Applied Cryptography and Network Security

William Garrison
bill@cs.pitt.edu
6311 Sennott Square

Cryptography Folklore

University of Pittsburgh

Outline

Today, we review Ch. 17: Folklore… essentially, a collection of useful little tips

- Changing encryption keys periodically
- Multiplexing encrypted flows
- Different keys for different purposes
- Hashing passwords
- Generating nonces and other random values
- Compression with cryptography
- Protocol versioning

In short, we'll jump around and discuss each issue briefly

Misconceptions about quantum computing

(We will study quantum computing in more detail soon)

These beliefs are sometimes shared in security:

- A quantum computer can solve any problem much faster than conventional
  - On the contrary, only a few quantum algorithms are known that would outpace equivalent conventional ones
- Quantum computing will break all cryptography
  - In fact, only public-key cryptography would be seriously affected, and cryptographers are looking for new public-key primitives
  - Hashing and symmetric-key cryptography will need (at worst) double-size keys/sizes
- Quantum computers can factor prime numbers
  - This one surprised me… because the factorization of prime p is just p
  - But yes, factoring is a problem where quantum beats conventional

Other cryptography misconceptions

Passwords should expire every so often

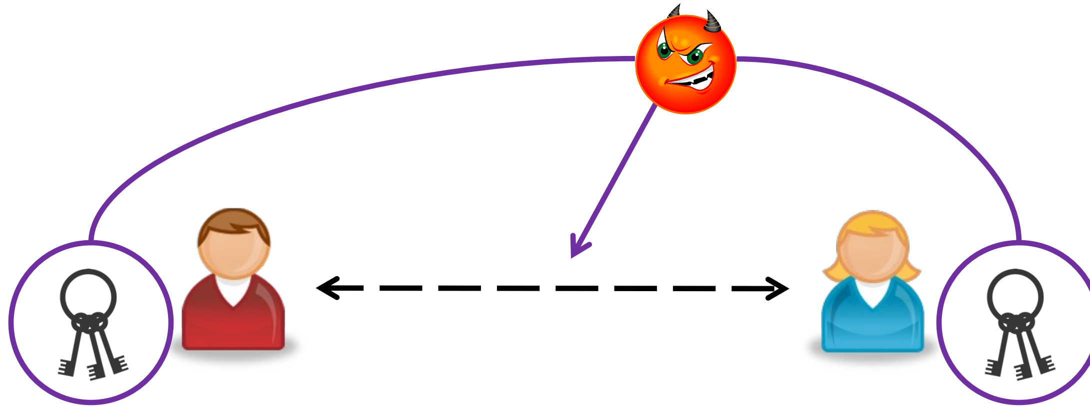
- In reality, data suggests this weakens passwords
- Users with strong passwords probably gain nothing but frustration
- Users with weak passwords probably make them even weaker when forced to change

Any use of MD5 (or ECB or DES or…) is insecure

- It's a good rule of thumb to avoid primitives that don't live up to their original intentions
- However, as always, security is relative
- To say for sure requires a careful study of the "broken" operation's role and the specifics of the known vulnerabilities

Perfect forward secrecy is an increasingly common and important property for handshakes

A key exchange protocol is said to have perfect forward secrecy if an adversary that (i) watches all messages exchanged and (ii) later compromises both parties cannot recover the agreed-upon session key



How do we achieve perfect forward secrecy?

- Generate a temporary secret
- Use only information that is not stored at the node long-term

---

We actually already know a protocol that provides this property!

(Signed) Diffie-Hellman key exchange!

