Outline

- Message Authentication
- Hash Functions
- Public-Key Encryption
- Digital Signatures
- **Message Authentication**
  - Hash Functions
  - Public-Key Encryption
  - Digital Signatures
Next Up: Integrity and Authentication

- Authenticity: Guarantee that adversary cannot change or insert ciphertexts
- Achieved with MAC = “Message Authentication Code”
Encryption Integrity: An abstract setting

Encryption satisfies **integrity** if it is infeasible for an adversary to send a new $C'$ such that $\text{Dec}_K(C') \neq \text{ERROR}$. 
Stream ciphers do not give integrity

\[ M = \text{please pay ben 20 bucks} \]
\[ C = \text{b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e446a782871c2d} \]
\[ C' = \text{b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e546a782871c2d} \]
\[ M' = \text{please pay ben 21 bucks} \]

Inherent to stream-cipher approach to encryption.
A **message authentication code (MAC)** is an algorithm that takes as input a key and a message, and outputs an “unpredictable” tag.
MAC Security Goal: Unforgeability

MAC satisfies **unforgeability** if it is infeasible for Adversary to fool Bob into accepting $M'$ not previously sent by Alice.
MAC Security Goal: Unforgeability

Note: No encryption on this slide.

\( M = \text{please pay ben 20 bucks} \)

\( T = 827851dc9cf0f92ddc552572ffd8bc \)

\( M', T' \)

\( M' = \text{please pay ben 21 bucks} \)

\( T' = \text{baeaf48a891de588ce588f8535ef58b6} \)

Should be hard to predict \( T' \) for any new \( M' \).
MACs In Practice: Use HMAC or Poly1305-AES


- Other, less-good option: AES-CBC-MAC (bug-prone)
Authenticated Encryption

Encryption that provides **confidentiality** and **integrity** is called **Authenticated Encryption**.

- Built using a good stream cipher and a MAC.
  - Ex: Salsa20 with HMAC-SHA2
- Best solution: Use ready-made Authenticated Encryption
  - Ex: AES-GCM is the standard
- Message Authentication
- Hash Functions
- Public-Key Encryption
- Digital Signatures
Next Up: Hash Functions

**Definition:** A hash function is a deterministic function $H$ that reduces arbitrary strings to fixed-length outputs.

![Diagram showing a hash function]

Some security goals:
- collision resistance: can’t find $M \neq M'$ such that $H(M) = H(M')$
- preimage resistance: given $H(M)$, can’t find $M$
- second-preimage resistance: given $H(M)$, can’t find $M'$ s.t. $H(M') = H(M)$

Note: Very different from hashes used in data structures!
Why are collisions bad?

The binary should hash to 3477a3498234f

MD5(100001)=3477a3498234f

Hashes to 3477a3498234f, so let's install!

MD5(100001)=3477a3498234f
## Practical Hash Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Output Len (bits)</th>
<th>Broken?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>1993</td>
<td>128</td>
<td>Super-duper broken</td>
</tr>
<tr>
<td>SHA-1</td>
<td>1994</td>
<td>160</td>
<td>Yes</td>
</tr>
<tr>
<td>SHA-2 (SHA-256)</td>
<td>1999</td>
<td>256</td>
<td>No</td>
</tr>
<tr>
<td>SHA-2 (SHA-512)</td>
<td>2009</td>
<td>512</td>
<td>No</td>
</tr>
<tr>
<td>SHA-3</td>
<td>2019</td>
<td>&gt;=224</td>
<td>No</td>
</tr>
</tbody>
</table>

Confusion over “SHA” names leads to vulnerabilities.
Hash Functions are not MACs

Both map long inputs to short outputs… But a hash function does not take a key.

**Intuition**: a MAC is like a hash function, that only the holders of key can evaluate.
**Goal:** Build a secure MAC out of a good hash function.

**Construction:** $MAC(K, M) = H(K || M)$

- Totally insecure if $H = MD5, SHA1, SHA-256, SHA-512$
- Is secure with SHA-3 (but don’t do it)

**Construction:** $MAC(K, M) = H(M || K)$

**Upshot:** Use HMAC; It’s designed to avoid this and other issues.

Later: Hash functions and certificates
Length Extension Attack

Construction: \( \text{MAC}(K, M) = H(K \| M) \)  

**Warning:** Broken

**Adversary goal:** Find new message \( M' \) and a valid tag \( T' \) for \( M' \)

**Need to find:** Given \( T = H(K \| M) \), find \( T' = H(K \| M') \) without knowing \( K \).

**In Assignment 3:** Break this construction!
Outline

- Message Authentication
- Hash Functions
- **Public-Key Encryption**
- Digital Signatures
Basic question: If two people are talking in the presence of an eavesdropper, and they don’t have pre-shared a key, is there any way they can send private messages?
Basic question: If two people are talking in the presence of an eavesdropper, and they don’t have pre-shared a key, is there any way they can send private messages?

