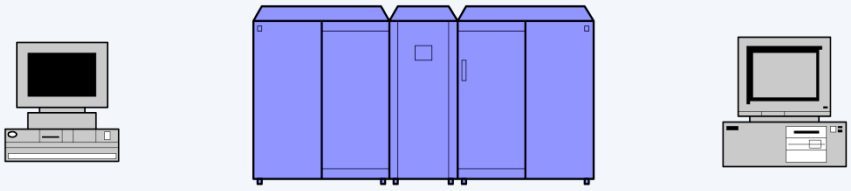
CS 138: Security

Check out other courses!

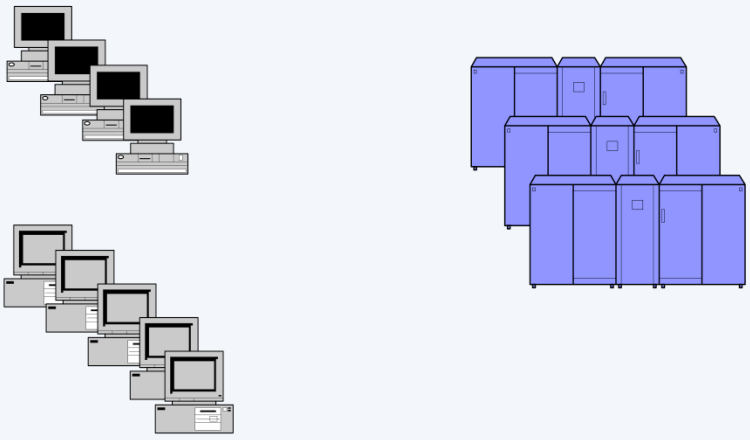
- CS166 – Introduction to Computer Systems Security
- CS151 – Introduction to Cryptography and Computer Security

Time-Sharing Systems

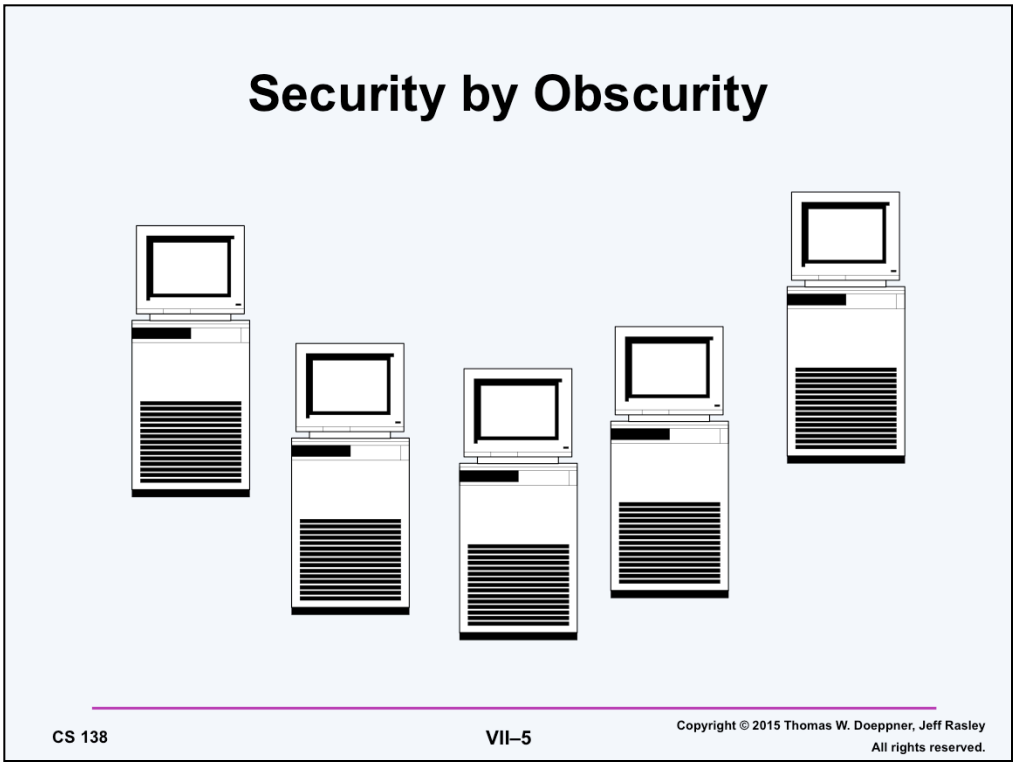


- Time-sharing systems of the ’60s and ’70s.
- These machines had no real networking, security concerns constrained to individual computers.
- One identified oneself to the system by responding to the login prompt, and then proved one’s identity by supplying a password.
- Assuming the OS was secure this was a reasonably secure means for controlling access to the system.
- An attacker may make multiple password attempts but this would alert a sysadmin

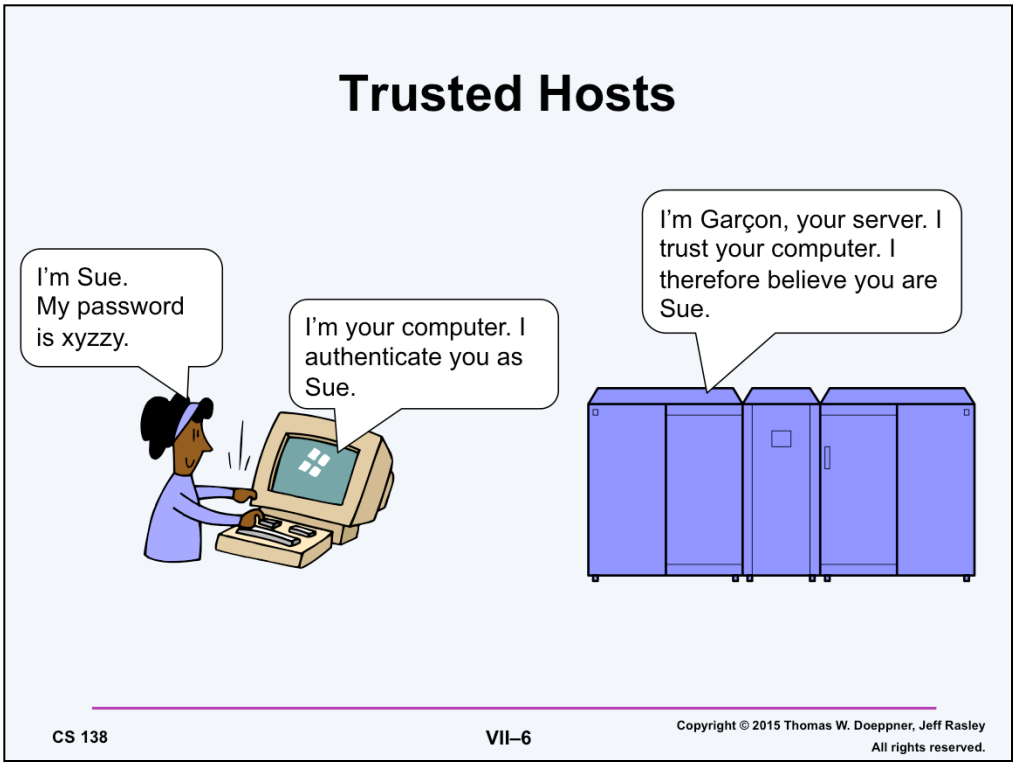
Networks



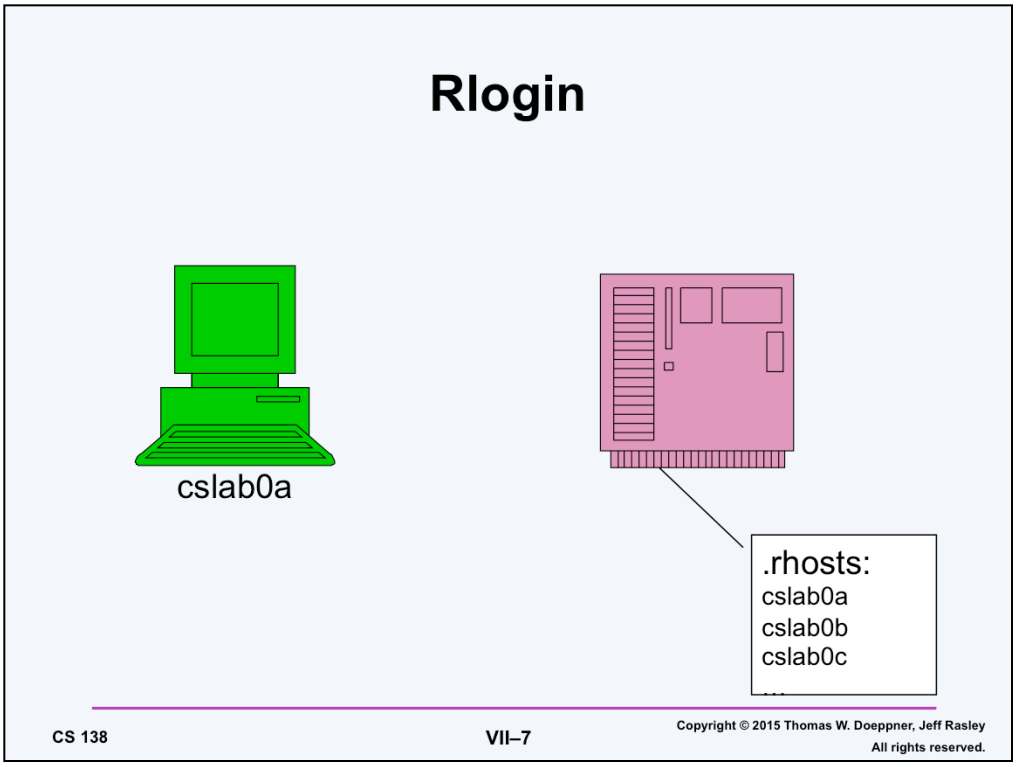
In the ’80s networks of computers started to become numerous. Pioneering work at UC Berkeley and elsewhere led to Berkeley Unix, its support of networked Unix machines, and the initial growth of the Internet.



