ISE Cryptography – Lecture 07

Authenticated Encryption, KDFs and Capstone Preparation

Final Exam

- 100 marks overall for 40% of your cryptography grade
  - 50 MCQs (1 mark each)
  - 10 written-response questions (5 marks each)
- Allocated time: 110 minutes (1 hour 50 minutes)
  - Suggested timing: 1 mark per minute
- No one leaves during the first 30 minutes or the last 15 minutes of the exam
  - Once you submit the exam, close your laptop
  - If you're done outside those timeslots, you can leave
  - Don't congregate in the kitchen so as not to disturb your fellow students
- You should have nothing open but Brightspace and notifications should be off
  - Minimum penalty for academic dishonesty is getting 0% on the final exam
- If you have a question, raise your hand
  - If you need a bathroom break, same applies

This Lecture

- The exam is behind you. This session closes the theoretical arc and prepares you for the project
- We're going to:
  - Prove the Encrypt-then-MAC CCA theorem we stated in Lecture 04
  - Look at AEAD operational details (AES-GCM internals, nonce misuse resistance)
  - Cover HKDF for deriving keys from shared secrets
  - Cover password hashing with PBKDF2 and Argon2
  - Walk through a lab exercise that previews the EPIC project's core cryptographic stack
- Pay close attention to the cryptography spec for the EPIC!
  - Make sure you deliver everything that's asked for
  - And make sure you can defend the choices you make!

The Encrypt-then-MAC CCA Proof

The theorem from Lecture 04, now proved.

Recall: The CCA Theorem

- In Lecture 04, we stated: Encrypt-then-MAC with a CPA-secure cipher and a secure MAC achieves IND-CCA2 security
- In Lecture 06, we defined IND-CCA2 formally
- Now let's prove it. The argument has two steps:
  - Step 1: The MAC prevents the adversary from using the decryption oracle effectively
  - Step 2: Without useful decryption queries, the adversary faces only CPA security

Step 1: The MAC Neuters the Decryption Oracle

- In the CCA2 game, the adversary can submit ciphertexts (c, t) for decryption
  - But they can't submit any (c_i, t_i) received from an encryption query
- For any other (c', t'), the MAC verification $V(k_m, c', t')$ must pass
  - If the MAC is secure (UF-CMA), the adversary can't produce a valid tag for a ciphertext they didn't receive
  - The probability of a valid forgery is $\mathrm{MAC}_{\mathrm{adv}}[\mathcal{A}, \mathcal{I}]$, which is negligible
- So with overwhelming probability, every decryption query returns `reject`
  - The decryption oracle gives the adversary no useful information!

Step 2: CPA Security Handles the Rest

- With the decryption oracle effectively disabled, the adversary is left with:
  - The public parameters
  - Encryption queries (which they can make freely)
  - Decryption queries that always return `reject` (useless)
- This is functionally equivalent to the CPA game
  - The CPA-secure cipher handles this case
- Formally: $\mathrm{CCA}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}_{\mathrm{EtM}}] \leq \mathrm{CPA}_{\mathrm{adv}}[\mathcal{B}, \mathcal{E}] + \mathrm{MAC}_{\mathrm{adv}}[\mathcal{C}, \mathcal{I}]$
  - Both terms are negligible, so the combined advantage is negligible
- This is why Encrypt-then-MAC is the gold standard for symmetric authenticated encryption

AEAD: Operational Details

AES-GCM internals and nonce misuse resistance.

GCM Structure

- Galois Counter Mode (GCM) is the most widely deployed AEAD cipher
  - Standardized by NIST in 2007; used in TLS 1.3, IPsec, and many other protocols
- GCM follows encrypt-then-MAC internally:
  - Encryption: AES in counter mode (CTR)
  - Authentication: Carter-Wegman MAC using GHASH over GF(2^{128})
- Single key k derives both the encryption key and the authentication key k_m
  - $k_m \leftarrow E(k, 0^{128})$ (encrypt the zero block)

