CS 261 Scribe Notes

November 13th, 2008

Scribe: Tavi Nathanson

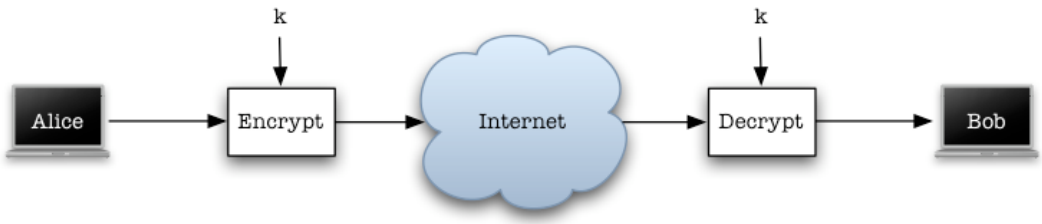
Introduction

- This lecture will be a whirlwind tour of various aspects of cryptography.
- Note: according to Bellovin, only 20% of security problems (or fewer) can be solved with cryptography! Keep that mind.
- Everyone thinks that encryption is the core of cryptography.

- The goal of cryptography is as follows: *I want to communicate securely, even though my communication medium is fundamentally insecure.*
    - Example: I want to communicate securely over the Internet even though it is fundamentally insecure (versus using a secure link, such as a fiber optic cable from base to base).

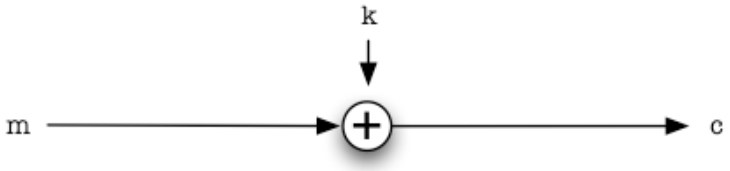
Symmetric-Key Cryptography

- Alice and Bob want to communicate with each other, so they need to have some shared secret *k* that they know and no one else knows (i.e. they got together and exchanged the secret number privately).



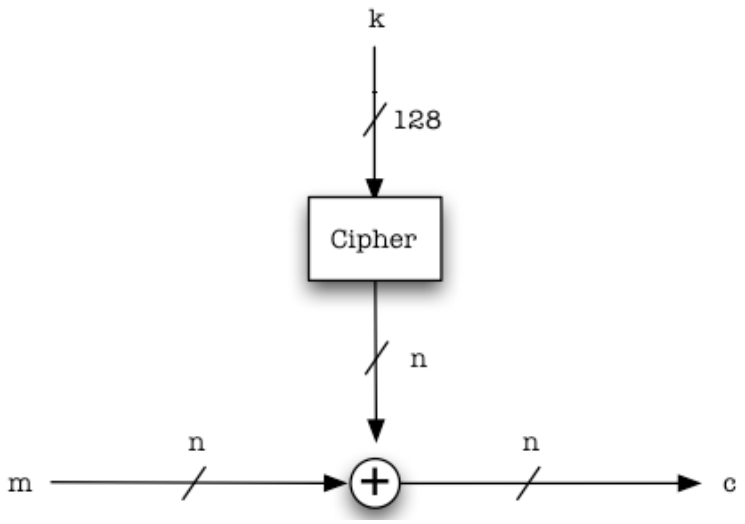
Encryption

- One-time pad (a bitwise exclusive or):



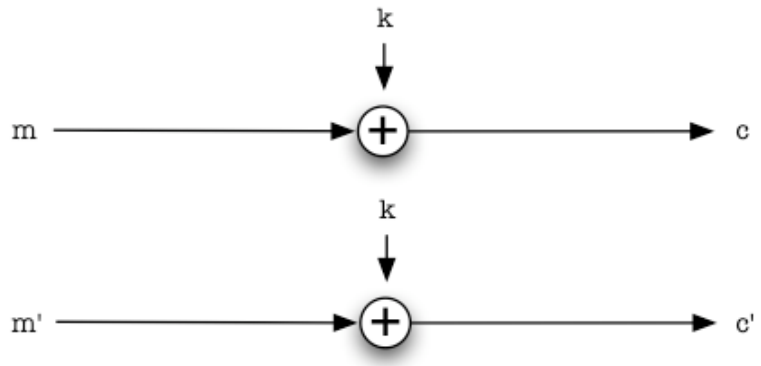
    - Attacker has no way to know what the message is.
    - *Provably* secure (the attacker gets *no* information) under the conditions that the key is only used for one message, and the key is as large as the message.