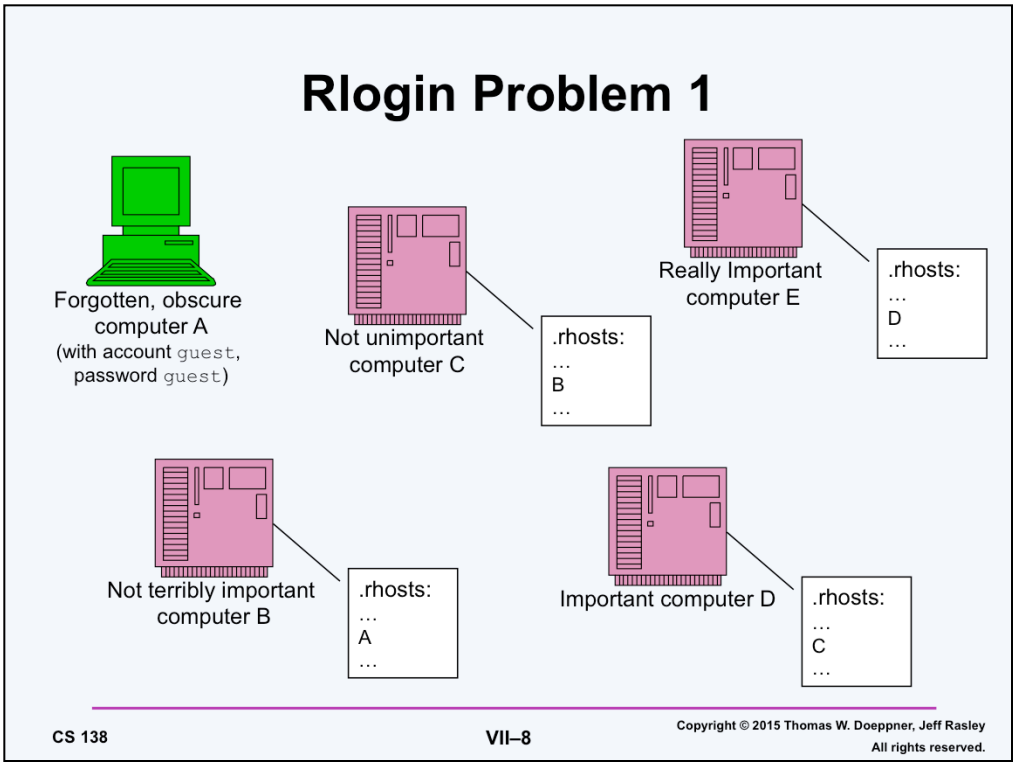
- Also in the ’80s was the commercialization of the computer workstation.
- Collections of computers were being made to appear as if they were one large time-sharing system.
- Security now started to become an issue — how could time-sharing security be provided across a network?
- One approach took advantage of the fact that not just anyone could connect to the network — the software and hardware required was proprietary.
- Problems here relate to reverse engineering and leaks



The "trusted hosts" approach to authentication was pioneered in early versions of Berkeley Unix. A server would have a list of clients it trusted, meaning that if it received a request from a user on such a trusted client, it would assume the user had been properly authenticated on that client and wouldn't require any further validation of the user.



Each host would maintain a .rhosts file containing the list of trusted hosts. (Not shown here, but the .rhosts file could also specify that only certain users on a remote host would be trusted.)



1986 @ Stanford: Someone gained access to A, then found a way to get superuser access on that machine (via a separate issue), they then changed their UID to a user that had access to B and hopped onto B … then hopped around finally to E.

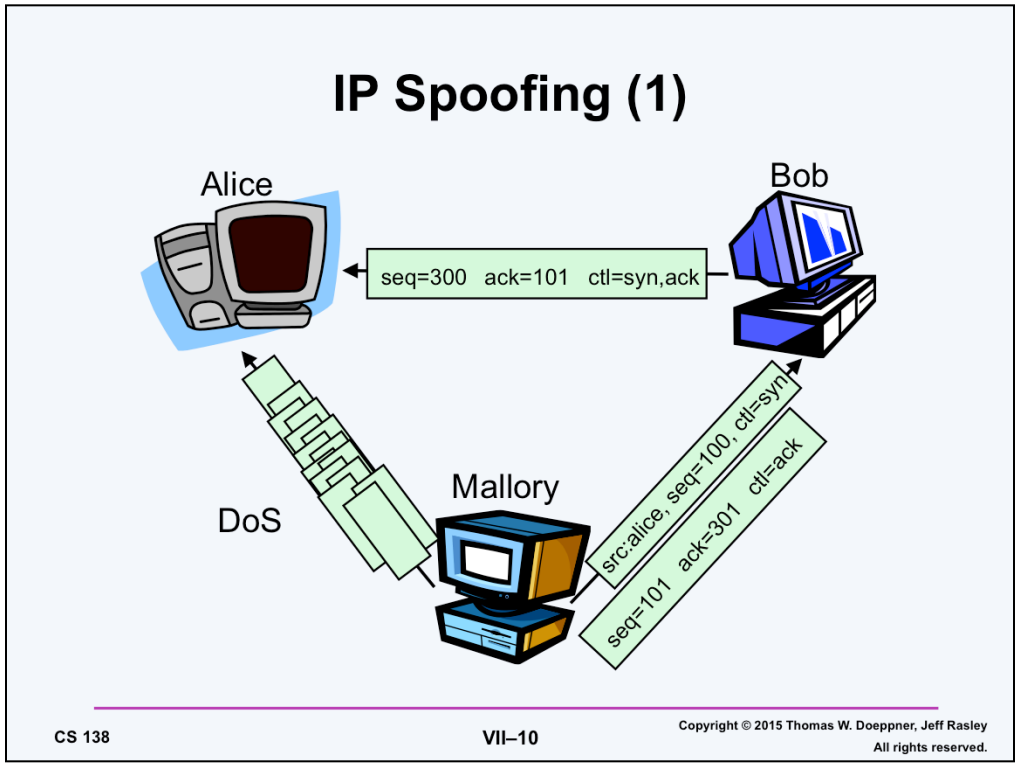
The slide illustrates a problem that actually did occur in September 1986. Someone was able to login to computer at Stanford that had a “guest” account with password “guest”. Once logged in, they gained superuser access by taking advantage of a common configuration bug in which a directory containing scripts used by the “at” daemon (which runs commands set up to be run at a later time (see “man at”)) was writable by everyone. Furthermore, the path used by superuser for finding commands contained the current directory (“.”). Taking advantage of superuser privilege, the attacker changed his/her UID to a user who was allowed to rlogin to machine B. Once on machine B, the attacker was able to rlogin to machine C, etc. Fortunately it appeared that the attacker’s goal was to see how many computer’s she/he could break into, and apparently no harm was done.

Rlogin Problem 2

- .rhosts file contains
  cslab4d.cs.brown.edu
- How do we know a request comes from that machine?
  - its IP address is 128.148.38.231
- Can it be forged?
  - yes!
    - IP spoofing (next two slides)
    - ARP spoofing
    - MAC cloning

Another major problem with the trusted hosts approach is that it's based on the assumption that the host a client request comes from can be reliably determined by the server.

This is definitely not the case, since the host name is determined strictly by the IP address, which is supplied by the caller in the IP header. Among the ways for this to be forged are IP spoofing (as described in the next two slides) as well as ARP spoofing and MAC cloning. The latter two approaches work only if one is on the same subnet as the machine whose IP address is being forged. ARP spoofing entails convincing the server's ARP (address resolution protocol, which maps IP addresses to MAC addresses) that the attacker's MAC address should be associated with the IP address of the trusted host. MAC cloning involves changing the MAC address on the attacker's machine so that it's the same as that of the trusted host. Both approaches work best if the trusted host is down.

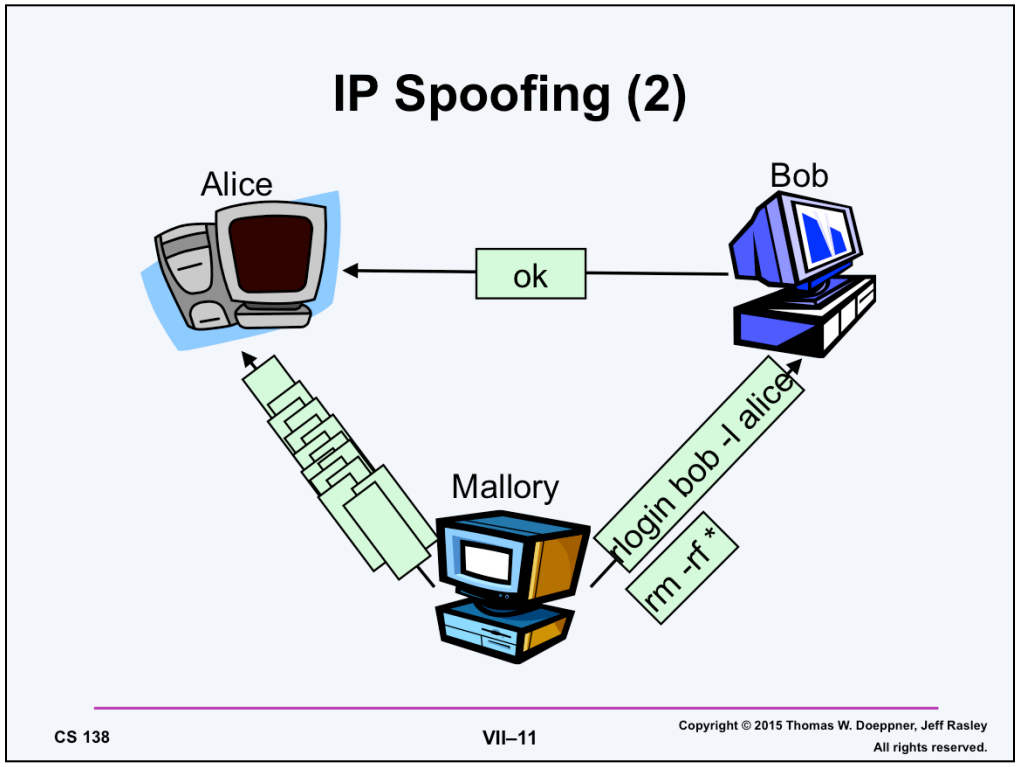


IP spoofing is an attack in which one sends packets with a forged source IP address. Here's an example of an application with TCP. Alice and Bob have computers (named after themselves) that often communicate with each other using the trusted hosts approach. Mallory wishes to attack Bob by sending his computer packets that appear to come from Alice's computer. However, to make certain that Alice doesn't get involved, Mallory starts bombarding Alice's computer with a constant stream of packets that keep Alice's computer so occupied that it can't deal with anything else.

While the attack on Alice is underway, Mallory sends a SYN packet to Bob, but with its source IP address set to Alice's computer's address. Bob responds by sending an ACK along with his own SYN and initial sequence number to Alice. However, Alice ignores it—her machine is too busy coping with Mallory's attack.

Mallory does not, of course, receive Bob's response to his SYN and thus does not receive Bob's initial sequence number. However, Mallory somehow guesses Bob's initial sequence number and sends Bob an ACK (with Alice's IP address as the source). Thus Bob feels that Alice has made a TCP connection to his computer, though Alice has no knowledge of it. Mallory, on the other hand, can use the connection, since she knows what the next sequence number expected by Bob is.

Keep in mind that Mallory is not getting any form of response from Bob; she can only hope that her guess is correct. However, Mallory may know enough about Bob's TCP implementation and the current state of its initial sequence number generator so that guessing an initial sequence number in a small number of tries is not unlikely. For example, Mallory may have made a few legitimate connections to Bob so that she knows how quickly Bob's initial sequence number advances.



Mallory takes advantage of the connection that Bob thinks is with Alice: she sends Bob an "rlogin" request, asking to log in to Bob as Alice. Bob's computer, thinking that this request is coming directly from Alice's computer, accepts the request without asking for proof of who the caller is. Mallory may now send to Bob any command she wishes, and have it executed using Alice's user ID.