GCM Encryption

- Input: key k, message m, associated data d, and nonce $n \in \{0,1\}^{96}$
- $\text{length}()$ returns a 64-bit value containing the length of the field in bits

$$
\begin{aligned}
&k_m \leftarrow E(k, 0^{128}) \\
&x \leftarrow (n \| 0^{31} \| 1) \in \{0,1\}^{128} \\
&x' \leftarrow x + 1 \\
&c \leftarrow \text{CTR-Encrypt}(k, x', m) \\
&d' \leftarrow \text{ZeroPad}(d, 128), \quad c' \leftarrow \text{ZeroPad}(c, 128) \\
&h \leftarrow \text{GHASH}(k_m,\ d' \| c' \| \text{length}(d) \| \text{length}(c)) \\
&t \leftarrow h \oplus E(k, x) \\
&\textbf{return } (c, t)
\end{aligned}
$$

GHASH

- GHASH is a Carter-Wegman MAC over GF(2^{128})
  - A polynomial hash function, similar to the UHF constructions from Lecture 04
- GF(2^{128}) is a Galois field defined by the irreducible polynomial:
  - $g(X) = X^{128} + X^7 + X^2 + X + 1$
  - Elements are 128-bit bitstrings; addition is XOR, multiplication is mod $g(X)$
- GHASH(k, z) for $z = (z_0, z_1, \ldots, z_{v-1}) \in$ GF(2^{128})^v:
  - $z_0 \cdot k^v + z_1 \cdot k^{v-1} + \ldots + z_{v-1} \cdot k \in$ GF(2^{128})
  - Evaluable incrementally via Horner's method (streaming)
- Modern processors have hardware support for Galois field multiplication
  - AES-GCM is extremely fast with AES-NI and PCLMULQDQ instructions

ChaCha20-Poly1305

- The alternative when AES hardware support is unavailable
  - ChaCha20 (stream cipher from Lecture 02) + Poly1305 (one-time MAC)
  - Combined as an AEAD cipher in RFC 7539
- Poly1305 is a one-time MAC based on polynomial evaluation
  - The same polynomial MAC construction from Lecture 04 (Section A4)
  - One-time key derived from ChaCha20's keystream for each message
- Both AES-GCM and ChaCha20-Poly1305 are available in TLS 1.3

Nonce Misuse Resistance: AES-GCM-SIV

- Standard AES-GCM has no nonce misuse resistance
  - Nonce reuse catastrophically breaks both confidentiality and integrity
  - This was covered in detail in Lecture 04
- AES-GCM-SIV (RFC 8452) provides some protection:
  - The nonce is mixed into the tag computation, and the tag is used as the IV for CTR mode
  - If the nonce repeats but the plaintext differs, the ciphertexts still differ
  - If both nonce and plaintext repeat, it only leaks that the same message was sent twice
  - Authentication is not broken by nonce reuse
- The cost: slightly more complex, slightly slower
  - A worthwhile trade-off when nonce uniqueness cannot be guaranteed

HKDF

Deriving keys from shared secrets.

The Problem

- A raw Diffie-Hellman output $g^{\alpha\beta}$ is not uniformly random
  - It has algebraic structure: it's an element of $\mathbb{G}$, not a random bitstring
  - Using it directly as an AES key would violate the assumptions of the cipher
- Similarly, a KEM output or any high-entropy but structured secret needs conditioning
  - We need a tool that takes non-uniform input and produces uniform-looking keys
- Hash functions aren't designed for this
  - No theoretical basis that SHA-256 extracts uniform randomness
  - Outputs may still leak structure or correlations from the input
- We need a purpose-built key derivation function

HKDF: Extract-and-Expand

- The HMAC-based Extract-and-Expand Key Derivation Function (HKDF)
  - Defined by RFC 5869
  - Derives keys from high-entropy, not necessarily uniform input
  - Not for passwords or passphrases! Their entropy is too low
- Takes a secret s, an optional salt, optional info, and output length L
- Two phases serve different purposes:
  - Extract: remove structure from the input key material
  - Expand: produce keying material of arbitrary length

HKDF: The Algorithm

Extract: $t \leftarrow \mathrm{HMAC}(\mathrm{salt}, s)$

Expand: $q \leftarrow \lceil L / \mathrm{HashLen} \rceil$

$z_0 \leftarrow \varepsilon$

for $i = 1$ **to** q **do:**

$z_i \leftarrow \mathrm{HMAC}(t,\ z_{i-1} \| \mathrm{info} \| \mathrm{octet}(i))$

return first L octets of $z_1 \| \cdots \| z_q$

HKDF in TLS 1.3

- TLS 1.3's key schedule is built entirely on HKDF
  - The ECDH shared secret is the input key material
  - Different info labels derive different keys: client traffic key, server traffic key, handshake key, etc.
  - The salt chains through the handshake: each stage's output salts the next extraction
- HKDF is also used inside DHKEM in HPKE (RFC 9180)
  - Students will encounter this in the EPIC project
- Security comes from combining PRF guarantees with structural separation
  - In the random oracle model, HKDF is provably secure when:
  - The secret is hard to guess
  - The salt is independent of the secret and non-malleable
  - HMAC is modeled as a PRF

Password Hashing

PBKDF2 and Argon2.

