Plan

1. Digital Signatures Recall
2. Plain RSA Signatures and their many weaknesses
3. A Strengthening: PKCS#1 v1.5 RSA Signature Padding
4. An implementation error and its grave consequences
Assignment 1 is Due Tonight

Error in Problem 3 Hint:
- Technique outlined there omits an XOR with previous block.

If you want to test your code:
- Run attack with `cnet_id=davidcash` and `cnet_id=ravenben`
- Flag sizes vary in problems 2 and 3; Your attack should be robust to this
- (Especially on 2, where extra tricks are required for long flags.)
**Definition.** A digital signature scheme consists of three algorithms \( \text{Kg} \), \( \text{Sign} \), and \( \text{Verify} \)

- **Key generation algorithm** \( \text{Kg} \), takes no input and outputs a (random) public-verification-key/secret-signing key pair \((\text{VK}, \text{SK})\)

- **Signing algorithm** \( \text{Sign} \), takes input the secret key \( \text{SK} \) and a message \( M \), outputs “signature” \( \sigma \leftarrow \text{Sign}(\text{SK},M) \)

- **Verification algorithm** \( \text{Verify} \), takes input the public key \( \text{VK} \), a message \( M \), a signature \( \sigma \), and outputs ACCEPT/REJECT \( \text{Verify}(\text{VK},M,\sigma) = \text{ACCEPT/REJECT} \)
Digital Signature Security Goal: Unforgeability

Scheme satisfies **unforgeability** if it is unfeasible for Adversary (who knows VK) to fool Bob into accepting M’ not previously sent by Alice.
“Plain” RSA with No Encoding

\[ VK = (N, e) \quad SK = (N, d) \quad \text{where} \quad N = pq, \quad ed = 1 \mod \phi(N) \]

Sign\((N, d), M\) = \(M^d \mod N\)

Verify\((N, e), M, \sigma\) : \(\sigma^e = M \mod N\) ?

\(e = 3\) is common for fast verification; Assume \(e=3\) below.

Broken
"Plain" RSA Weaknesses

Assume $e=3$.

**Sign**\((N, d), M\) = $M^d \mod N$

**Verify**\((N, 3), M, \sigma\) : $\sigma^3 = M \mod N$?

To forge a signature on message $M'$: Find number $\sigma'$ such that $(\sigma')^3 = M' \mod N$

**M=1 weakness:** If $M'=1$ then it is easy to forge. Take $\sigma'=1$:

$$(\sigma'^3) = 1^3 = 1 = M' \mod N$$

**Cube-M weakness:** If $M'$ is a perfect cube then it is easy to forge. Just take $\sigma' = (M')^{1/3}$; i.e. the usual cube root of $M'$:

**Example:** To forge on $M'=8$, which is a perfect cube, set $\sigma'=2$.

$$(\sigma'^3) = 2^3 = 8 = M' \mod N$$

(Intuition: If cubing does not “wrap modulo $N$”, then it is easy to un-do.)
Further “Plain” RSA Weaknesses

Sign((N, d), M) = M^d mod N \quad \text{Verify}((N,3), M, \sigma) : \sigma^3 = M \mod N?

To forge a signature on message $M'$: Find number $\sigma'$ such that $(\sigma')^3 = M' \mod N$

**Malleability weakness:** If $\sigma$ is a valid signature for $M$, then it is easy to forge a signature on $8M \mod N$.

Given $(M, \sigma)$, compute forgery $(M', \sigma')$ as

$$M' = (8 \times M \mod N), \text{ and } \sigma' = (2 \times \sigma \mod N)$$

Then Verify((N, 3), M', \sigma') checks:

$$(\sigma')^3 = (2 \times \sigma \mod N)^3 = (2^3 \times \sigma^3 \mod N) = (2^3 \times M \mod N) = 8M \mod N$$

$\sigma^3 = M \mod N$ b/c $\sigma$ is valid sig. on $M$
Further “Plain” RSA Weaknesses

Sign\((N, d), M\) = \(M^d \mod N\)  Verify\(((N,3), M, \sigma) : \sigma^3 = M \mod N\) ?

To forge a signature on message \(M'\): Find number \(\sigma'\) such that \((\sigma')^3 = M' \mod N\)

**Malleability weakness:** If \(\sigma\) is a valid signature for \(M\), then it is easy to forge a signature on \(8M \mod N\).

**General form of malleability weakness:** If \(\sigma\) is a valid signature for \(M\), then it is easy to forge a signature on \(M' = (x*M \mod N)\) for any perfect cube \(x\).

\[
M' = x*M \mod N, \text{ and } \sigma' = (x^{1/3} * \sigma \mod N)
\]

Then Verify\(((N,3),M',\sigma')\) checks:

\[
(\sigma')^3 = (x^{1/3} * \sigma \mod N)^3 = (x * \sigma^3 \mod N) = (x*M \mod N) = (M' \mod N)
\]

\(\sigma^3 = M \mod N\) b/c \(\sigma\) is valid sig. on \(M\)
Further “Plain” RSA Weaknesses

\[ \text{Sign}((N, d), M) = M^d \mod N \quad \text{Verify}((N, 3), M, \sigma) : \sigma^3 = M \mod N? \]