Basic Security Requirements

- **Authentication:** Who is this actor?
  - Spoofing, phishing
- **Authorization:** is actor allowed to do this action?
  - Access controls
- **Confidentiality:** Can adversary read the data?
  - Sniffing, man-in-the-middle
- **Integrity:** Do messages arrive in original form?
- **Availability:** Will the network deliver data?
  - Infrastructure compromise, DDoS
- **Replay Protection:** Can adversary replay messages?

Other Desirable Security Properties

- **Audit/Forensics:** what occurred in the past?
  - A broader notion of accountability/attribution
- **Appropriate use:** is action consistent with policy?
  - E.g., no spam; no games during business hours; etc.
- **Anonymity:** can someone tell I sent this message?

Basic Forms of Cryptography

Confidentiality through Cryptography

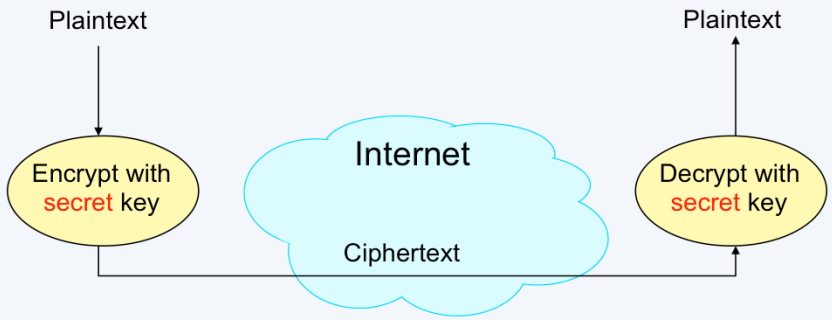
- **Cryptography:** *communication over insecure channel in the presence of adversaries*
- **Studied for thousands of years**
- **Central goal:** how to encode information so that an adversary can't extract it …but a friend can
- **General premise:** a ***key*** is required for decoding
  - **Give it to friends, keep it away from attackers**
- **Two different categories of encryption**
  - **Symmetric: efficient, requires key distribution**
  - **Asymmetric (Public Key): computationally expensive, but no key distribution problem**

Symmetric Key Encryption

- Same key for encryption and decryption
  - Both sender and receiver know key
  - But adversary does not know key
- For communication, problem is key distribution
  - How do the parties (secretly) agree on the key?
- What can you do with a huge key? One-time pad
  - Huge key of random bits
- To encrypt/decrypt: just XOR with the key!
  - Provably secure! …. provided:
    - You never reuse the key … and it really is random/unpredictable
  - Spies actually use these

Using Symmetric Keys

- **Both the sender and the receiver use the same secret keys**

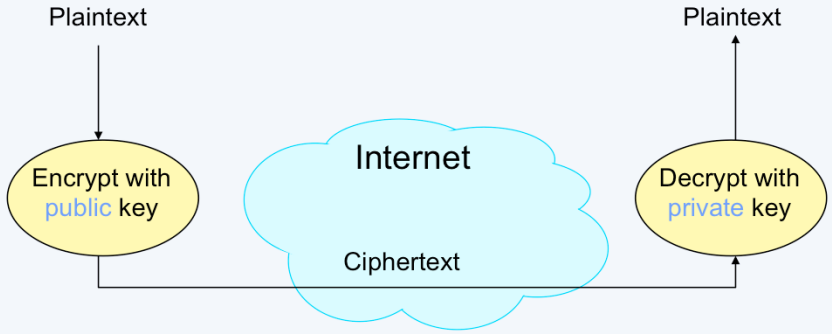


Asymmetric Encryption *(Public Key)*

- Idea: use two *different* keys, one to encrypt (*e*) and one to decrypt (*d*)
  - A key pair
- Crucial property: knowing *e* does not give away *d*
- Therefore *e* can be public: everyone knows it!
- If Alice wants to send to Bob, she fetches Bob's public key (say from Bob's home page) and encrypts with it
  - Alice can't decrypt what she's sending to Bob …
  - … but then, neither can anyone else (except Bob)

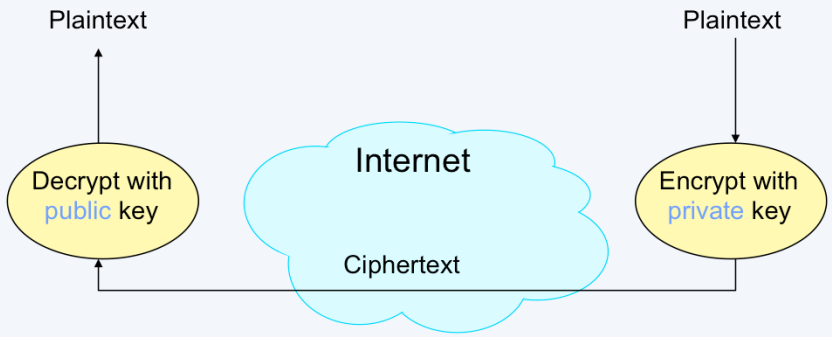
Public Key / Asymmetric Encryption

- Sender uses receiver's public key
  - Advertised to everyone
- Receiver uses complementary private key
  - Must be kept secret



Works in Reverse Direction Too!

- Sender uses his own private key
- Receiver uses complementary public key
- Allows sender to prove he knows private key



Realizing Public Key Cryptography

- Invented in the 1970s
  - *Revolutionized* cryptography
  - (Was actually invented earlier by British intelligence)
- How can we construct an encryption/decryption algorithm with public/private properties?
  - Answer: Number Theory
- Most adopted approach: RSA
  - Rivest / Shamir / Adleman, 1977; RFC 3447
  - Based on modular multiplication of very large integers
  - Very widely used (e.g., SSL/TLS for `https`)
- Other schemes exist, e.g., elliptic curve cryptography

Cryptographic Toolkit

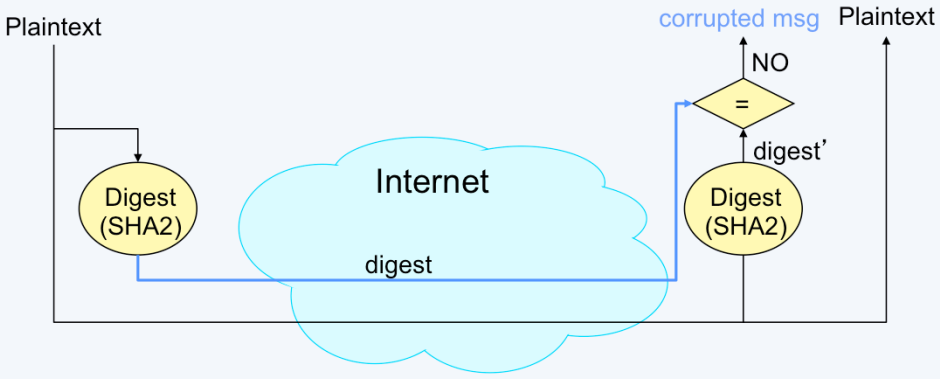
Cryptographic Toolkit

- **Confidentiality:** Encryption
- **Integrity:** ?
- **Authentication:** ?
- **Provenance:** ?

Integrity: Cryptographic Hashes

- Sender computes a *digest* of message *m*, i.e., *H(m)*
  - H() is a publicly known *hash function*
- Send *m* in any manner
- Send digest *d* = *H(m)* to receiver in a secure way:
  - Using another physical channel
  - Using encryption *(why does this help?)*
- Upon receiving *m* and *d*, receiver re-computes *H(m)* to see whether result agrees with *d*

Operation of Hashing for Integrity

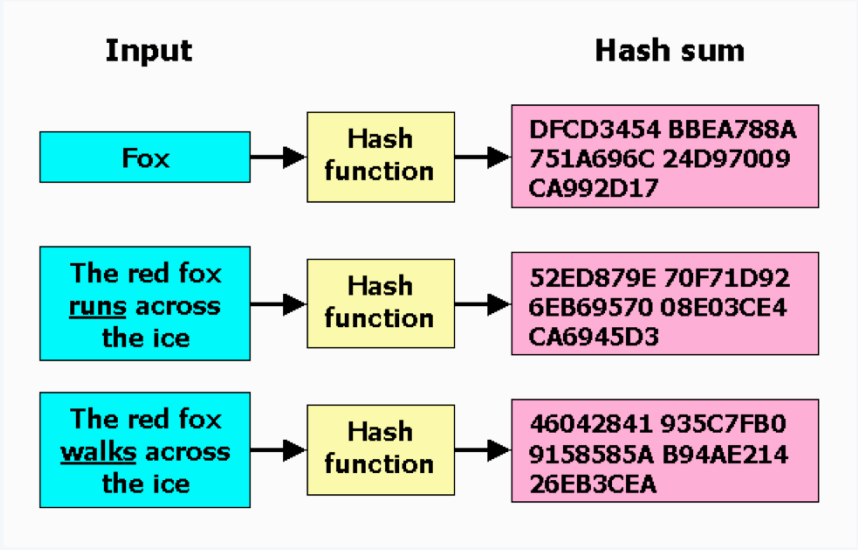


Cryptographically Strong Hashes

- Hard to find collisions
  - Adversary can't find two inputs that produce same hash
  - Someone cannot alter message without modifying digest
  - Can succinctly refer to large objects
- Hard to invert
  - Given hash, adversary can't find input that produces it
  - Can refer obliquely to private objects (e.g., passwords)
    - Send hash of object rather than object itself