- Stream cipher:



  - This comes into play because symmetric-key cryptography tries to make use of *one* 128-bit key, versus *many* message-length keys.

  - *Question*: can you hide the fact that you reused the same *k*?
  - *Answer*: no.
  - Example:



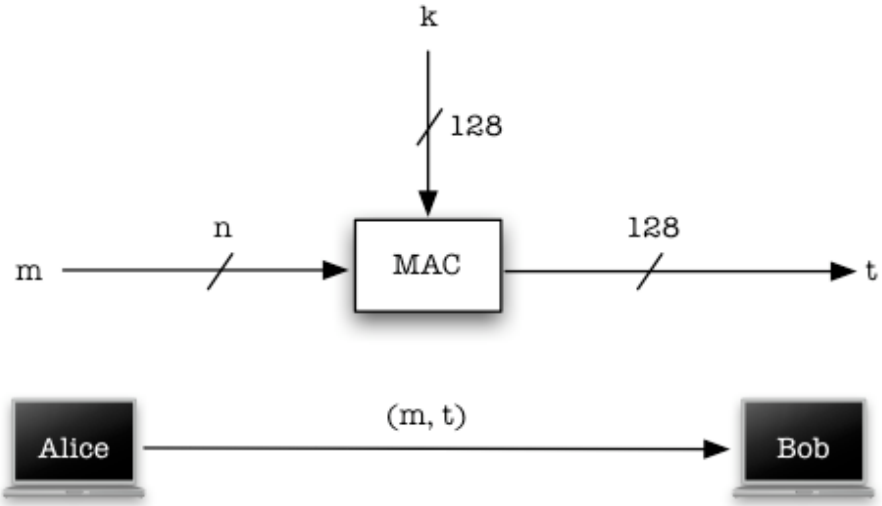
    - c XOR c' = m XOR m'
    - Thus, we can't hide that *k* is reused

  - Known plaintext attack: suppose you are sending a bunch of messages , and it happens that an attacker knows *one* message. From one pair of message/ciphertext, an attacker can recover the key and decrypt the other messages.

- In stream cipher, there are two mechanisms to deal with keystream reuse:
    - The cipher is stateful (the sender and receiver are stateful and synchronized). It is seeded with the 128-bit key; you turn the crank and get more pseudorandom bits every time (until you get as many bits as the length of your message).
    - The cipher can have an initialization vector, where the same key with a different initialization vector gets a different pseudorandom stream. The stream coming out *looks* random (i.e. no algorithm can distinguish the bits from truly random bits), and we know that true randomness is secure (see one-time pad).

Authentication

- Encryption only provides confidentiality: we also want *integrity*. In other words, we want to know that a message came from Bob (and no one else), and it wasn't tampered with.
- We have a MAC (Message Authentication Code) for message authentication in the symmetric-key setting:

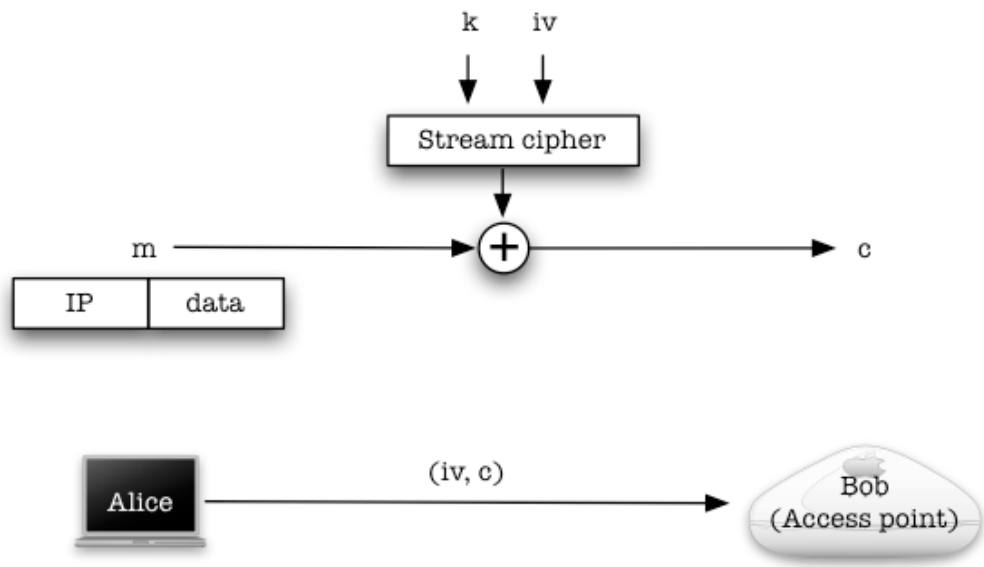


- If an attacker wants to forge a new message, even if he sees a bunch of (m, t) pairs he cannot predict what the valid MAC on the new message is.
- Note: standard notation is $\{M\}_k$, usually indicating encryption and authentication together. Wagner's notation, on the other hand, is:
    - $\{M\}_k$ = authentication
    - $\{[M]\}_k$ = authentication and encryption

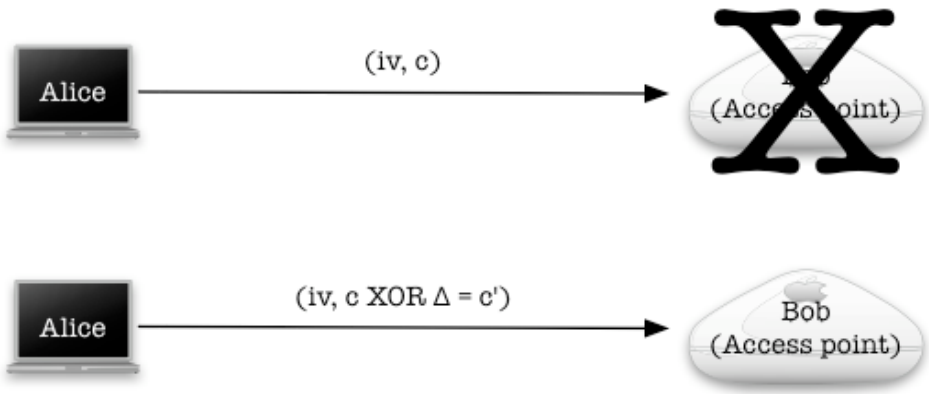
- Note: with a shared key we *first* encrypt, then authenticate.

- The *most common mistake* with symmetric-key cryptography is using an encryption algorithm to prevent eavesdropping, without providing authentication.
    - Encryption without authentication is insecure. Often, an attacker can breach confidentiality by modifying messages in transit (and thus, gaining information about the original messages).

- WEP, the security layer built into 802.11, is a favorite example of failed cryptography. Basically, WEP applies a stream cipher.
    - The EECS department WEP key is shared amongst hundreds of people.
    - WEP is *broken*: even without knowing the key or guessing the key, there are still some attacks that let you read encrypted traffic and introduce forged packets.

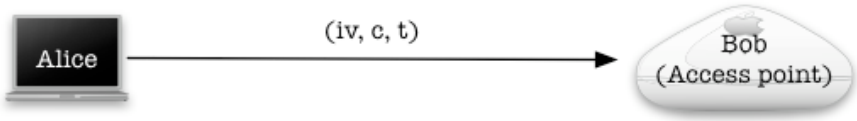


- Attack: imagine that the attacker observes the laptop transmit the ciphertext, prevents Bob from receiving it, and makes some modifications:



- c' XOR k = (c XOR Δ) XOR k = (m XOR k XOR Δ) XOR k = m XOR Δ
- Intuitively: if an attacker flips the 7th bit, then that corresponds to flipping the 7th bit in the message!
- Thus, an attacker can make controlled changes to data.

- An attacker that defeats confidentiality: flip bits in the destination IP address. If he can guess that the packet is going to somewhere else in the EECS network, he can also guess some of the bits. Upon flipping the right bits (after looking at what bits differ between his network address and EECS), the packet will be sent somewhere else.
  - What went wrong? This wasn't an attack on the stream cipher; encryption was perfectly good and the key was never recovered. Somehow the attacker exploited interaction between cryptography and what the recipient was going to do with the message to fool the recipient into doing the wrong thing. There was nothing to protect the integrity of the data, or tampering with the data.
  - Fix: instead of just encrypting the data, also send a checksum:



    - If the checksum is invalid, don't do anything.