To forge a signature on message \( M' \): Find number \( \sigma' \) such that \( (\sigma')^3 = M' \mod N \)

**Combining signatures weakness:** If \( \sigma_1 \) is a valid signature for \( M_1 \), and \( \sigma_2 \) is a valid signature for \( M_2 \)…

… then it is easy to compute signature \( \sigma' \) on \( M' = (M_1 \cdot M_2 \mod N) \)

\[ M' = (M_1 \cdot M_2 \mod N) \text{ and } \sigma' = (\sigma_1 \cdot \sigma_2 \mod N) \]

Then \( \text{Verify}((N, 3), M', \sigma') \) checks:

\[ (\sigma')^3 = (\sigma_1 \cdot \sigma_2 \mod N)^3 = (\sigma_1^3 \cdot \sigma_2^3 \mod N) = (M_1 \cdot M_2 \mod N) = (M' \mod N) \]

b/c \( \sigma_1, \sigma_2 \) are valid sigs
Further “Plain” RSA Weaknesses

Sign\((N, d), M\) = \(M^d \mod N\)  
Verify\(((N, 3), M, \sigma) : \sigma^3 = M \mod N?\)

To forge a signature on message \(M'\): Find number \(\sigma'\) such that \((\sigma')^3 = M' \mod N\)

**Backwards signing weakness:** Generate *some* valid signature by picking \(\sigma'\) first, and then defining \(M' = (\sigma'^3 \mod N)\)

Then \(\text{Verify}((N, 3), M', \sigma')\) checks:

\((\sigma')^3 = (M' \mod N)\)
Further “Plain” RSA Weaknesses

\[ \operatorname{Sign}((N,d), M) = M^d \mod N \quad \operatorname{Verify}((N,3), M, \sigma) : \sigma^3 = M \mod N? \]

To forge a signature on message \( M' \): Find number \( \sigma' \) such that \( (\sigma')^3 = M' \mod N \)

Summary:
- Plain RSA Signatures allow several types of forgeries
- It was sometimes argued that these forgeries aren’t important: If \( M \) is english text, then \( M' \) is unlikely to be meaningful for these attacks
- But often they are damaging anyway
RSA Signatures with Encoding

$VK = (N, e)$ $SK = (N, d)$ where $N = pq$, $ed = 1 \mod \phi(N)$

Sign($\langle N, d \rangle, M$) = encode($M$)$^d \mod N$

Verify($\langle N, e \rangle, M, \sigma$) : $\sigma^e = $ encode($M$)$ \mod N$?

decode maps bit strings to numbers in $\mathbb{Z}_N^*$

Encoding needs to address:
- Perfect cubes
- Malleability
- Backwards signing

Encoding must be chosen with extreme care.
RSA Signature Padding: PKCS #1 v1.5 (simplified)

**Note**: We already saw PKCS#1 v1.5 encryption padding. This is *signature* padding. It is different.

<table>
<thead>
<tr>
<th>N: n-byte long integer.</th>
<th>Ex: SHA-256, m=32</th>
</tr>
</thead>
<tbody>
<tr>
<td>H: Hash fcn with m-byte output.</td>
<td></td>
</tr>
</tbody>
</table>

**Sign((N,d),M):**
1. digest ← H(M) // m bytes long
2. pad ← FF||FF||…||FF // n-m-3 ‘FF’ bytes
3. X ← 00||01||pad||00||digest
4. Output σ = X^d mod N

**Verify((N,3),M,σ):**
1. X ← (σ^3 mod N)
2. Parse X → aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
   or Y≠(FF)^(n-m-3) or digest≠H(M):
   Output REJECT
4. Else: Output ACCEPT

Encoding needs to address:
- Perfect cubes
- Malleability
- Backwards signing

The high-order bits + digest means X is large and random-looking, rarely a cube.
Stopped by hash, ex: H(2*M)≠2*H(M)
Stopped by hash: given digest, hard to find M such that H(M)=digest.
RSA Signature Padding: PKCS #1 v1.5 (simplified)

**Note**: We already saw PKCS#1 v1.5 *encryption* padding. This is *signature* padding. It is different.

**N**: n-byte long integer.

**H**: Hash fcn with m-byte output.

**Ex**: SHA-256, m=32

<table>
<thead>
<tr>
<th>Sign((N,d),M):</th>
<th>Verify((N,3),M,σ):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. digest←H(M) // m bytes long</td>
<td>1. X←(σ^3 mod N)</td>
</tr>
<tr>
<td>2. pad←FF</td>
<td></td>
</tr>
<tr>
<td>3. X←00</td>
<td></td>
</tr>
<tr>
<td>4. Output σ = X^d mod N</td>
<td>or Y≠(FF)^n-m-3 or digest≠H(M):</td>
</tr>
<tr>
<td></td>
<td>Output REJECT</td>
</tr>
<tr>
<td></td>
<td>4. Else: Output ACCEPT</td>
</tr>
</tbody>
</table>

Introduces new weakness:

- Hash collision attacks: If H(M) = H(M’), then …

\[ \text{Sign((N,d),M)} = \text{Sign((N,d),M')} \]