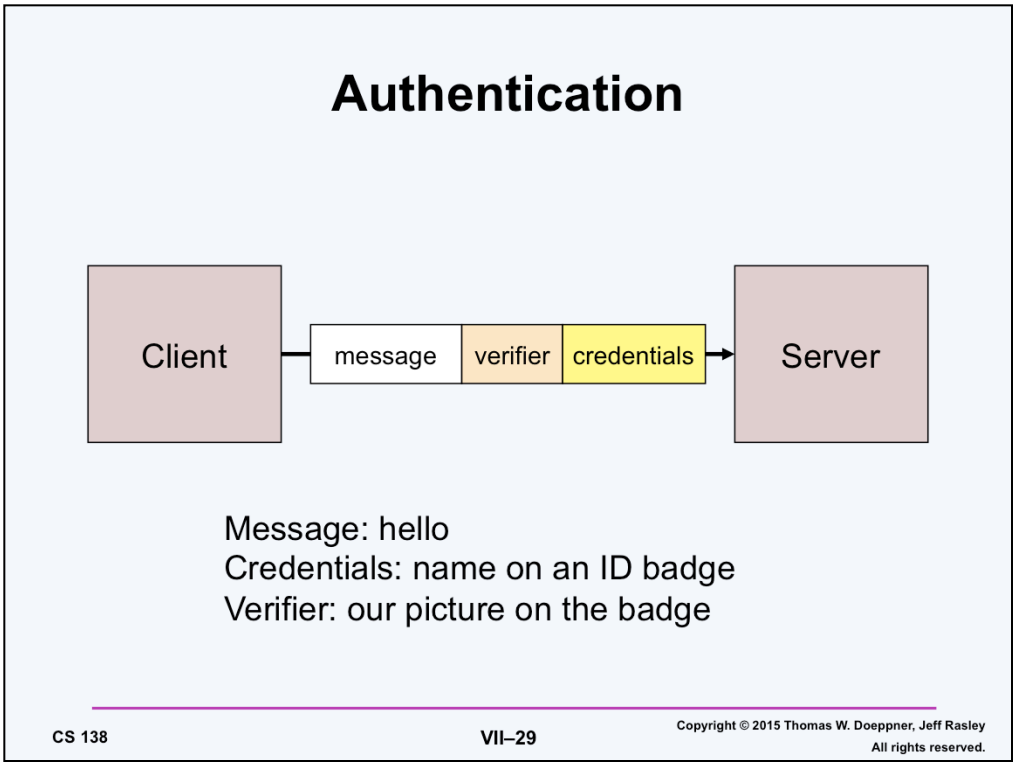
Properties

Effects of Cryptographic Hashing

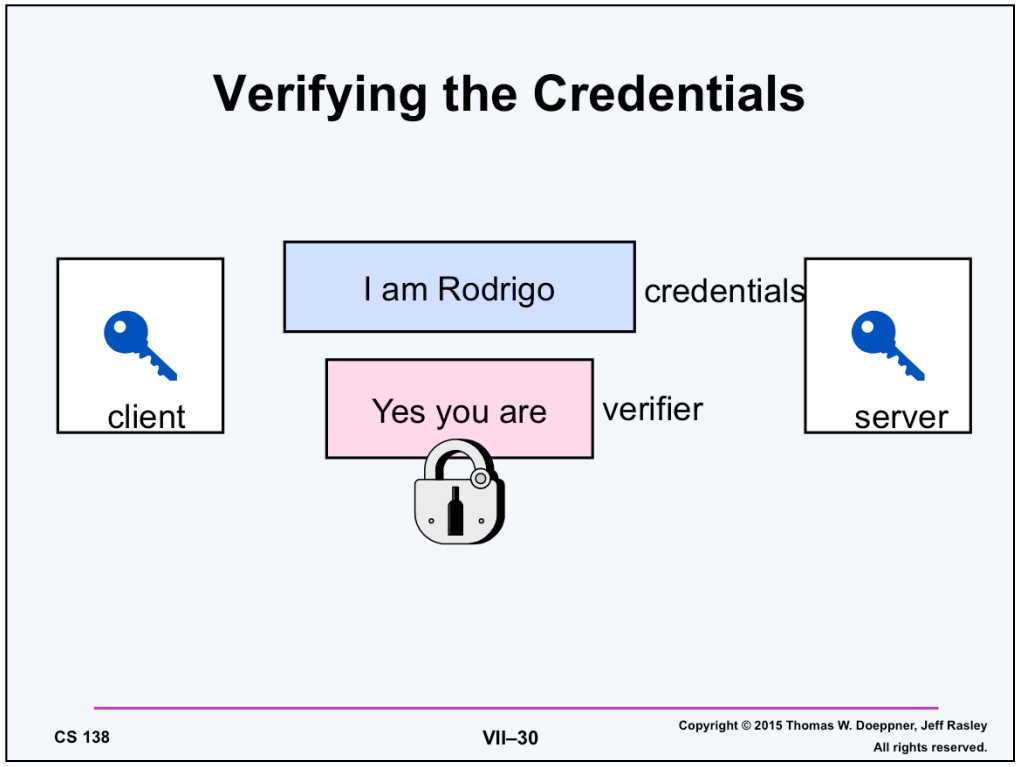


Cryptographic Toolkit

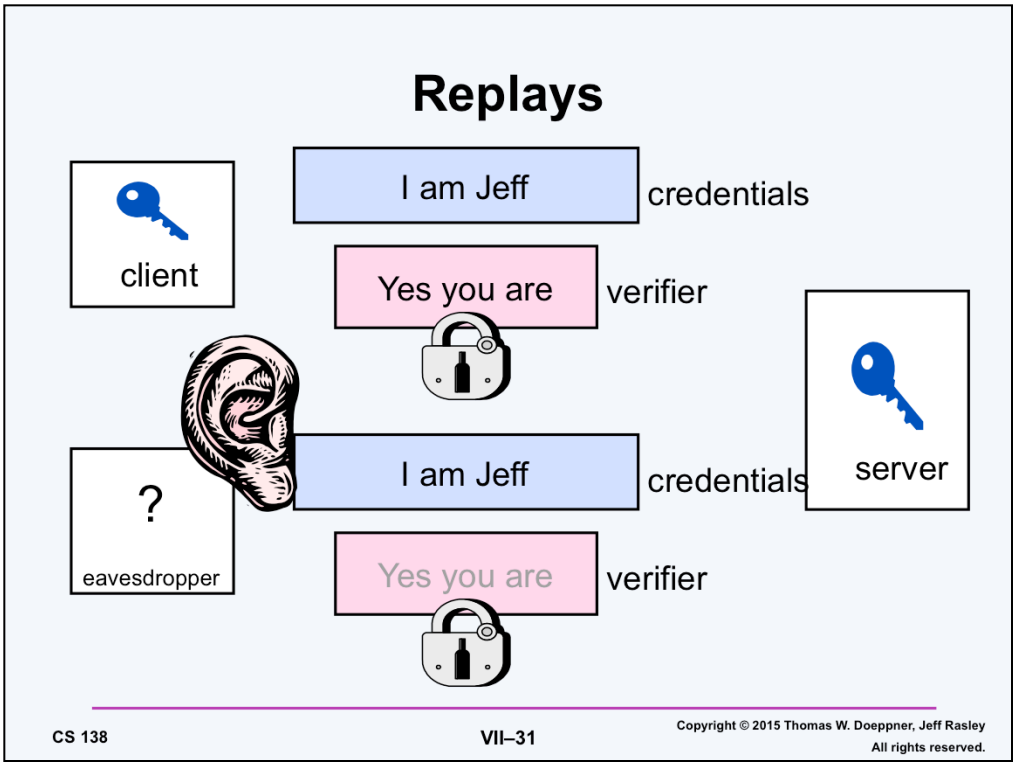
- Confidentiality: Encryption
- Integrity: Cryptographic Hash
- Authentication: ?
- Provenance: ?



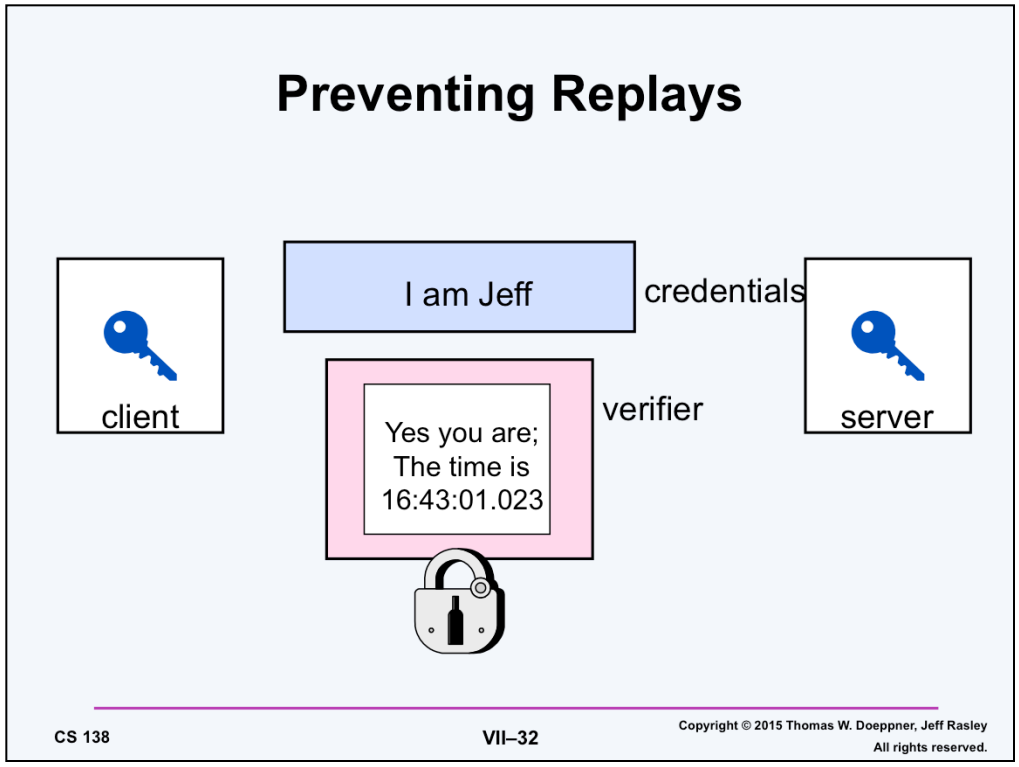
How do we handle authentication? The approach is that we provide, along with the message, two pieces of information. The first is our *credentials*, which asserts our identity. The second is some sort of *verifier* that proves that the credentials really are ours. For example, our credentials might be a name on a badge, while the verifier is our picture (which is also on the badge).



A slightly more computer-related analogy is shown here. Our credentials are a string of text. The verifier is something that proves this, but is locked up in a box, for which only the two communicating parties have a key. To turn this into something that really is computer-related, think of locking the box as encrypting its contents, using an encryption key known only by the communicators.



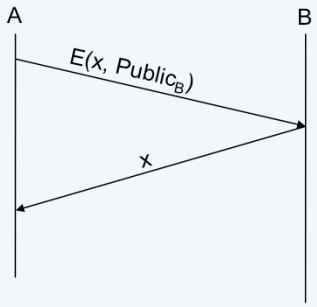
A potential problem occurs if there's an eavesdropper who can make copies of messages, along with the credentials and verifiers. Even though the eavesdropper cannot decrypt the verifier, it could send it to the server along with its own message and make the server believe that the new message came from the original, properly authenticated client.