Choose b at random

$[g^b \bmod q]k_B^{-1}$ →

Choose a at random

← $[g^a \bmod q]k_A^{-1}$

Compute $K = g^{ba} \bmod q$
Forget b

Compute $K = g^{ba} \bmod q$
Forget a

$h(\text{Bob} \parallel K)$ →

← $h(\text{Alice} \parallel K)$

← *Session Encrypted with* K →

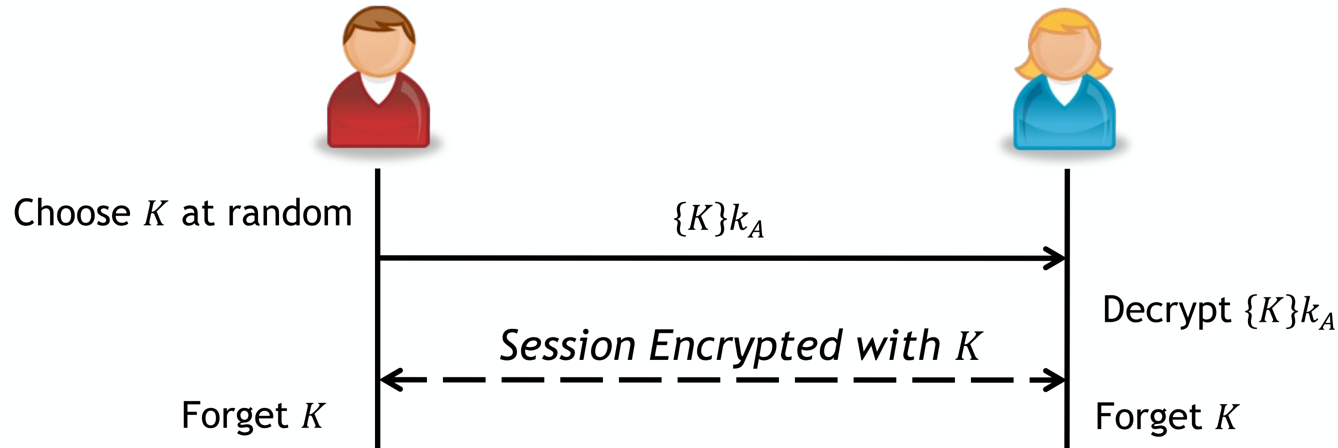
Forget K

Forget K

Question: Why does this protocol provide perfect forward secrecy?

- Adversary learns $g^a \bmod q$ and $g^b \bmod q$ by snooping
- Adversary learns k_B^{-1} and k_A^{-1} by compromise
- The important pieces are a, b, and K
  - These are not stored in the long term

Not all seemingly reasonable key exchange protocols provide perfect forward secrecy



This completely reasonable hybrid cryptosystem also fails to provide perfect forward secrecy (Why?)

What does the adversary learn?

- Monitoring: $\{K\}k_A$
- Compromising Alice and Bob: k_A^{-1} and k_B^{-1}

Given k_A^{-1} and $\{K\}k_A$, the adversary can recover K even though Alice and Bob have both forgotten it!

The previous protocol is similar to the "RSA key exchange" method supported in SSL/TLS

Many TLS cipher suites use RSA key

- TLS_RSA_EXPORT_WITH_RC4_40_MD
- TLS_RSA_WITH_RC4_128_MD5
- TLS_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_IDEA_CBC_SHA
- TLS_RSA_WITH_DES_CBC_SHA
- TLS_RSA_WITH_3DES_EDE_CBC_SHA
- …

Using these cipher suites, one party
the RSA key of the other party

In November 2011, Google added su
using the ECDHE family of key excha

https://mail.google.com/mail/ca/u/0/#

mail.google.com
Identity verified

Permissions | Connection

The identity of this website has been verified by Google Internet Authority G2 but does not have public audit records.

Certificate Information

Your connection to mail.google.com is encrypted with 256-bit encryption.

The connection uses TLS 1.2.

The connection is encrypted and authenticated using CHACHA20_POLY1305 and uses ECDHE_ECDSA as the key exchange mechanism.

Encryption keys should be changed periodically

Keys can be weakened through extensive use

- The threshold depends on the block mode and block size
- Key rollover can be scheduled based on risk level

In CBC for an n-bit block, after $2^{n/2}$ blocks, it is likely to find two identical ciphertexts

- Because of CBC, this is unlikely to mean the plaintexts match
- However, using both ciphertexts, one can acquire the XOR of the two plaintexts

In GCM mode, the 96-bit per-message IV should never be reused

- To avoid the need to keep track of all previous IVs, rekey after $\sim 2^{32}$ messages
  - This holds collisions to less than 1 in a billion

Common misconception: An attacker cannot modify encrypted data without detection

Some block modes include integrity protection, but otherwise this is not true!