Diffie and Hellman in 1976: Yes!
Turing Award, 2015, + Million Dollars

Rivest, Shamir, Adleman in 1978: Yes, differently!
Turing Award, 2002, + no money

Cocks, Ellis, Williamson in 1969, at GCHQ: Yes…
Basic question: If two people are talking in the presence of an eavesdropper, and they don’t have pre-shared a key, is there any way they can send private messages?

Formally impossible (in some sense): No difference between receiver and adversary.
The Seed of Public-Key Cryptography

**Basic question:** If two people are talking in the presence of an eavesdropper, and they don’t have pre-shared a key, is there any way they can send private messages?

Message $M$  
$R \leftarrow \text{rand}()$

<some bits>  
<some bits>  
<some bits>

$R' \leftarrow \text{rand}()$

Receive $M$

Doesn’t know $R, R'$, Can’t “try them all” (too many)
A public-key encryption scheme consists of three algorithms: KeyGen, Encrypt, and Decrypt.

**KeyGen**: Outputs two keys. PK published openly, and SK kept secret.

**Encrypt**: Uses PK and M to produce a ciphertext C.

**Decrypt**: Uses SK and C to recover M.
Public-Key Encryption in Action

PK = public key
known to everyone

SK = secret key
known by Receiver only

M \rightarrow PK \rightarrow C = \text{Enc}(PK, M) \rightarrow SK \rightarrow M

PK

SK

PK

SK
Some RSA Math

RSA setup

p and q be large prime numbers (e.g. around $2^{2048}$)

$N = pq$

$N$ is called the **modulus**

Called “2048-bit primes”

$p=7$, $q=11$ gives $N=77$

$p=17$, $q=61$ gives $N=1037$
RSA “Trapdoor Function”

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

\[ \text{RSA}((N, e), x) = x^e \mod N \]

\[ \text{RSA}^{-1}((N, d), y) = y^d \mod N \]

Setting up RSA:
- Need two large random primes
- Have to pick \( e \) and then find \( d \)
- Not covered in 232/332: How this really works.

Never use directly as encryption!  

Warning: Broken
Encrypting with the RSA Trapdoor Function

“Hybrid Encryption”:
- Apply RSA to random \( x \)
- Hash \( x \) to get a symmetric key \( k \)
- Encrypted message under \( k \)

\[
\text{Enc}((N,e),M): \\
1. \text{Pick random } x \quad \text{// } 0 \leq x < N \\
2. c_0 \leftarrow (x^e \text{ mod } N) \\
3. k \leftarrow H(x) \\
4. c_1 \leftarrow \text{SymEnc}(k,M) \quad \text{// symmetric enc.} \\
5. \text{Output } (c_0,c_1)
\]

\[
\text{Dec}((N,d), (c_0,c_1)): \\
1. x \leftarrow (c_0^d \text{ mod } N) \\
2. k \leftarrow H(x) \\
3. M \leftarrow \text{SymDec}(k,c_1) \\
4. \text{Output } M
\]

\textbf{Do not implement yourself!}

\textit{Warning: Broken}

- Use RSA-OAEP, which uses hash in more complicated way.
# Factoring Records and RSA Key Length

- Factoring N allows recovery of secret key
- Challenges posted publicly by RSA Laboratories

<table>
<thead>
<tr>
<th>Bit-length of N</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>1993</td>
</tr>
<tr>
<td>478</td>
<td>1994</td>
</tr>
<tr>
<td>515</td>
<td>1999</td>
</tr>
<tr>
<td>768</td>
<td>2009</td>
</tr>
<tr>
<td>795</td>
<td>2019</td>
</tr>
</tbody>
</table>

- Recommended bit-length today: 2048
- Note that fast algorithms force such a large key.
  - 512-bit N defeats naive factoring
Outline

- Message Authentication
- Hash Functions
- Public-Key Encryption
- **Digital Signatures**
A digital signature scheme consists of three algorithms **KeyGen**, **Sign**, and **Verify**.

- **KeyGen**: Outputs two keys. PK published openly, and SK kept secret.
- **Sign**: Uses SK to produce a “signature” σ on M.
- **Verify**: Uses PK to check if signature σ is valid for M.
Digital Signature Security Goal: Unforgeability

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

$$PK = (N, e) \quad SK = (N, d) \quad \text{where} \quad N = pq, \quad ed = 1 \mod \phi(N)$$

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

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

e = 3 \quad \text{is common for fast verification.}
RSA Signatures with Encoding

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

\[
\text{Sign}((N, d), M) = \text{encode}(M)^d \mod N
\]

\[
\text{Verify}((N, e), M, \sigma) : \sigma^e = \text{encode}(M) \mod N?
\]

\text{encode} maps bit strings to numbers between 0 and N

\text{Encoding must be chosen with extreme care.}
Example RSA Signature: Full Domain Hash

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

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 \(\sigma^e = X \mod N\)
Other RSA Padding Schemes: PSS (In TLS 1.3)

- Somewhat complicated
- Randomized signing
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 even more ripe for implementation errors

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

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