The Entropy Gap

- A typical password has around 40 bits of entropy
  - A cryptographic key needs 128 bits or more
- KDFs don't create entropy! They stretch and condition it
  - And add computational cost to slow brute force
- Why not just use HKDF?
  - HKDF assumes high-entropy input
  - Passwords are low-entropy: an adversary can enumerate the entire input space
  - We need computational cost to make enumeration infeasible

Passwords as Keys

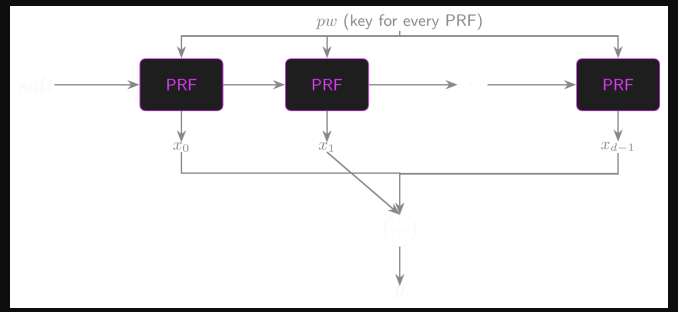
- Uniformly random bitstrings aren't exactly easy to remember...
  - 5FB6 5A5E C29E 17B1 looks like a mess!
- English text has around 1.1 bits of entropy per character
  - Given partial text, you can guess the rest
  - This is a feature for communication, not for cryptographic keys
- Common passwords have been exposed by breaches
  - Dictionary attacks can cover 50% of all passwords
  - SHA-256 hashes crackable in under a minute on a GPU
- Salt prevents precomputed attacks (rainbow tables)
  - A random string stored alongside the hash
  - Forces the attacker into a per-user dictionary attack
  - But doesn't slow down the core computation

PBKDF2

- PBKDF2 is an iterated, salted PBKDF using a PRF (usually HMAC-SHA256)

$$
\begin{aligned}
&\mathrm{PBKDF2}_{\mathrm{PRF}}(pw, salt, d): \\
&\quad x_0 \leftarrow \mathrm{PRF}(pw, salt) \\
&\quad \textbf{for } i = 1, \ldots, d-1: \\
&\quad\quad x_i \leftarrow \mathrm{PRF}(pw, x_{i-1}) \\
&\quad y \leftarrow x_0 \oplus x_1 \oplus \ldots \oplus x_{d-1} \in \mathcal{X} \\
&\quad \textbf{return } y
\end{aligned}
$$

- The loop cannot be parallelised: each iteration depends on the previous one
- OWASP recommends at least 600,000 iterations for PBKDF2-HMAC-SHA256



PBKDF2: Limitation

- PBKDF2 is only compute-hard, not memory-hard
  - A GPU can evaluate PBKDF2 extremely fast in parallel
  - Even with d=10,000, 500 high-end GPUs can crack all 8-character passwords in less than a day!
- We need something that is both compute-hard and memory-hard
  - Forcing the attacker to allocate large amounts of memory limits parallelism
  - This yields a huge slowdown for hardware attacks in practice

Argon2id

- Argon2 won the Password Hashing Competition in 2015
  - An AES-style contest to standardise a new password hashing function
  - Technically a family of three: Argon2i, Argon2d, Argon2id
  - Published attacks on Argon2i; use Argon2id
- Argon2id is memory-hard, parallelizable, and resistant to side-channel attacks
- Three difficulty parameters instead of one:
  - Memory size m: total RAM in kilobytes
  - Iterations t: number of passes over memory
  - Parallelism p: number of parallel threads/lanes
  - Use recommended parameters from OWASP or another reputable source
- Argon2id is the current gold standard for password hashing
  - Tunable parameters allow it to be future-proof