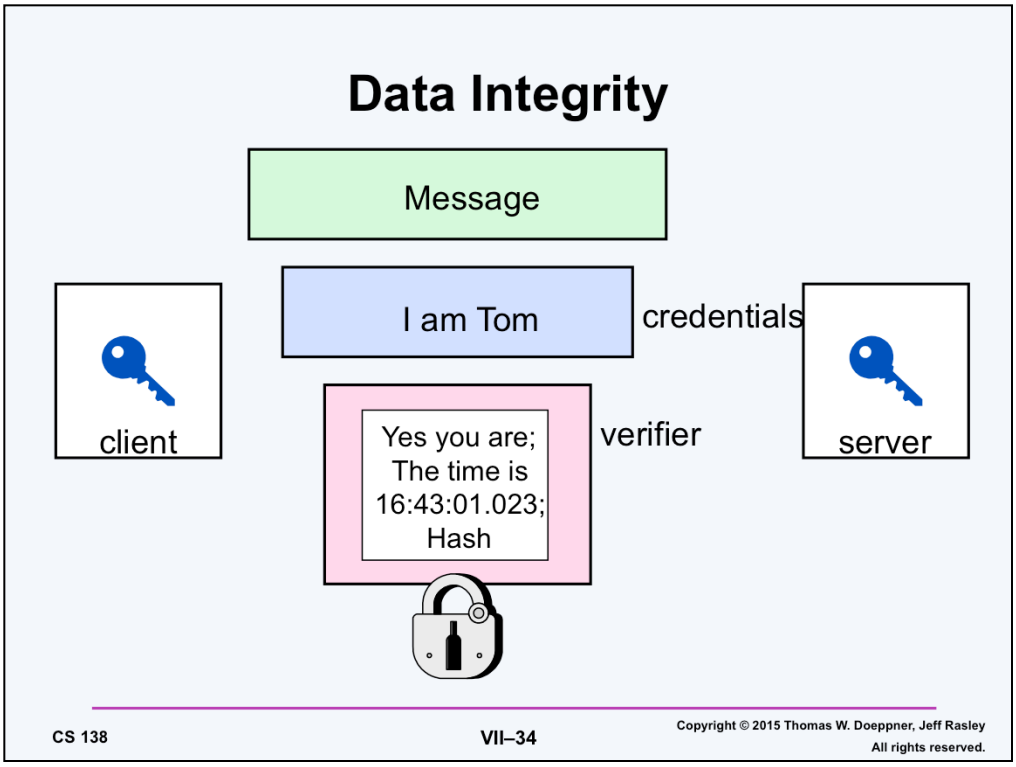


This eavesdropping problem is easily dealt with by including something within the message that the receiver can use to detect that the item has been retransmitted. A possible choice here is the time of day. The receiver holds on to recently received messages; it rejects those whose time stamps are too old and those it has seen recently. Of course, for this to work, the clocks of the sender and receiver must be reasonably in sync.

Public Key Authentication

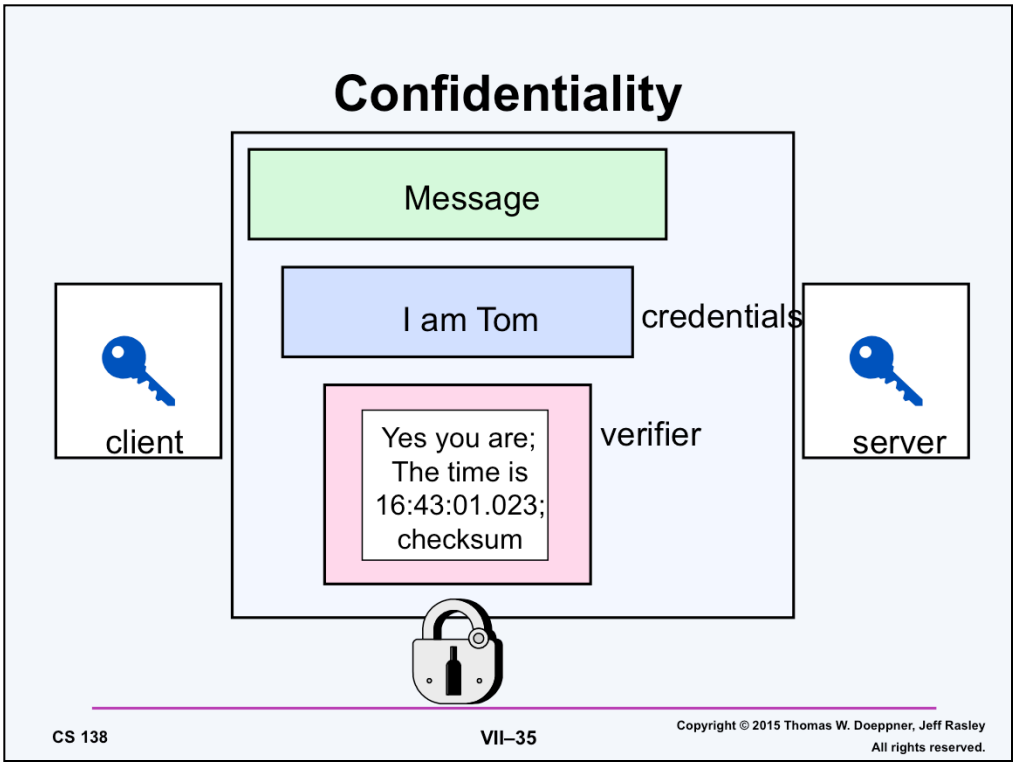
- Each side need only to know the other side's public key
  - No secret key need be shared
- A encrypts a nonce (random number) *x* using B's public key
- B proves it can recover *x*
- A can authenticate itself to B in the same way





Another issue is data integrity (i.e., assurance that a message hasn't been tampered with). This can be provided through the use of a one-way hash function, such as SHA-1: one simply supplies the hash of message along with the message. To make certain that an attacker can't simply modify the hash as well, what is supplied is not simply the hash of the message, but either the encrypted hash of the message (using the secret key) or the hash of the concatenation of the message and the key.

When the hash is based on both the message contents and a secret key, it's known as a *Message Authentication Code* (MAC) and sometimes as a *Message Integrity Code* (MIC).



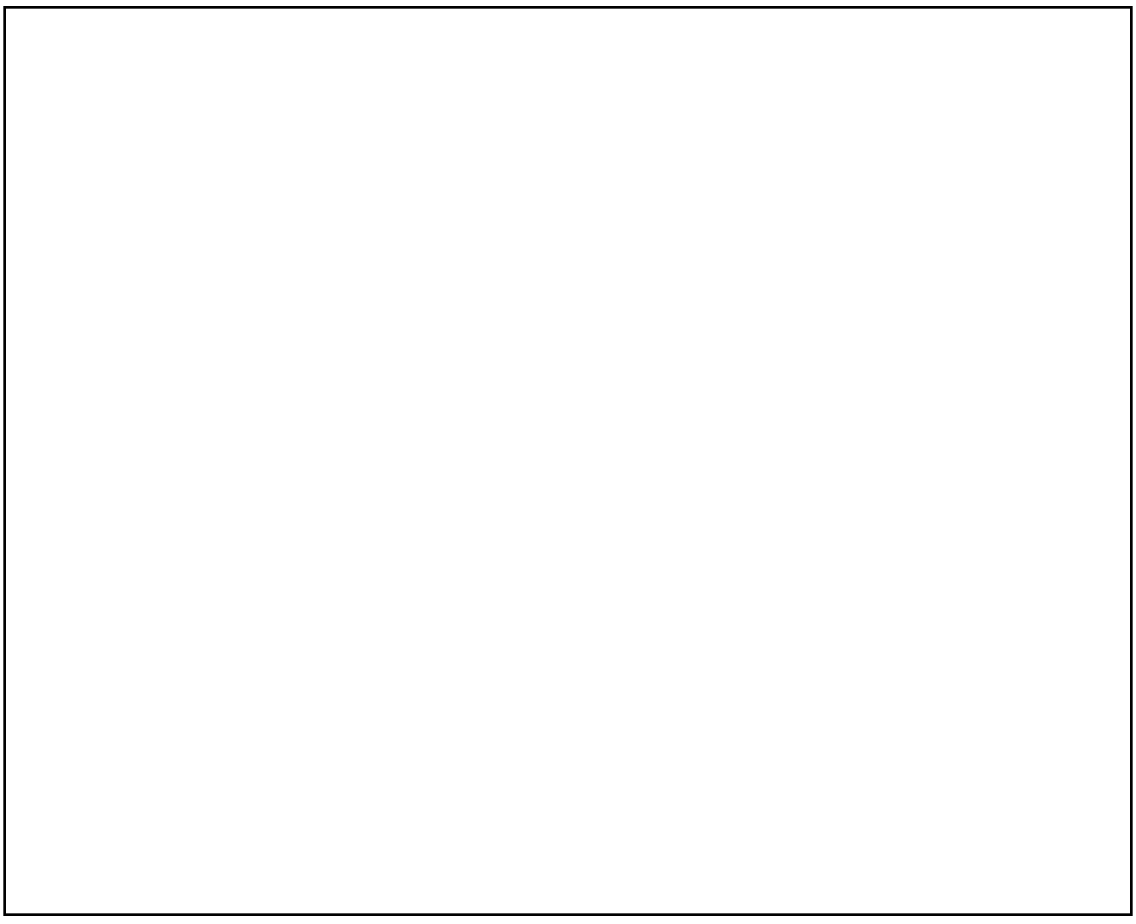
The ultimate step is confidentiality. Providing this is straightforward—everything is encrypted.

Cryptographic Toolkit

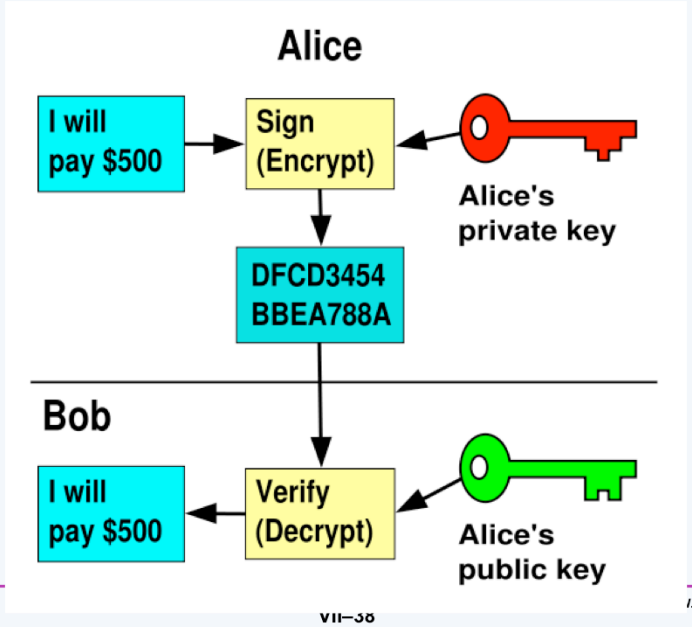
- **Confidentiality:** Encryption
- **Integrity:** Cryptographic Hash
- **Authentication:** Encrypted nonce, Verifier
- **Provenance:** ?

