Outline

- Message Authentication
- Hash Functions
- Public-Key Encryption
- Digital Signatures
Outline

- **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 $\text{Dec}_K(C') \neq \text{ERROR}$. 

$C \leftarrow \text{Enc}_K(M)$

$C' \leftarrow \text{Dec}_K(C')$ or “ERROR”
AES-CTR does not satisfy 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 unfeasible for Adversary to fool Bob into accepting $M'$ not previously sent by Alice.
MAC Security Goal: Unforgeability

Note: No encryption on this slide.

$M = \text{please pay ben 20 bucks}$

$T = 827851dc9cf0f92ddcdc552572fffd8bc$

$M', T'$

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

$T' = \text{baeaf48a891de588ce588f8535ef58b6}$

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

- Don’t worry about how it works.

- 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 cipher and a MAC.
  - Ex: AES-CTR with HMAC-SHA2
- Best solution: Use ready-made Authenticated Encryption
  - Ex: AES-GCM is the standard
Authenticated Encryption Security: Adversary cannot recover any useful information about plaintexts that it didn’t form itself OR fool party into accepting some $c’$ that wasn’t sent.
Building Authenticated Encryption

Encrypt_{K1,K2}(M)

$M \xrightarrow{} \text{Enc}_{K1}() \xrightarrow{} \text{MAC}_{K2}() \xrightarrow{} \text{C} \xrightarrow{} \text{T}$

Output: $(C,T)$

Decrypt_{K1,K2}(C,T)

$C \xrightarrow{} \text{MAC}_{K2}() \xrightarrow{} T' \xrightarrow{} T = T'$

Output: $M' \text{ if } T' = T$

$\bot \text{ if } T' \neq T$

- Summary: MAC the ciphertext, not the message
Outline

- 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.

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(100 001) = 3477a3498234f

Hashes to 3477a3498234f, so let's install!

MD5(100 001) = 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.
MACs from Hash Functions

**Goal:** Build a secure MAC out of a good hash function.

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

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

**Warning:** Broken

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

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

**Warning:** Broken

**Just don’t**

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

**Later:** Hash functions and certificates
Outline

- Message Authentication
- Hash Functions
- **Public-Key Encryption**
- Digital Signatures
Public-Key Encryption

**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?
Public-Key Encryption

**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.
Public-Key Encryption

**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)
Public-Key Encryption

**Definition.** A public-key encryption scheme consists of three algorithms $Kg$, $Enc$, and $Dec$

- **Key generation algorithm** $Kg$, takes no input and outputs a (random) public-key/secret key pair $(PK, SK)$

- **Encryption algorithm** $Enc$, takes input the public key $PK$ and the plaintext $M$, outputs ciphertext $C \leftarrow Enc(PK, M)$

- **Decryption algorithm** $Dec$, is such that
  \[
  Dec(SK, Enc(PK, M)) = M
  \]
Public-Key Encryption in Action

PK = public key
known to everyone

SK = secret key
known by Receiver only

Kg

PK, SK

M → PK

PK

C = Enc(PK, M)

SK

M →

PK

C
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

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

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

Do not implement yourself!

- Use RSA-OAEP, which uses hash in more complicated way.

Warning: Broken
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
Key Exchange and Hybrid Encryption (TLS next week)

\((Kg, \text{Enc}, \text{Dec})\) is a public-key encryption scheme.

**Goal:** Establish secret key \(k\) to use with Authenticated Encryption.

Pick random AES key \(k\)

\[ C = \text{Enc}(PK, k) \]

\(k\) is the message

Maybe be long-term key or “ephemeral” key pair, used only once.
- After up-front cost, bulk encryption is very cheap
- TLS Terminology:
  - “Handshake” = key exchange
  - “Record protocol” = symmetric encryption phase
An alternative approach to key exchange

- They modulus N for RSA is relatively large
  - Mostly important because it slows down encryption/decryption

- Now: A totally different, faster approach based on different math
  - Invented in 1970s, but new ideas have recently made it the standard choice
  - Strictly speaking, not public-key encryption, but can adapted into it if needed
The Setting: Discrete Logarithm Problem

**Discrete Logarithm Problem:**

**Input:** Prime \( p \), integers \( g, x \).

**Output:** integer \( r \) such that \( g^r = x \mod p \).

- Different from factoring: Only one prime.
- Contrast with logarithms with real numbers, which are easy to compute. *Discrete* logarithms appear to be hard to compute.
- Largest solved instances: 795-bit prime \( p \) (Dec 2019)
Diffie-Hellman Key Exchange