- In modes such as CTR, if we can guess the (structure of) the plaintext, we can make targeted changes via XOR
- In other modes, it may be harder to predict the result, but changes can still be made

When can an unpredictable change nonetheless cause issues?

- What if we are transmitting a file and won't check the contents until we download it sometime in the future?
- What if we are transmitting structured requests? How does the receiving parser detect and handle errors?

If we want to detect modifications, we must use integrity protections

Multiplexing flows in a single secure session can lead to splicing attacks

Consider conversations A-to-B and C-to-D, with A and C in one office and B and D in another

- It might be tempting to send A's messages to B, and C's messages to D, in the same session between two local servers

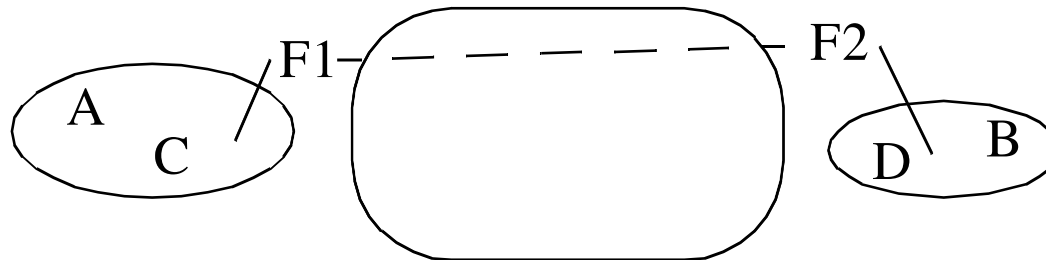


Figure 17-1. Multiplexing A-B and C-D Traffic over One Secure Session

Assume the ciphertext encodes: A-to-B: Msg1; C-to-D: Msg2;

- An active attacker who is also part of one of these conversations can read the other
- Say, D can splice in "C-to-D" over top of the "A-to-B" section and read A's messages to B!

Use different keys for different purposes!

When both encryption and integrity are needed, use different keys for each

- We have seen the problems with same key for CBC-mode encryption and CBC-MAC
- No attacks have been found against most combinations
  - But we know it's possible, and using different keys is an easy mitigation

Recall the reflection attack, where responses from one party are played back to them as if from the other

- One prevention strategy encodes the sender: MAC(k, sender || challenge) rather than MAC(k, challenge)
- Another prevents overlap: Client must send odd challenges, server must send even challenges
- A simple alternative uses different keys for client-to-server and server-to-client
  - Both parties must know both keys, but this very explicitly encodes intended direction

Use different key(-pair)s for different purposes!

Consider using the same keypair for:

- Decrypting a challenge in an authentication protocol
- Decrypting a symmetric-key in the header of a hybrid-encrypted message

What could go wrong in this scenario?

- Consider an eavesdropper who collected the hybrid message, $\{K\}K_b \mid\mid \{M\}K$
- The server is expecting to receive $\{R\}K_b$ and return R
- The eavesdropper can send $\{K\}K_b$ instead: Obtain K!
- How can we prevent this without changing keys?
- Note that similar problems can happen with encrypting and signing

The best solution: Use different keys for these two applications

Rules of thumb when establishing session keys

Ensuring both parties contribute to the key has multiple advantages:

- If one party has a weak or backdoored RNG, the key can still be secure (sufficiently random)
- Neither party's messages can be replayed later, since the key is responsive to both

Examples where they do? Do not?

More generally, do not let either party determine the key

- What if we let each party choose an R, then XOR them to produce the key?
- The second party can choose R_2 after seeing R_1 and force the key of their choice!
- This can (for instance) allow a MITM to go undetected!
- One fix: Use H("my keygen" || R_1 || R_2) rather than $R_1 \oplus R_2$

Wait, why did we insert a constant when hashing the nonces?

