IDATT2503, Part 2 Cryptography

Lecture 6: Remaining topics.

Hans J. Rivertz

14th. November

Plan for the lecture

1. ElGamal Encryption
2. Passwords
3. Kerberos 5 — Quick Summary

ElGamal Encryption

1. ElGamal Encryption
2. Passwords
3. Kerberos 5 — Quick Summary

ElGamal Cipher

Overview:

- ElGamal is a public-key encryption system based on the **Discrete Logarithm Problem (DLP)**.
- Relies on a large prime p and a generator g of $\mathbb{Z}_p^*$.
- Each user has a private key x and a public key $k = g^x \bmod p$.

ElGamal Encryption / Decryption

Encryption:

- To send a message m, choose a random y.
- Compute the ciphertext pair:

$$c_1 = g^y \bmod p, \quad c_2 = m \cdot k^y \bmod p$$

Decryption:

- Recover m using private key x:

$$m = c_2 \cdot (c_1^x)^{-1} \bmod p$$

Security: Based on the assumption that solving the discrete logarithm problem is hard.

ElGamal: Parameters

Setup:

- Prime $p = 47$
- Generator $g = 5$

Key generation:

- Private key $x = 15$, (chosen random)
- Public key $k = g^x \bmod p = 5^{15} \bmod 47 = 41$

Resulting keys:

Public key: $(p = 47,\ g = 5,\ k = 41)$,   Private key: $x = 15$

ElGamal: Encryption

Message: $m = 10$ **Ephemeral key:** $y = 7$ (Random number)

Compute:

$$c_1 = g^y \bmod p = 5^7 \bmod 47 = 11$$
$$h^y \bmod p = 31^7 \bmod 47 = 15$$
$$c_2 = m \cdot (h^y \bmod p) \bmod p = 10 \cdot 15 \bmod 47 = 9$$

Ciphertext: $(c_1, c_2) = (11,\ 9)$

ElGamal: Decryption

Ciphertext: $(c_1, c_2) = (11, 9)$ **Private key:** $x = 15$

Compute shared secret:

$$s = c_1^x \bmod p = 11^{15} \bmod 47 = 10$$

Find inverse of s:

$$s^{-1} \bmod 47 = 19 \quad (\text{since } 10 \cdot 19 \bmod 47 = 1)$$

Recover the message:

for Large p use Euclid's algorithm.

$$m = c_2 \cdot s^{-1} \bmod p = 9 \cdot 19 \bmod 47 = 10$$

Decrypted message: $m = 10$

Malleability Attack on ElGamal

Key property: ElGamal is *multiplicatively homomorphic*:

$$(c_1, c_2) = (g^r,\ m \cdot h^r)$$

Attack idea: An attacker who knows the plaintext m can modify the ciphertext to change the message.

Attack step: Given ciphertext (c_1, c_2) of message m, produce

$$(c_1,\ nm^{-1} \cdot c_2)$$

Receiver decrypts to: $(nm^{-1} \cdot c_2) \cdot c_1^{-x} = n$

Consequence:

- Attacker can transform a ciphertext of m into a ciphertext of any value n
- No need for the secret key
- Enables tampering / man-in-the-middle attacks

Countermeasures Against Malleability

1. **Authenticated Encryption (Recommended)**
   - Use ElGamal only to derive a symmetric key
   - Encrypt the message with AEAD (e.g., AES-GCM, ChaCha20-Poly1305)
   - Tampering breaks authentication tag
2. **Add Authentication to ElGamal**
   - Sign the ciphertext or attach a MAC (HMAC)
   - Modified ciphertexts fail verification
3. **Redundant Encoding (Weak Mitigation)**
   - Add unforgeable redundancy or structured headers
   - Helps detect tampering but weaker than AEAD/CCA

Passwords

Passwords

- One of the most important authentication methods.
- Can we make it redundant?
- Secret that controls access to resources and services.
- Leakage of passwords happens (in plaintext or hashes).