Parameters: (fixed in standards, used by everyone):
Prime $p$ (1024 bit usually)
Number $g$ (usually 2)

$(p, g)$
Diffie-Hellman Key Exchange

Parameters: (fixed in standards, used by everyone):
Prime $p$ (1024 bit usually)
Number $g$ (usually 2)

Network Working Group
Request for Comments: 5114
Category: Informational

M. Lepinski
S. Kent
BBN Technologies
January 2008

Additional Diffie-Hellman Groups for Use with IETF Standards

Status of This Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Further discussion
may occur on the IETF-dh-interest@listserv.net mailing list. This
memo is unlimited.

Abstract

This document describes eight Diffie-Hellman groups in conjunction with IETF protocols for Internet key management communications. The groups allow use with a variety of security protocols, including (SSH), Transport Layer Security (TLS), and Internet Key Exchange (IKE).

3. 2048-bit MODP Group

This group is assigned id 14.

This prime is: $2^{2048} - 2^{1984} - 1 + 2^{64} \times \{2^{1918} \pi + 124476 \}$

Its hexadecimal value is:

FFFFF_FF_8999_F5_AE9F2411_7C4B1FE6_49286651_ECE45B3D
C007CB8_A163BF05_98DA4836_1C55D39A_69163FA8_FD24CF5F
83655D23_DCA3AD96_1C62F356_208552BB_9ED52907_7096966D
670C354E_4ABC9804_F1746C08_CA18217C_32905E46_2E36CE3B
E39E772C_180EB603_9B2783A2_ECO7A28F_B5C55DF0_6F4C52C9
DE2BCBF6_95581718_3955497C_EA956AE5_15D2261B_98FA0510
15728E5A_8AACA68_FFFFF_FF_8999_F5_AE9F2411_7C4B1FE6_49286651_ECE45B3D

The generator is: 2.
Diffie-Hellman Key Exchange

Parameters: (fixed in standards, used by everyone):
Prime $p$ (1024 bit usually)
Number $g$ (usually 2)

$(p, g)$

Pick $r_A \in \{1, \ldots, p-1\}$
$X_A \leftarrow g^{r_A} \mod p$

Pick $r_B \in \{1, \ldots, p-1\}$
$X_B \leftarrow g^{r_B} \mod p$

$K \leftarrow X_B^{r_A} \mod p$

Correctness: $X_B^{r_A} = (g^{r_B})^{r_A} = g^{r_A r_B} = (g^{r_A})^{r_B} = X_A^{r_B} \mod p$
Modern Key Exchange: *Elliptic Curve* Diffie-Hellman

- Totally different math from RSA
- Advantage: Bandwidth and computation (due to higher security)
  - 256 bit vs 2048-bit messages.
- Used by default in TLS
Public-Key Encryption/Key Exchange Wrap-Up

- RSA-OAEP and Diffie-Hellman (either mod a prime or in an elliptic curve) are unbroken and run fine in TLS/SSH/etc.
- Elliptic-Curve Diffie-Hellman is preferred choice going forward.

Quantum computers will break:
- RSA (any padding)
- Diffie-Hellman

- First gen quantum computers will be far from this large
- “Post-quantum” crypto = crypto not known to be broken by quantum computers (i.e. not RSA or DH)
- On-going research on post-quantum cryptography from hard problems on lattices, with first beta deployments in recent years
Outline

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

- Key generation algorithm $Kg$, takes no input and outputs a (random) public-verification-key/secret-signing key pair $(PK, SK)$

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

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

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

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

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

\[ e = 3 \quad \text{is common for fast verification; Assume } e=3 \text{ below.} \]
PK = (N, e)  SK = (N, d)  where  N = pq,  ed = 1 \mod \phi(N)

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

encode maps bit strings to numbers between 0 and N

Encoding needs to address:
- Small M or M = perfect cube
- “Malleability”
- “Backwards signing”
- …
Example RSA Signature: Full Domain Hash

N: n-byte long integer.
H: Hash fcn with m-byte output.
k = ceil((n-1)/m)

Ex: SHA-256, m=32

Sign((N,d),M):
1. X←00||H(1||M)||H(2||M)||…||H(k||M)
2. Output σ = X^d mod N

Verify((N,e),M,σ):
1. X←00||H(1||M)||H(2||M)||…||H(k||M)
2. Check if σ^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)
The End