  - So, if you want confidentiality, you need authentication!

- Another case where authentication is important: let's say you are using an ATM, and the back office sends a message to the ATM telling it to dispense $300. Imagine if that message was encrypted with a stream cipher but no MAC was used; a nice attack would be for someone who had access to flip bits in the ciphertext (i.e. flip the 20th bit so that you get a huge number + $300). The ATM would start spitting out money, while the back office would think that only $300 was deducted!

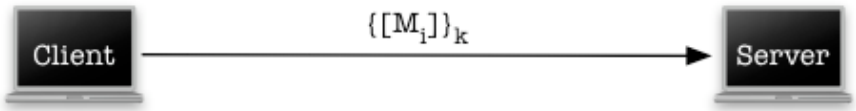
Hash Functions

- A hash function maps an arbitrary length bitstream to a specific-length bitstream. For example: $H: \{0, 1\}^* \rightarrow \{0, 1\}^{160}$

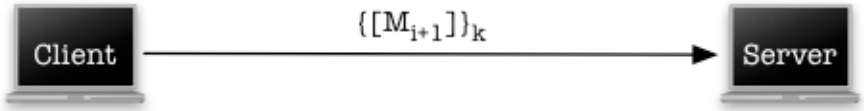
- Cryptographic hash functions have certain properties:
    1. One-way: knowing the hash doesn't reveal what the original bitstream is, or any other bitstream that would hash to that hash.
        - Given $y = H(x)$, it is infeasible to find x' s.t. $H(x') = y$
        - It is also important that you can't find any part of x', although the literature strangely ignores this (at least in the description of the one-way property).
        - This allows hash functions to be used for passwords: store the hash instead of the password itself, and when you log in later the system compares the hash of the password you enter to the hash in the database.
    2. Collision-resistant: it is infeasible to find x, x' s.t. $x \neq x'$ and $H(x) = H(x')$
        - This means that you never have to worry about seeing a pair of messages that hash to the same thing, even though there *must* exist such messages when you map longer bitstreams to shorter bitstreams. You can't find such pairs within your lifetime, though, so you can ignore the risk of that happening and can treat hash functions as perfectly collision-resistant.
        - Example: an email system can store the hash of an attachment; then, if a million people get an attachment, the attachment itself only needs to be stored once.
        - Example: checksum (we can hash two different files to check that they are the same, because of collision-resistance).
    3. "Behave like a random function": i.e. you tweak part of the input and the hash *totally* changes, as if it is a giant truth table.
        - This addresses the leaking of partial information.
        - Knowing the hash for one message won't give away the hash for a very similar message.

Using the Building Blocks

- Applications need a secure channel: a two-way connection that's secure (that has similar security properties to a dedicated link that no adversary has access to).



- Properties:
    - Confidentiality
    - Integrity
    - ~~Availability~~ (cryptography can't provide availability; if an attacker controls a router along your path, you're out of luck).



- In the above setup, the server receives a message authenticated by key *k*; there is a different key *k* with each of its clients, but let's just think about it as if the single server and client are the only two parties in the world.

In what ways does this fall short of the security properties you'd get if the client and server had a dedicated secure link?

1. Replay attacks: an attacker can observe an encrypted message and resend it at a later time.
    - The server might think that the client has just sent that message again.
    - For messages like "spit out $300," this can have grave consequences!

2. Reflection: an attacker can observe a message being sent from client to server, but send it back to the client.
    - Since the client might assume that the server is the only machine that knows keys *k*, it might be fooled into thinking that the message came from the server. Of course, it really came from itself! The client also knows *k*!

3. An attacker router can refuse to forward any packets at all; or more subtly, only refuse to allow some of them through.

- Ex: an attacker might block the 2nd of 3 packets from being sent from client to server; then, if the server works by concatenating those packets under the assumption that they are reliable, he may have changed the semantics by deleting the 2nd packet. In the case of a direct link, on the other hand, you may be able to assume reliable inorder transmission.