Choosing a Hash Function

- As a general ruleset for picking a password hashing function:
  - Use Argon2id if it's available
  - Use scrypt if you can't use Argon2id
  - Use bcrypt only if you can't use either Argon2id or scrypt
  - If you need FIPS-140 validation, use PBKDF2
- For all of the above, pick appropriate parameters recommended by NIST or OWASP
  - Weak parameters will make your password hashes vulnerable, no matter how good the algorithm
- SecureRide requires password storage for the student authentication path
  - Students must choose and justify a password hashing scheme in their design document

Lab: Authenticated Messaging

Building the EPIC project's core cryptographic stack.

Lab Overview

- Build a small authenticated messaging system in stages
  - Each step corresponds to a concept from Lectures 04-06
  - The final system is a miniature version of the EPIC project's secure messaging component
- The five stages:
  - Stage 1: Generate an ECDH shared secret (Lecture 05: B4-B5)
  - Stage 2: Derive a symmetric key using HKDF (this lecture: D3)
  - Stage 3: Encrypt and authenticate messages using AES-GCM (Lecture 04: A9-A10)
  - Stage 4: Add a signature on each message for non-repudiation (Lecture 06: C4)
  - Stage 5: Swap in PBKDF2 for the password-derived key path (this lecture: D4)

Stage 1: ECDH Shared Secret

- Alice and Bob each generate an X25519 key pair
  - `alice_sk, alice_pk = generate_x25519_keypair()`
  - `bob_sk, bob_pk = generate_x25519_keypair()`
- They exchange public keys and compute the shared secret
  - `shared_secret = x25519(alice_sk, bob_pk)`
  - This should equal `x25519(bob_sk, alice_pk)`
- The shared secret is not uniformly random! It's a point on Curve25519
  - Don't use it directly as an encryption key

Stage 2: HKDF Key Derivation

- Derive a 256-bit symmetric key from the shared secret
  - `key = HKDF(secret=shared_secret, salt=session_id, info="messaging-key", length=32)`
- The salt should be a unique session identifier
  - Could be a hash of both public keys concatenated, or a negotiated session ID
- The info field provides domain separation
  - Different info strings derive different keys from the same secret

Stage 3: AES-GCM Encryption

- Encrypt messages with AES-GCM using the derived key
  - `nonce = generate_random_nonce(12)` (96 bits)
  - `ciphertext, tag = AES_GCM_encrypt(key, nonce, plaintext, associated_data)`
- The associated data should include any metadata that must be authenticated but not encrypted
  - E.g. message sequence number, sender ID, timestamp
- Remember: never reuse a nonce under the same key!

Stage 4: Signatures for Non-Repudiation

- Each message is signed with the sender's ECDSA or Ed25519 key
  - `signature = sign(sender_sk, ciphertext || associated_data)`
  - The recipient verifies: `verify(sender_pk, ciphertext || associated_data, signature)`
- This provides non-repudiation: the recipient can prove to a third party who sent the message
  - MACs alone can't do this (both parties share the key)
- Sign the ciphertext, not the plaintext
  - The signature should cover exactly what was transmitted

Stage 5: Password-Derived Key Path

- Replace the ECDH shared secret with a password-derived key
  - `key = PBKDF2(password, salt, iterations=600000, length=32)`
  - Or use Argon2id for better security
- This models the scenario where Alice and Bob share a password instead of performing key exchange
  - The rest of the pipeline (HKDF, AES-GCM, signatures) stays the same
- Design decisions students make in this lab are the same ones they'll need to justify in the EPIC project design document

Conclusion

What did we learn?

So, what did we learn?

Where do we go from here?

- The theoretical arc is complete:
  - Lecture 04: Message integrity and MACs
  - Lecture 05: Public key cryptography and key exchange
  - Lecture 06: Digital signatures and CCA security
  - Lecture 07: Proofs, AEAD, KDFs, and password hashing
- Focus from here is the EPIC project
  - Full spec will be released on Monday
  - Plan is to run a hybrid model for the remaining sessions
  - Specific discussions about and help with projects
  - Some sessions on useful topics geared towards the EPIC

For next time...

- Get ready to hit the ground running with the EPIC on Monday
  - That's it!

Questions?

Ask now, catch me after class, or email eoin@eoin.ai