Digital Signatures

- Suppose Alice has published public key K_E
- If she wishes to prove who she is, she can send a message *x* encrypted with her private key K_D
  - Therefore: anyone w/ public key K_E can recover *x*, verify that Alice must have sent the message
  - It provides a digital signature
  - Alice can't deny later deny it ⇒ non-repudiation



RSA Crypto & Signatures, con't

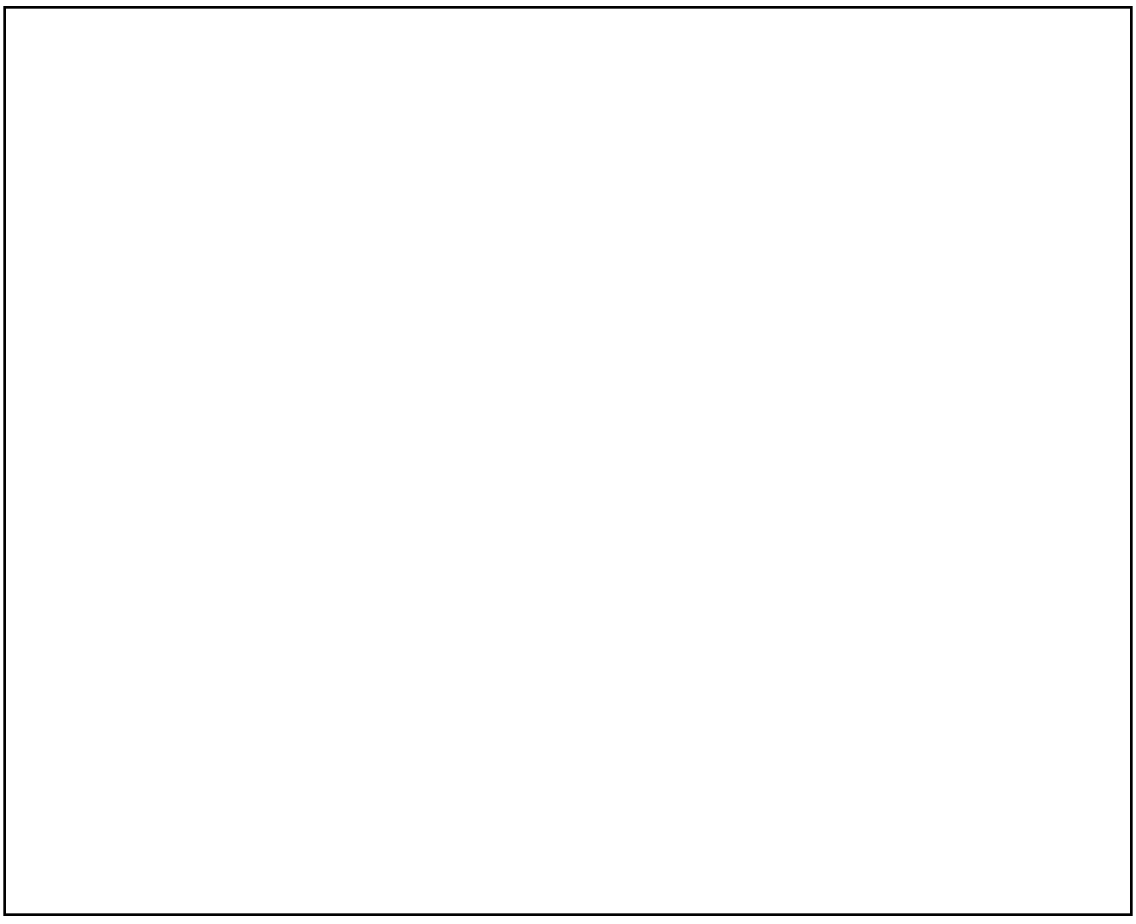


Cryptographic Toolkit

- **Confidentiality:** Encryption
- **Integrity:** Cryptographic Hash
- **Authentication:** Encrypted nonce, Verifier
- **Provenance:** Signatures

Summary of Our Crypto Toolkit

- If we can securely distribute a key, then
  - Symmetric ciphers (e.g., AES) offer fast, presumably strong confidentiality
- Public key cryptography does away with problem of *secure* key distribution
  - But not as computationally efficient
  - Often addressed by using public key crypto to exchange a session key
  - And not guaranteed secure
    - but major result if not



Summary of Our Crypto Toolkit, con't

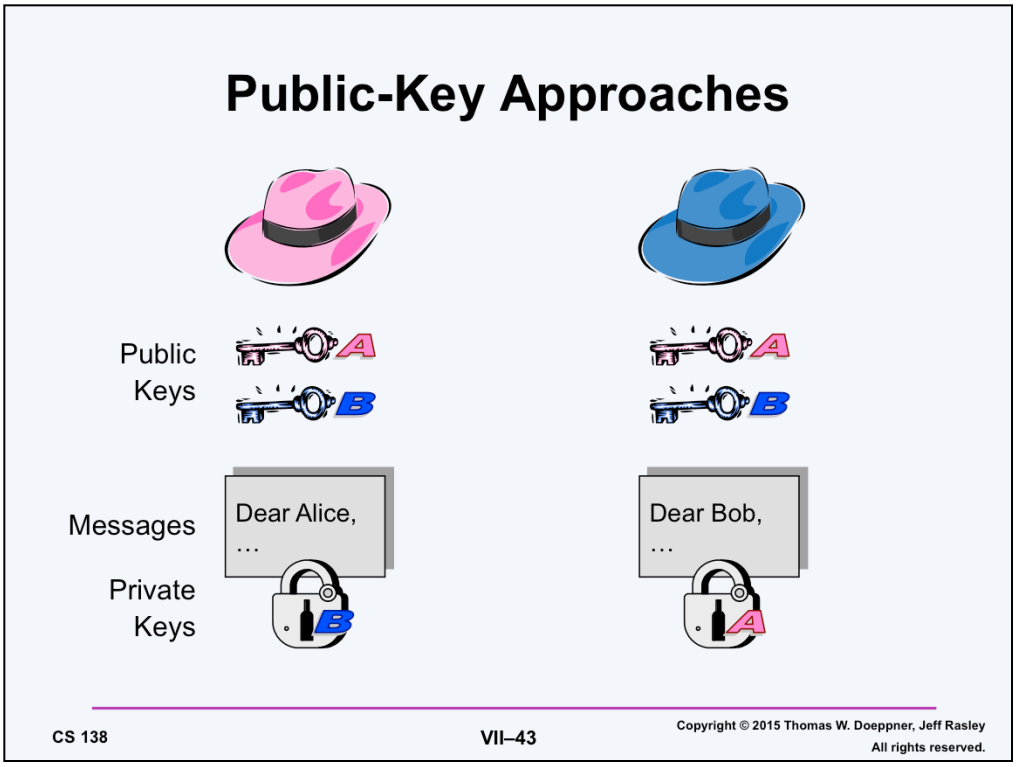
- Cryptographically strong hash functions provide major building block for integrity (e.g., SHA-256)
  - As well as providing concise digests
  - And providing a way to prove you know something (e.g., passwords) without revealing it (non-invertibility)
  - But: worrisome recent results regarding their strength
- Public key also gives us signatures
  - Including sender non-repudiation
- Turns out there's a crypto trick based on similar algorithms that allows two parties *who don't know each other's public key* to securely negotiate a secret key even in the presence of eavesdroppers

We'll talk about symmetric key distribution next class

Issues

- Distribution of public keys
- Cryptographic attacks
- Cost of encryption and decryption

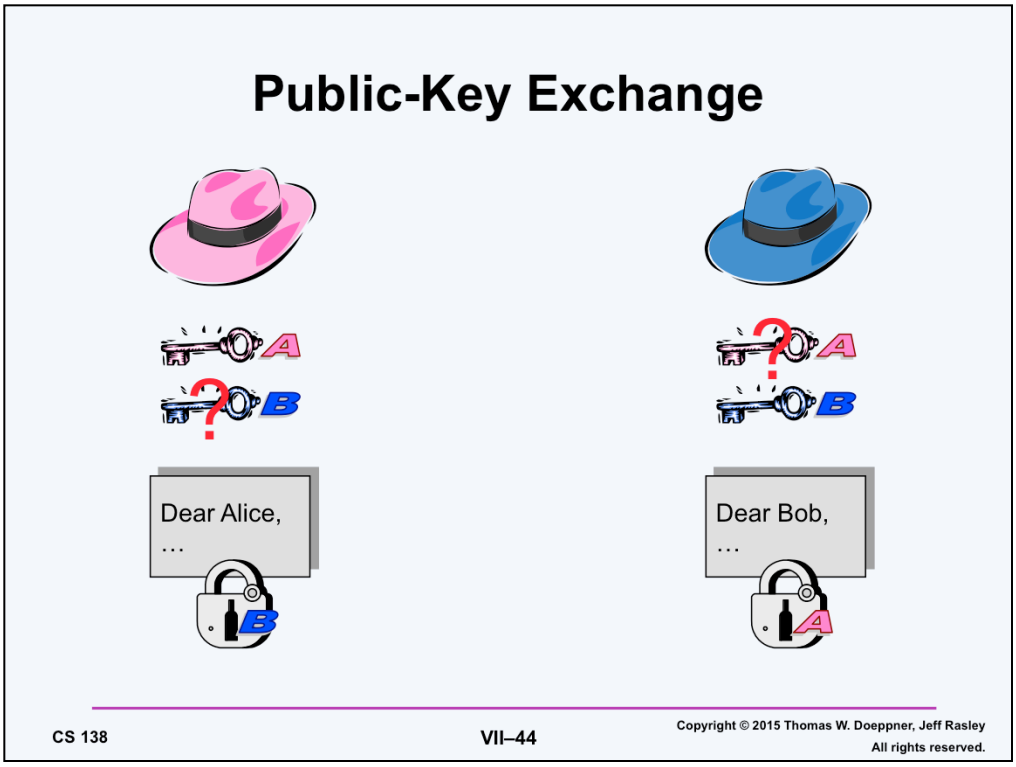
There are a number of tough issues that must be dealt with before public-key technology can be really useful.



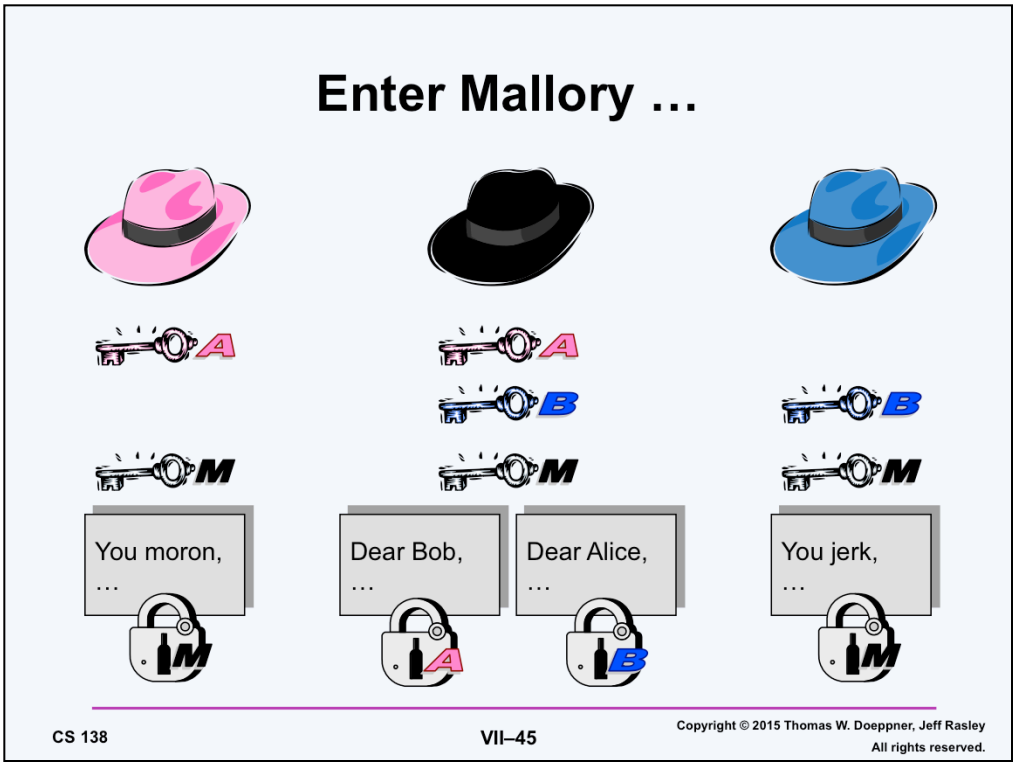
(Obs: we deal with symmetric key approaches, exemplified by Kerberos, in the next lecture)

Rather than use the secret-key approach in conjunction with a key-distribution server, one might use the public-key approach, involving two different keys: one for encrypting and one for decrypting. (In some schemes, either key can be used for either purpose: the keys are effective inverses of each other. In other schemes, one key is used only for encryption; the other used only for decryption.) One of the two keys is made available to everyone, the other is kept private. Of crucial importance is that an immense amount of time is required to compute the value of the private key from that of the public key.