Passwords have often been:

- Weak (short, easy to guess), reused
- Stored with poor security measures (weak or no encryption)

Password security also has psychological issues, not only technical.

Password Transmission

- Passwords are always sent over **HTTPS (TLS)**.
- HTTPS protects against eavesdropping and tampering.
- The client does not need to know the salt.

Salt will be explained.

Password Attacks

We consider one type: **Brute Force**.

- Using lists of likely passwords or common patterns.
- Attack types:
  - Online: same interface as user.
  - Offline: access to account hashes. (Password leaks.)
- Targeting:
  - General: any user
  - Targeted: specific users

Password Security

Focus: cryptographic storage and transmission.

- How should we store passwords?

Why Not Store Passwords Directly?

- Plaintext passwords can be leaked in data breaches.
- Users often reuse passwords across multiple services.
- Solution: store only the **hash** of the password.

Storing Passwords

- Plain text? **No.**
- Encrypted password? Problem: needs a key!
  - Compromised key = compromised passwords
  - Encryption reveals plaintext length
  - Only account owner should know the password
- Hash of password: better
  - Good: fixed length, one-way
  - Bad: not resistant to brute force, same passwords yield same hash

Salted Hashes and Specialized Algorithms

- Salted hash: better
  - Randomness: same password $\rightarrow$ different hashes
  - Still fast for brute force
- Specialized password hashing algorithms
  - Same advantages as salted hash
  - Slows attacks (CPU/memory intensive)
  - Parameters: cost, memory, parallelism

Salted Hash for Security

- A **salt** is a random string added to a password before hashing.
- Example:

  Password: "mypassword" + Salt: "A1B2C3"

  $\Rightarrow$ Hash: `7c6a180b36896a0a8c02787eeafb0e4c`

- Benefits:
  - Even same passwords produce different hashes.
  - Protects against rainbow table attacks.

Password Hashing Algorithms

Password hashing is a critical step in protecting user credentials. In the next few slides, we'll take a closer look at three key algorithms:

- **PBKDF2:** A long-established, FIPS-140-compliant algorithm used in WPA, Android, TrueCrypt and other systems.
- **scrypt:** A more modern alternative designed for high memory usage — recommended when more advanced options aren't available.
- **Argon2:** The current recommended choice — winner of the Password Hashing Competition (PHC) in 2015 and widely regarded as the state-of-the-art.

FIPS 140 Compliance

FIPS 140 (Federal Information Processing Standard 140) defines security requirements for cryptographic modules used by U.S. government systems.

- Ensures algorithms and implementations meet rigorous standards.
- PBKDF2 is **FIPS 140-compliant**, which is why it's widely used in WPA, Android, and TrueCrypt.
- Compliance does not always mean "most secure" today — modern alternatives like Argon2 may offer better resistance against GPU/ASIC attacks.

Key takeaway: FIPS 140 compliance guarantees validated implementation, but algorithm choice still matters for modern threat models.

PBKDF2 (Password-Based Key Derivation Function 2)

- **Purpose:** Derive cryptographic keys from passwords.
- **Mechanism:**
  - Applies a pseudorandom function (e.g., HMAC) repeatedly.
  - Uses a salt to defend against precomputed attacks.
- **Configurable parameters:**
  - Iteration count c (slows brute force attacks)
  - Output key length $dkLen$
- **Example:** $DK = \text{PBKDF2}(\text{password}, \text{salt}, c, dkLen)$
- **Pros:** Simple, widely supported.
- **Cons:** CPU-bound only, memory-efficient attacks are possible.

Scrypt

- **Purpose:** Memory-hard password-based key derivation function.
- **Mechanism:**
  - Uses both CPU and significant memory to resist ASIC attacks.
  - Mixes input with large memory blocks via `ROMix`.
- **Configurable parameters:**
  - N – CPU/memory cost factor
  - r – block size
  - p – parallelization factor
- **Example:** $DK = \text{scrypt}(\text{password}, \text{salt}, N, r, p, dkLen)$
- **Pros:** Strong defense against hardware attacks.
- **Cons:** More complex, slower than PBKDF2.

