext block
\( C_n \) is the final, short ciphertext block
\( C' \) is just an intermediate result (not transmitted)
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 lying 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
  – 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 (more later)
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., is data integrity not always ensured?)
• If answer to these is “yes” for all questions, block replay is probably possible.
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”
**Horton Principle: ESAPI**

<table>
<thead>
<tr>
<th>Order</th>
<th>Size (in octets)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>KDF PRF &amp; version #</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>timestamp</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>xformLen</td>
</tr>
<tr>
<td>4</td>
<td>xformLen octets</td>
<td>cipherXform</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>keysize</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>blocksize</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>ivLen</td>
</tr>
<tr>
<td>8</td>
<td>ivLen octets</td>
<td>IV</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>ciphertextLen</td>
</tr>
<tr>
<td>10</td>
<td>ciphertextLen octets</td>
<td>rawCiphertext</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>macLen</td>
</tr>
<tr>
<td>12</td>
<td>macLen octets</td>
<td>MAC</td>
</tr>
</tbody>
</table>

MAC = HMAC-SHA256( authKey, IV || rawCiphertext )
where '||' denotes concatenation.
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)
More on Authenticated Encryption (AE)

• Common uses:
  – EtM: IPSec (and ESAPI 2.x :-)
  – E&M: SSH
  – MtE: SSL/TLS

• Of the 3 approaches, only EtM is proven to be strong against all known attacks.

• References:
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 real 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
Detour: Random Oracle

- Think “oracle” as in “Oracle of Delphi”, not as in Oracle, the software company.
  - Complexity theory: Oracle is an abstraction used to study decision problems.
  - Black box that decides yes / no to a given query.
- In cryptography, an oracle responds to a unique query with a truly random response,
  - But, the same query is answered the same each time it is submitted.
  - Cryptographers try to show a system is secure if modeled as a random oracle.
  - If one can distinguish a system is different than a random oracle, it is biased and therefore insecure.
Padding Oracle Attack (1/7)

• A chosen-ciphertext attack (CCA), generally performed as a side-channel attack performed against the padding of the ciphertext.
• The “side-channel” acts as the “oracle”... simplistically, the oracle might be something like “is the padding correct”. Ideally, this should be indistinguishable from a random oracle. When it is not, it is something that leaks a bit (or more) of information to the adversary.
Padding Oracle Attack (2/7)

- Side-channel can be:
  - Different error messages or exceptions
  - Error message or exception only on certain failures
  - To user or to log file (WYTM?)
  - Subtle differences in timing, CPU utilization, memory consumption, memory cache hits, etc.
Padding Oracle Attack (3/7)

• What can a padding oracle attack reveal?
  – The plaintext
  – Sometimes, allow encrypting arbitrary plaintext
  – With some additional work, sometimes the actual encryption key! (Rizzo and Duong)

• How can it be prevented?
  – By using an AE cipher mode like CCM or GCM
  – By using an EtM approach like IPSec or ESAPI
    • With EtM, want separate (derived) keys for encryption and MAC operations.
    • No “oracles” present when decryption error occurs.
Padding Oracle Attack (4/7)

• How does it work?
  – Explanation would add about 15-25 minutes to this talk. Search for “oracle padding attack” on YouTube.
  – Suggested references:
    • Holyfield’s OWASP presentation: http://blog.gdssecurity.com/storage/presentations/Padding_Oracle_OWASP_NYC.pdf
    • Dan Boneh lecture (padding oracle in TLS; 14 minute total): https://www.youtube.com/watch?v=evrgQkULQ5U
Padding Oracle Attack (5/7)

• Requirements for padding oracle attack
  – Must be using padding! (Duh!)
  – Some usable oracle must be available to the adversary that leaks information if padding error during decryption
    • Examples: Different error messages, different exception types, measurable timing differences, different messages logged, etc.
  – Adversary must be able to manipulate the ciphertext (or IV and ciphertext)
    • Usually an adaptive chosen ciphertext attack variation used
    • Examples: Encrypted HTTP parameters
Follow the logic of what happens when a BadPaddingException occurs.
– Is this logic different in any way than any other decryption error?
  • Exception type or message
  • Logged message contents (or even length difference)
  • Timing difference (timing side-channel)
    – 20 milliseconds or so is sufficient if we can take sufficient measurements to factor out the statistical network lag.
Padding Oracle Attack (7/7): Removing timing side channels as oracles

• Ideally, rewrite the code to eliminate the timing differences by going through the same logic for all error cases.
  – May not always be possible, especially if side channel is in another 3rd party library.
• Add small, but random sleeps for all cases
  – Approximate delay dependent on timing difference
• Ensure all take same amount of time by sleeping for N – t seconds where ‘t’ is amount of time taken for execution and N is something large (e.g., 2 seconds).
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
Three common algorithms for asymmetric encryption