Using this scheme, I might encrypt my message (or verifier) with my private key. Since everyone has a copy of my public key, everyone can decrypt it. However, since encrypting the message required my private key, which is known only to me, the recipients are assured that the message must have come from me.



How do Alice and Bob learn each other's public keys in the first place? A simple approach is for the two of them to simply exchange their public keys.



However, Mallory has somehow gotten between them (perhaps by taking over a router) and has launched a "man-in-the-middle" attack. He intercepts the public keys as they are being exchanged and substitutes his own public key for each. Thus what Alice thinks is Bob's public key is actually Mallory's, and what Bob thinks is Alice's public key is also actually Mallory's. Alice sends a nice message to Bob, encrypted in her private key. It's intercepted by Mallory, who decrypts it using Alice's public key. He can either reencrypt it in his private key, or send something else to Bob (encrypted in Mallory's private key). Bob, on receipt of the message, decrypts it using what he thinks is Alice's public key, but which is really Mallory's. He's convinced that he's received a message from Alice, but, of course, it's really from Mallory. Something similar happens to the message Bob sends to Alice.

Note that this form of attack works even if the public keys are stored in a secure, public database. Mallory, being incredibly resourceful, could always intercept traffic to and from the database.

Digital Signatures: Non-Repudiation

If my seal has not been broken, the information stored herein has not been tampered with and it is irrefutable that I signed this message.

signed

We need an effective technique for thwarting Mallory. One possibility is the digital signature, which is accomplished using a technique similar to how tamper protection is achieved. To sign a message, one first computes a cryptographic checksum (e.g., using MD5), then encrypts the checksum using the signer's private key, thus producing the signature. To verify the signature and to test the integrity of the message, one obtains the signer's public key, decrypts the checksum, and checks to make certain that it is indeed the checksum of the message. One property of digital signatures is that if you sign a message, you cannot later deny the fact that you signed it (for example, by claiming that the signature is a forgery). We'll see soon how this can be used against Mallory, but such signatures are also being touted as a legal replacement for handwritten signatures.

One problem with public keys is reliably determining what someone's public key is. As we've seen, you can't simply ask someone for their public key — your question might be answered by Mallory. Even if we ask a trusted third party for someone's public key, Mallory might still intervene. However, suppose we have a trusted third party that delivers signed messages, known as *certificates*, containing people's public keys. Since Mallory doesn't know the public key of the trusted third party, she can't create or alter such certificates. Thus Alice and Bob can ask the third party (known as the *certificate authority*) for certificates containing each other's public key, and be able to rely on their contents.

Of course, we're in some sense begging the question. How do Alice and Bob reliably obtain the public key of the certifying authority? For this we'll have to assume some sort of trusted channel. For example, the public key comes with your operating system or your browser. (This, of course, is still begging the question. At some point there has to be something that we're willing to trust.)

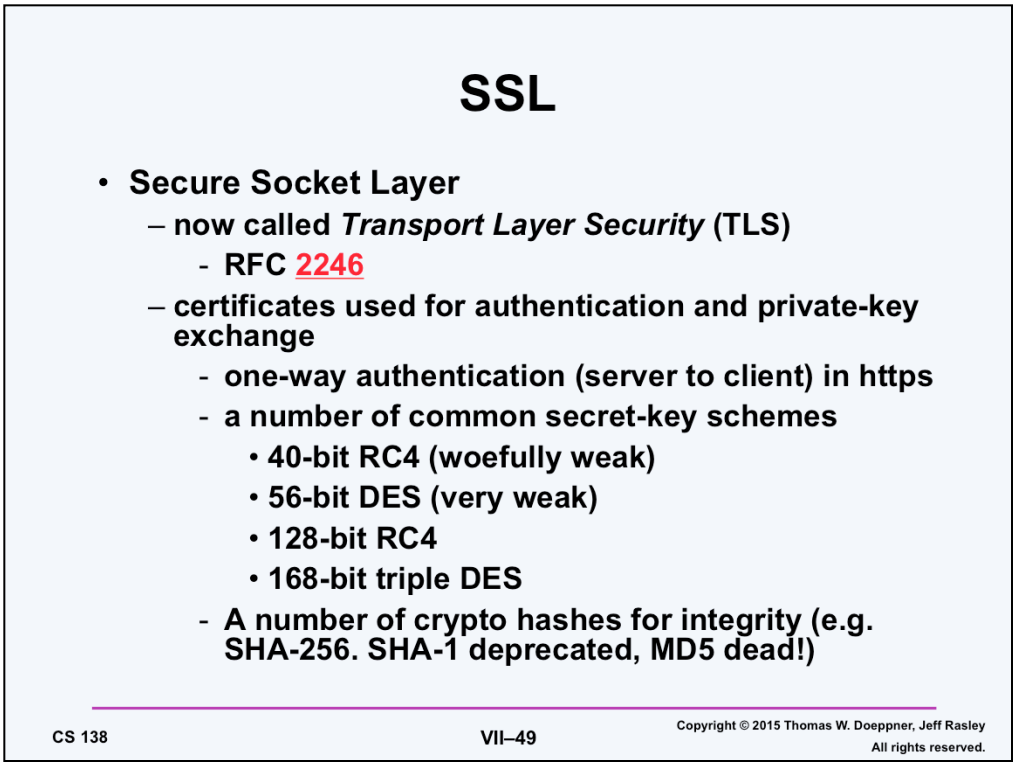
Note that the certificate contains both a validity period and the domain name of the certified party. The latter is a crucial part of the defense against man-in-the-middle attacks. If Mallory (our woman in the middle) tries to use her own certificate rather than Alice's, it would say, perhaps, "mallory.com" rather than "alice.com": one can determine whom a certificate is for.

Another concern is the ability to revoke certificates. For example, Alice is concerned that someone might have stolen her private key. She can inform the certificate authority that she's revoking her old certificate. The authority makes a list of such revoked certificates and asks that people validating certificates first check the list to see if they've been revoked. The validity period puts a limit on how long the revoked certificate must appear in the list.

Hybrid Schemes

- Public-key schemes are close to unbreakable
  - With ~2048-bit keys
- Key distribution simpler than with symmetric (secret-key) schemes
- Generally >1000 times more expensive than symmetric schemes
- Compromise:
  - use public-key scheme to distribute secret keys (known as *session keys*)

Public-key schemes are generally too expensive for general use, despite their other fine properties. What's done in practice is to use public-key schemes to distribute the keys for secret-key schemes.



An example of such a hybrid scheme is SSL. Though originally developed by Netscape, it has become the standard means for security on the web. Sun Microsystems used to have an excellent tutorial on SSL on their web site, but unfortunately both the company and the web site no longer exist.

TLS: Typical Use

- Client connects to server
- Using TLS
  - client authenticates server (via server's certificate)
  - secure channel created
- Over secure channel
  - server authenticates client using user name and password

Client Certificates

- Good idea
  - clients don't give away private information (such as passwords)
- Requires PKI (public-key infrastructure)
  - who vouches for client identites?
- Doesn't scale in practice

Trusting the Authorities

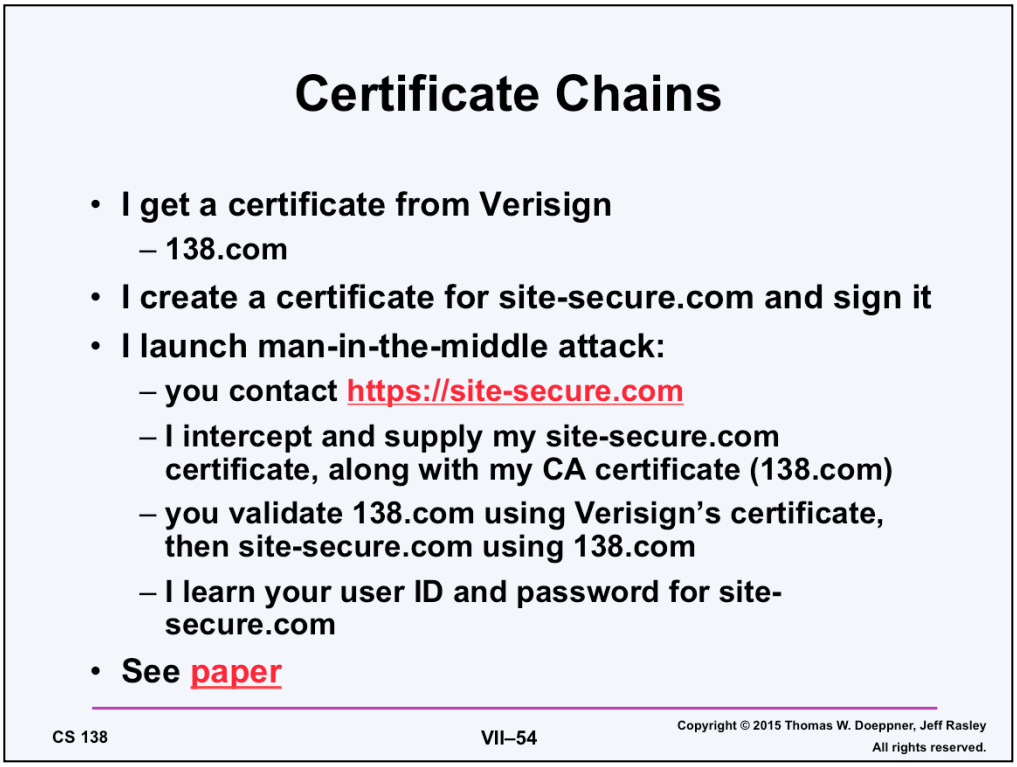
- How do you know the CA's public key?
  - its certificate came with the browser
  - it's certified by a CA whose certificate came with the browser
- "I trust this certificate because I trust *TC TrustCenter* of Hamburg, Germany"
  - huh?
- "I trust *TC TrustCenter* because most browser companies made a deal with it to include its certificate in their browsers"
  - well, maybe …

Exactly whom you should trust and why isn't always exactly clear. (*TC TrustCenter* is a certificate authority whose certificate (and public key) is included with *Internet Explorer* and *Firefox*.)

Is SSL/TLS Secure?