- i.e., can reuse a signature for M as a signature for M’
Now: A Buggy Implementation, with an Attack

- Padding check is often implemented incorrectly

- Enables forging of signatures on *arbitrary* messages

Real-world attacks against:
- OpenSSL (2006)
- Apple OSX (2006)
- Apache (2006)
- VMWare (2006)
- All the biggest Linux distros (2006)
- Firefox/Thunderbird (2013)

…
(too many to list)
Buggy Verification in PKCS #1 v1.5 RSA Signatures

Sign((N,d),M):
1. digest←H(M) // m bytes long
2. pad←FF||FF||...||FF// n-m-3 ‘FF’ bytes
3. X←00||01||pad||00||digest
4. Output σ = X^d mod N

Verify((N,3),M,σ):
1. X←(σ^3 mod N)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00 or Y≠(FF)^n-m-3 or digest≠H(M):
   Output REJECT
4. Else: Output ACCEPT

BuggyVerify((N,3),M,σ):
1. X←(σ^3 mod N)
2. Parse X→aa||bb||rest
3. If aa≠00 or bb≠01:
   Output REJECT
4. Parse rest=(FF)^p||00||digest||..., where p is any number
5. If digest≠H(M): Output REJECT
6. Else: Output ACCEPT

Checks if rest starts with any number of FF bytes followed by a 00 byte.
If so, it takes the next m bytes as digest.

Correct: X = 00 01 FF FF FF FF FF FF FF FF FF 00 <DIGEST>

Buggy: X = 00 01 FF 00 <DIGEST> <IGNORED ............ BYTES>

One or more FF bytes
Attacking Buggy Verification

One or more FF bytes

**Buggy:** \[ X = 00 \ 01 \ FF \ 00 \ \text{<DIGEST>} \ \text{<IGNORED \ ............ \BYTES>} \]

To forge a signature on message \( M' \): Find number \( \sigma' \) such that

\[
(\sigma')^3 = 00 \ 01 \ FF \ 00 \ H(M') \ \text{<JUNK>} \mod N
\]

We'll use one FF byte \( m \) bytes long \( n-m-4 \) bytes free for attacker to pick

Freedom to pick \( \text{<JUNK>} \) means we can take any \( \sigma' \) such that:

\[00 \ 01 \ FF \ 00 \ H(M') \ 00 \ldots \ 00 \leq (\sigma')^3 \leq 00 \ 01 \ FF \ 00 \ H(M') \ FF \ldots \ FF\]

**Sufficient to find:** Any perfect cube in the given range. Then apply perfect cube attack.

**Easy!** (exercise)
Steps in Attack

1. Pick $m$ you want to forge on

2. Compute lower and upper bounds (numbers), using $H(m)$.

3. Find a perfect cube $x$ within allowed range

4. Output cube root of $x$ as forged signature $\sigma$. 

Attack Summary

- When padding check allows variable number of FF bytes, forging is easy
  - Only requires a simple search for a perfect cube in a given range
- *Why did so many make this error?*
  - I don’t know
  - My guesses:
    - Plugging in libraries for padding removal without context
    - Traditional unit testing is hard to apply to crypto.
    - The details omitted in my description of the padding make parsing much harder. (Actual version includes in X an ASN.1 identifier of hash function, which is complicated in full generality.)
- Attack defeated by using large $e=65537$
Lesson with Implementing Signatures

- **Verify** should simply re-run signing and check if same signature comes out
- Not strictly possible if **Sign** is randomized.
Other RSA Padding Schemes: Full Domain Hash

\[ N: \text{n-byte long integer.} \]
\[ H: \text{Hash fcn with m-byte output.} \]
\[ k = \text{ceil}((n-1)/m) \]

Ex: SHA-256, \( m = 32 \)

**Sign\((N,d),M\):**
1. \( X \leftarrow 00 \ || H(1 \ | \ M) \ || H(2 \ | \ M) \ | \ldots | \ H(k \ | \ M) \)
2. Output \( \sigma = X^d \mod N \)

**Verify\((N,e),M,\sigma\):**
1. \( X \leftarrow 00 \ || H(1 \ | \ M) \ || H(2 \ | \ M) \ | \ldots | \ H(k \ | \ M) \)
2. Check if \( \sigma^e = X \mod N \)

**Bonus:** Can prove security, in a strong sense.
Other RSA Padding Schemes: PSS

- Somewhat complicated
- *Randomized* signing

**Bonus:** Can *prove* security, in a strong sense.
RSA Signature Summary

- Plain RSA signatures are very broken
- PKCS#1 v.1.5 is widely used, in TLS, and fine if implemented correctly
- Full-Domain Hash and PSS should be preferred
- Don’t roll your own RSA signatures!
Other Practical Signatures: DSA/ECDSA

- Based on ideas related to Diffie-Hellman key exchange
- Secure, but ripe for implementation errors

Hackers obtain PS3 private cryptography key due to epic programming fail? (update)

```java
int getRandomeNumber()
{
    return 4;  // chosen by fair dice roll.
    // guaranteed to be random.
}
```
The End