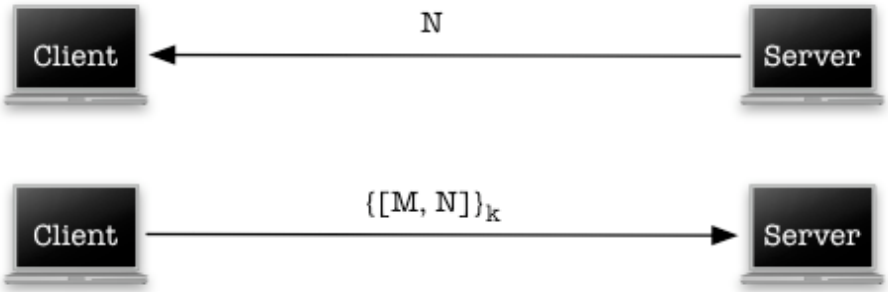
4. Typically the ciphertext is the same length as the message (plus a fixed overhead); that means that an adversary who is an eavesdropper can deduce message lengths.
    - This can be done by determining when a packet was sent and how much data is in it, and could enable information leaks (i.e. this can tell you if a machine is likely to be awake or asleep based on traffic analysis).
    - The following is an attack demonstrating that knowing lengths can hurt:
        - With SSL encrypted traffic, an eavesdropper can observe the lengths of the HTTP GET request and response.
        - Just by observing those lengths, one can deduce what pages the web browser was visiting on the website!
        - In some cases the GET request length discloses the length of the URL, and the length of the response is related to the length of the HTML. Also, one can determine partial information about links (because if the user is on page 1 and goes to page 2, it's likely that there was a link from page 1 to page 2), the number of inline images and how large they are, etc.
    - Another example: when streaming movies, different movies will have different packet sending patterns (i.e. times when there is a lot of data being transmitted and times when there isn't). Thus, we can identify movies based on packet lengths!

Solutions

1. Replay attacks:
    - Add a sequence number to packets or a little state to the endpoints in order to ensure that the packet is not one you've already seen.
    - We can also use timestamps: the client adds the current timestamp to a message, and server looks to see if the timestamp is very old upon receiving the message.
        - If the timestamp is relatively recent and the underlying protocol provides inorder delivery, it can just check that timestamps are increasing.
        - If out-of-order delivery is possible, the server can remember the timestamps of all the packets seen recently

(say, the past 5 minutes) and check to see if it's been seen already. Beyond those 5 minutes, the server can simply reject the packet.

- Or we can use a nonce / call-response:



    - The server will only accept a message if the nonce in the encrypted message messages the nonce it sent to the client.

2. Reflection:
    - The source and destination addresses can be included in the message.
    - If we want to minimize space in messages, we can optimize the above with a "direction bit" (a single bit that differentiates the source to destination direction from the opposite).
    - Or, we can use a different key in each direction.

3. Drop messages:
    - Sequence numbers allow the receiver to recognize the dropping of messages, so it can insist on retransmission.
    - Since availability cannot be provided, the best that can be done is ensuring that the sequence of messages received at the receiver is a *prefix* of the messages sent at the sender. In other words, the messages are in the same order, but nothing is received beyond the point of a sequence number gap.

4. Traffic analysis:
    - Make all packets one size via padding.
    - Can also handle timing: for example, once every microsecond, *always* send a packet (and use the contents of the packet to indicate whether a packet actually needed to be sent).
        - Of course, this is a horrific bandwidth hog! In fact, the reason networks work efficiently is because people are normally only using a small fraction of them. Thus, not only is this a bandwidth hog, but networks would actually die.

    - Random padding? The problem with random padding is that it's potentially susceptible to statistical analysis, as the length of the ciphertext is correlated with the length of the message. An attacker might be able to distinguish the signal from the noise.

Conclusion

- Now we know how to build a secure channel used shared keys at two endpoints, although we probably want to use existing libraries instead of building our own secure channels!

- What's also useful is establishing a *new* secure channel, throwing away the key when done.
- If I have a secure channel to you and you have a secure channel to your friend, can you imagine using that information to allow for me to have a shared key with your friend?
    - Keep an eye on that when reading the Kerberos paper. Basically, they are building a secure channel.

- Who wants to discuss public-key cryptography?