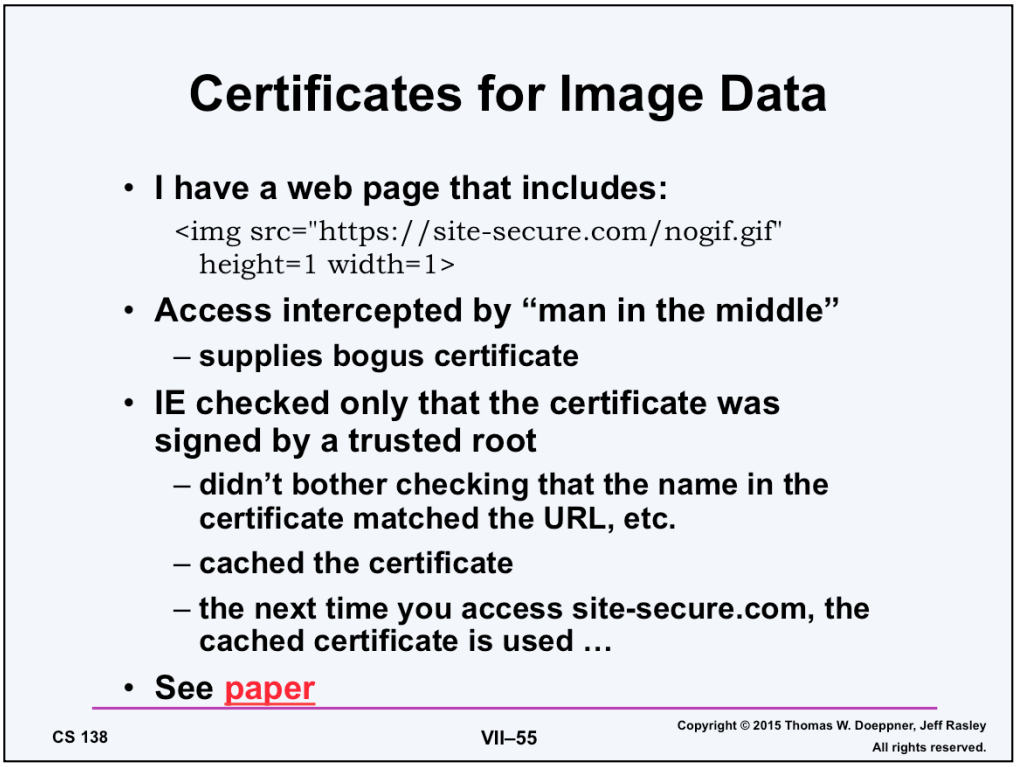
- Probably …
- But, watch out for the implementation!
  - long history of exploitable bugs …
    - heartbeat msgs in TLS allowed dumped random memory (ouch!)
    - poor random-number generator in early Netscape
    - improper checking of certificates in IE
      - "is CA" field in IE 6
      - certificates for image data in IE 6
    - etc.

Heartbleed was reported April 7th 2014 but was vulnerable and undetected for a long time (2+ years?).



This attack was publicized in August 2002 and fixed the following month. The problem was that certificates are supposed to indicate whether they are those of a certificate authority (CA). Only those certificates marked as such can be used to verify another certificate. For non-CA certificates, Verisign omitted the field. **IE, if the field was omitted, assumed the certificate was that of a CA.**

See *http://www.thoughtcrime.org/ie-ssl-chain.txt* for details.



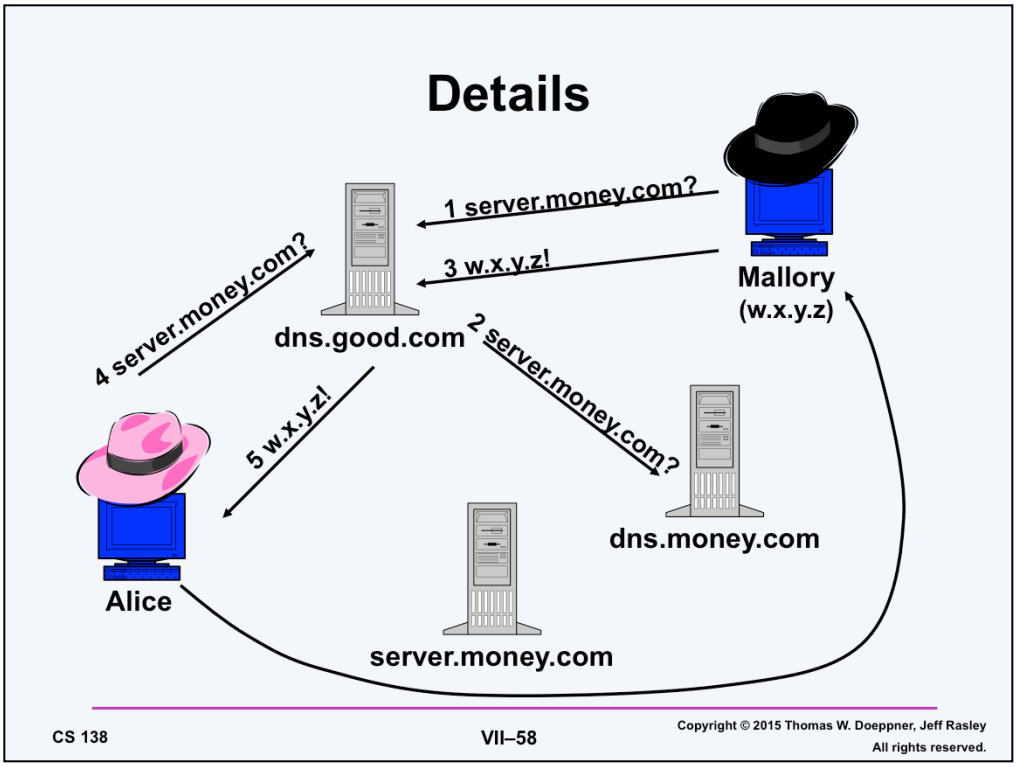
This bug, described in *http://www.sans.org/reading_room/whitepapers/threats/ssl_maninthemiddle_attacks_480/*, announced in December 2001, and fixed shortly thereafter, facilitated man-in-the-middle attacks.

Other PKIs

- DNS-Sec
  - Goal: sign DNS responses
  - Prevent DNS cache poisoning
  - Certificate delegation follows DNS hierarchy
- BGP
  - Goal: sign AS advertisements (I'm AS 2334 and I own prefix 190.221/16) or (I'm AS 445 and you can reach AS 2334 through me)

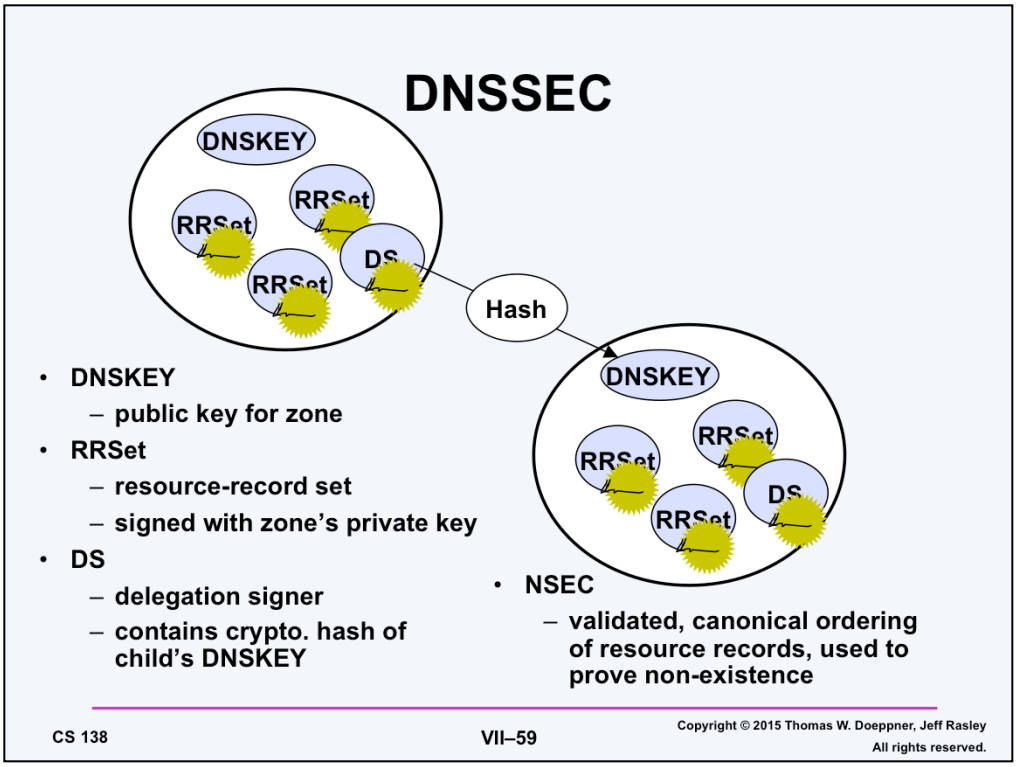
DNS Cache Poisoning

- Idea: put bogus entry in DNS cache
- Trick clients into connecting to wrong machine



Here Mallory is using a DNS-cache-poisoning attack to put an entry in *dns.good.com*'s cache to direct lookups of *server.money.com* to Mallory's machine. Since DNS is layered on top of UDP, the only thing that relates responses to queries is a Query ID number. Mallory first makes a number of queries to *dns.good.com* to determine what these are. She then sends a recursive query to *dns.good.com*, asking for the address of *server.money.com*. *Dns.good.com* recursively sends the result to *dns.money.com*. However, Mallory, spoofing the source address, sends her own response back to *dns.good.com*, stating that *server.money.com* is at *w.x.y.z*, which is the IP address of Mallory's machine. *Dns.good.com* puts this into its cache (for the stated TTL). So, when Alice looks up *server.money.com*, to which she'll be sending her account number, etc., she gets the address of Mallory's machine in response.

This problem goes away, of course, if one uses SSL (why? Hint: we assume that no certificate authority will give Mallory a certificate for *server.money.com*). Netscape had a problem with its implementation of SSL in releases up through 4.72 in which it assumed that all new connections to an already-authenticated IP address required no further authentication. Thus if Alice had an SSL connection with Mallory at w.x.y.z, then tried to contact *server.money.com* (with the cache poisoned as described above), her browser wouldn't try to authenticate this new connection but would assume that the connection to w.x.y.z was properly authenticated for *server.money.com*.



With DNSSEC, each zone has a public/private-key pair, used for signing and validating all information maintained by the zone. Thus, with a validated copy of the public key, one can validate all information obtained from the zone. The public key is provided in the record DNSKEY. Zones have a DS (delegation signer) record for each child zone, containing the hash of the child's DNSKEY. Thus the parent zone validate's the child's public key.

Resolvers (i.e., clients) must have either a DNSKEY or a DS to get started. All validation is relative to such "trust anchors". See http://www.ietf.org/rfc/rfc4033.txt for details.

To verify that something doesn't exist, NSEC records represent the order of records in a zone and thus represent holes.