Memory-Intensive Hash Functions

- Traditional hash functions (e.g., SHA-256) are **CPU-bound**:
  - Very fast to compute
  - Require minimal memory
  - Vulnerable to brute-force attacks on specialized hardware (GPUs, ASICs)
- **Memory-hard hash functions** intentionally require large amounts of RAM:
  - Store intermediate blocks in memory during computation
  - Force attackers to allocate large memory per attempt
  - Makes parallel attacks expensive
- Examples: **Scrypt**, **Argon2**

Kerberos 5 — Quick Summary

Kerberos 5 — Overview

- Kerberos is a network authentication protocol that provides **mutual authentication** and **single sign-on (SSO)** using **tickets**.
- Main components:
  - **Client (C)** — user or process requesting access
  - **Key Distribution Center (KDC)** = AS + TGS
  - **Service (S)** — target service (e.g., file server)
- Cryptography:
  - Symmetric keys (derived from passwords, and long-term keys for services/KDC)
  - Time-stamped authenticators to prevent replay attacks
- Security goals:
  - Never send plaintext password on the network
  - Short-lived tickets / session keys limit exposure
  - Mutual authentication (optional/depends on flow)

Kerberos 5 — Protocol Flow (Notation)

Notation

- K_c: key derived from client's password
- K_{tgs}, K_s: long-term keys for TGS and service
- $K_{c,tgs}$: session key between client and TGS
- $K_{c,s}$: session key between client and service
- TGT: Ticket Granting Ticket

Kerberos 5 — Protocol Flow (Message sequence)

1. **AS-REQ:** (Authentication Service Request)
   $C \rightarrow AS$: ClientID, Realm, Nonce, ...
   (Request a Ticket Granting Ticket (TGT))
2. **AS-REP:** (Authentication Service Reply)
   $AS \rightarrow C : e(K_{c,tgs}, K_c)$, $e(\mathrm{Ticket}_{tgs}, K_{tgs})$
   — client uses K_c (from password) to decrypt $K_{c,tgs}$
3. **TGS-REQ:** (Ticket-Granting Service Request)
   $C \rightarrow TGS$: $\mathrm{Ticket}_{\cancel{gs}}$, $\mathrm{Authenticator}_c$ (encrypted with $K_{c,tgs}$), ServiceID tgs
4. **TGS-REP:** (Ticket-Granting Service Reply)
   $TGS \rightarrow C : e(K_{c,s}, K_{c,tgs}), e(\mathrm{Ticket}_s, K_s)$
5. **AP-REQ (to service):** (Application Request)
   $C \rightarrow S$: Ticket_s, $\mathrm{Authenticator}_c$ (encrypted with $K_{c,s}$)
6. **AP-REP (optional mutual auth):** (Application Reply)
   $S \rightarrow C : e(Timestamp + 1, K_{c,s})$

Kerberos 5 — Protocol Flow (Key points).

Key points

- Passwords are only used locally to derive K_c; they are never sent on the wire.
- Tickets are opaque blobs to clients (encrypted to the service/TGS).
- Timestamps and nonces prevent replay attacks; ticket lifetimes limit risk.

Important Keys in Kerberos 5

- K_c — Client's long-term key
  - Derived from the user's password (string-to-key)
  - Used to decrypt the AS reply containing $K_{c,tgs}$
- K_{tgs} — TGS long-term secret key
  - Only known by the Key Distribution Service (AS/TGS)
  - Used to encrypt and verify the Ticket Granting Ticket (TGT)
- $K_{c,tgs}$ — Client–TGS session key
  - Issued by the AS in the AS-REP
  - Used for secure communication between the client and TGS
- K_s — Service server's long-term key
  - Known only by the service and the KDC
  - Used to encrypt the service ticket: $e(\mathrm{Ticket}_s, K_s)$
- $K_{c,s}$ — Client–Service session key
  - Issued by the TGS in the TGS-REP
  - Used between client and service during the AP exchange