- RSA – based on the integer factorization problem
- ElGamal – based on the Diffie-Hellman key exchange and the discrete logarithm problem
- Elliptic curve – based on the elliptic curve logarithm problem

Will only focus on RSA
- Because it won’t make your head explode
- EC is nuanced and not well supported (in Java at least)
  - Oracle does not yet support Elliptic Curve Integrated Encryption (ECIE) in Java 7, but only Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA).
Cipher modes for asymmetric encryption (applies to all algs) (1/2)

• Asymmetric cipher algorithms are on the order of 1000 times slower than their symmetric cipher counterparts.
• Therefore,
  – We very rarely (some would say never) encrypt more than 1 block of plaintext.
  – Usually only symmetric encryption keys, occasionally passwords.
• Implying,
  – We always use ECB mode.
Cipher modes for asymmetric encryption (applies to all algs) (1/2)

• Therefore, other modes need not apply.
• Cryptographer David Hopwood’s comment on using asymmetric ciphers with modes other than ECB:

  Some existing JCE providers will accept the use of a block cipher mode and padding with an asymmetric cipher (e.g. "RSA/CBC/PKCS#7"); this is not recommended, and new providers MUST reject this usage.
Common Asymmetric Padding Schemes

• No padding
• PKCS#1 v1.5 (simply called “PKCS1Padding” in Java)
• Optimal Asymmetric Encryption Padding (OAEP)
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 chose 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.
• 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#.
Asymmetric Cryptography: Digital Signatures
Digital Signature Issues

• There are standard attacks and specialized attacks on digital signatures in general and on specific digital signature schemes in particular. Not detailed here. See *Handbook of Applied Cryptography* if interested.

• Biggest problem is one of impersonation.
  ✓ How can Alice verify that Bob’s public key actually belongs to Bob and vice-versa.
  ✓ Several easy attacks (MITM, social engineering, etc.)
Digital Signatures: Other problems

• The private (signing) key is not properly secured.
• Alice may have multiple keys, especially over her lifetime, as she moves from job to job and one email address to another.
• If public key is not in a structure that ensures authenticity (e.g., a certificate in a key store with a passphrase) if can be changed.
What to look for

- Usually in Java, key pair is kept in a key store file. (In .NET, it often is just in a special XML file and not secured.)
- If in key store file, check:
  - Is private key secured with passphrase (to prevent loss of confidentiality)?
  - Is key store itself secured with (preferably different) passphrase (to prevent tampering)?
- If Alice’s key in X.509 cert, does Bob properly validate cert?
Miscellaneous Topics
Rekeying 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, rekey at least every $2^{32} \times 64$-bits of plaintext
  – For AES in CBC mode, every $2^{64} \times 128$-bits
  – General: every $2^{N/2} \times \text{cipher\_block\_size}$ bits, where $N$ is key size in bits.
TLS / SSL

- Dodgy things to look for:
  - Null cipher suites $\Rightarrow$ No encryption!
  - Assuming that SSLSocket / SSLSocketFactory correctly do server authentication
    - They correctly (in most cases) validate the server-side certificate, BUT
    - Early versions fail to do host name verification, so MITM attacks are still possible.
  - Same is true for URL and HttpUrlConnection when using an “https:” URL and early versions of Apache HttpClient
TLS/SSL Null Cipher Suites

- 8 in total
  - TLS_RSA_WITH_NULL_SHA256
  - TLS_ECDHE_ECDSA_WITH_NULL_SHA
  - TLS_ECDHE_RSA_WITH_NULL_SHA
  - SSL_RSA_WITH_NULL_SHA
  - TLS_ECDH_ECDSA_WITH_NULL_SHA
  - TLS_ECDH_RSA_WITH_NULL_SHA
  - TLS_ECDH_anon_WITH_NULL_SHA
  - SSL_RSA_WITH_NULL_MD5

- All disabled by default in JDK 7; all but the first disabled by default in JDK 6.
SSLSSocket and Server Authentication

• SSLSSocket (or any other SSLSSocket subclass) created by SSLSSocketFactory does not do host name verification or cert pinning by default. Hence, MITM attacks are possible.

  – Must implement your own. Two approaches:
  • Subclass SSLSSocket; see http://www.velocityreviews.com/forums/t958287-adding-hostname-verification-to-sslsocket.html
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 2\textsuperscript{nd} arg
• Determined by position; defaults to what is in java.security.
• This concept extends to crypto providers
What could possibly go wrong?

```java
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.SunMSC API
What could possibly go wrong?

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

```java
...
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 or email me at kevin.w.wall@gmail.com)