This is another good rule of thumb: Add a system-specific constant when generating hash values

- Even assuming the same nonce(s), the resulting keys for two systems will be different
- This is sort of like a per-system salt

Where is this most important?

- Passwords!
- We discussed the use of per-user salt for secure password authentication databases
- Per-system salt can also be useful for password-derived keys
  - e.g., H("my app" || P) rather than H(P)
  - This prevents the use of a lookup table across systems

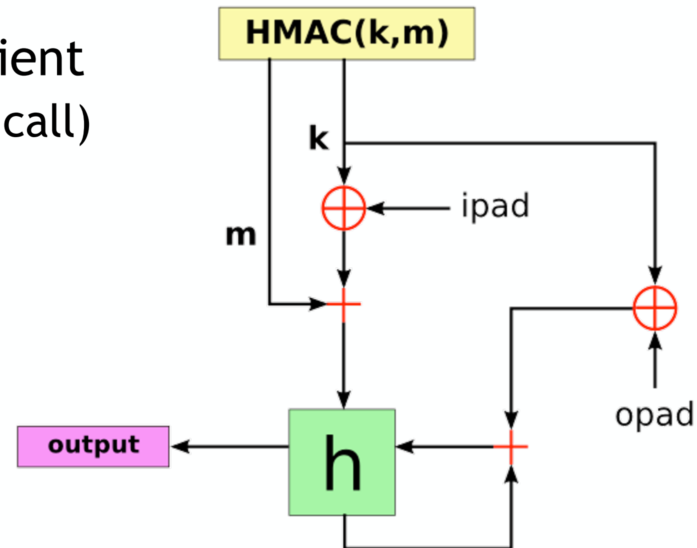
Use HMAC rather than keyed hash

We've discussed extension attacks on MD hashes

- H(k || m) can be extended to H(k || m || o) without knowing k

This same principle arises in challenge-response protocols

- Consider the use of H(R || K) as the response to authentication challenge R
- HMAC was invented to mitigate theoretical attacks against such constructions
- HMAC(K, R) is also very efficient
  - (and is usually a simpler API call)



Compression and encryption

If compression and confidentiality are both desired, compression must be done first

- Compression relies on structure, repetition, predictability
- Encryption obfuscates these features and will minimize compression

Over time, beliefs around compression with encryption has shifted

- Early cryptographers believed compression would improve security, since it would make "plaintexts" less recognizable
- We have since found information leakage from compression
  - If data is organized in fixed-size plaintext blocks, smaller ciphertext indicates higher compression (more repetition)
  - Consider voice chat, where silence compresses better than speech and reveals who is talking when
  - Some attacks have even used compressed size to determine spoken phrases!

Just enough security, or over-design?

Some protocols are designed so every component is necessary

- One divergence from spec yields a vulnerability

It has become more common to see protocols with redundant protections

- Say, n ways of mitigating a threat where only 1 is needed
  - Thus, $n-1$ can be done incorrectly without impact
  - For instance, three random nonces where no harm is caused if at least one is random
- Be cautious about following this approach
  - It's natural to get complacent if your work has no effect!
  - Each nonce might end up being chosen poorly, assuming the others will pick up the slack

Some protocols combine primitives with varying weaknesses

- e.g., a well-studied public-key system that could be broken using quantum computers, with a new and under-studied post-quantum algorithm that might be broken with further study

Checksums and version numbers in protocols

In general, add integrity checks at the end

- When sending, we only need to process the data once
- Checksum at the beginning requires buffering when sending
- Checksum in the middle requires the verifier to remove it

To create forward-compatible protocols, include a version number

- Since fields can be added in later versions, indicate a version number:
  - toward the beginning
  - in a fixed location (that doesn't move between versions)
- Version negotiation can use the latest version both sides support
- Be cautious of attacks that can force a downgrade to a more vulnerable version!

Conclusions

The best practices we discussed today:

- were developed over years of discovering vulnerabilities
- were passed down through generations of cryptographic engineers
- will continue to change over time!

As such, it's important to continue to study (new and old) protocols and attacks against them

Next: TLS, a general-purpose, extensible cryptographic communication protocol