15-410

"...What about gummy bears?..."

Security Applications
Dec. 2, 2005

Dave Eckhardt

Bruce Maggs

PGP diagram shamelessly stolen from 15-441

Synchronization

P3extra and P4 hand-in directories have been created

- Please check ***IMMEDIATELY*** to make sure yours is there
- Please make sure you can store files there
- Check disk space

Outline

Today

- Warm-up: Password file
- One-time passwords
- Review: private-key, public-key crypto
- Kerberos
- SSL
- PGP
- Biometrics

Disclaimer

- Presentations will be key ideas, not exact protocols

Password File

Goal

- User memorizes a small key
- User presents key, machine verifies it

Wrong approach

- Store keys (passwords) in file
- Why is this bad? What is at risk?

Hashed Password File

Better

- Store hash(key)
- User presents key
- Login computes hash(key), verifies

Password file no longer must be secret

- It doesn't contain keys, only key *hashes*

Vulnerable to *dictionary* attack

- Cracker computes hash("a"), hash("b"), ...
- Once computed, hash ⇒ password list attacks *many users*

Can we make the job harder?

Salted Hashed Password File

Choose random number when user sets password

- Store #, hash(key,#)

User presents key

- Login looks up user – gets #, hash(key,#)
- Login computes hash(typed-key,#), checks

Comparison

- Zero extra work for user, trivial space & work for login
- Cracker must compute a *much larger* dictionary
  - (all “words”) X (all #'s)

Can we do better?

Shadow Salted Hashed Password File

Protect the password file after all

"Defense in depth" - Cracker must

- Either
  - Compute enormous dictionary
  - Break system security to get hashed password file
  - Scan enormous dictionary
- Or
  - Break system security to get hashed password file
  - Run dictionary attack on each user in password file

There are probably easier ways into the system

- ...such as bribing a user!

One-time passwords

What if somebody *does* eavesdrop?

- Can they undetectably impersonate you forever?

Approach

- System (and user!) store key *list*
  - User presents head of list, system verifies
  - User and system *destroy key at head of list*

Alternate approach

- Portable cryptographic clock (“SecureID”)
  - Sealed box which displays E(time, key)
  - Only box, server know key
  - User types in display value as a password

Private-Key Cryptography

Concept: *symmetric* cipher

```
cipher = E(text, Key)
text = E(cipher, Key)
```

Good

- Fast, intuitive (password-like), small keys

Bad

- Must share a key *(privately!)* before talking

Applications

- Bank ATM links, secure telephones

Public-Key Cryptography

Concept: *asymmetric* cipher (aka "magic")

```
cipher = E(text, Key1)
text = D(cipher, Key2)
```

Keys are *different*

- Generate *key pair*
- Publish "public key"
- Keep "private key" *very* secret

Public Key Encryption

Sending secret mail

- Locate receiver's public key
- Encrypt mail with it
- Nobody can read it
  - *Not even you!*

Receiving secret mail

- Decrypt mail with your private key
  - No matter who sent it

Public Key Signatures

Write a document

Encrypt it with your private key

- Nobody else can do that

Transmit plaintext *and ciphertext* of document

Anybody can decrypt with your public key

- If they match, the sender knew your private key
  - ...sender was you, more or less

(really: send E(hash(msg), K_p))

Public Key Cryptography

Good

- No need to privately exchange keys

Bad

- Algorithms are slower than private-key
- Must trust key directory

Applications

- Secret mail, signatures

Comparison

Private-key algorithms

- Fast crypto, small keys
- *Secret-key-distribution problem*

Public-key algorithms

- "Telephone directory" key distribution
- Slow crypto, *keys too large to memorize*

Can we get the best of both?

Kerberos

Goals

- Use fast private-key encryption
- Require users to remember one *small* key
- Authenticate & encrypt for N users, M servers

Problem

- Private-key encryption requires shared key to communicate
- Can't deploy & use system with NxM keys!

Intuition

- *Trusted third party* knows single key of *every* user, server
- Distributes temporary keys to (user,server) on demand

Not Really Kerberos

Client contacts server with a *ticket*

- Specifies *identity* of holder
  - Server will use identity for access control checks
- Specifies *session key* for encryption
  - Server will decrypt messages from client
  - Also provides authentication – only client can encrypt with that key
- Specifies time of issuance
  - Ticket “times out”, client must re-prove it knows its key

Not Really Kerberos

Ticket format

- Ticket={client,time,$K_{session}$}K_s
  - Notation: client, time, session key; DES-encrypted with server's secret key

Observations

- Server knows K_s, can decrypt & understand the ticket
- Clients can't fake tickets, since they don't know K_s
- Session key is provided to server via encrypted channel
  - Eavesdroppers can't learn session key
  - Client-server communication will be secure

How do clients get tickets?

- Only server & Kerberos Distribution Center know K_s...

Not Really Kerberos

Client sends to Key Distribution Center

- "I want a ticket for the printing service"
- {client, server, time}

KDC sends client two things

- $\{K_{session}, server, time\}K_c$
  - Client can decrypt this to learn session key
  - Client knows expiration time contained in ticket
- $Ticket = \{client, time, K_{session}\}K_s$
  - Client cannot decrypt ticket
  - Client *can* transmit ticket to server as opaque data

Not Really Kerberos

Results (client)

- Client has session key for encryption
  - Can trust that only desired server knows it

Results (server)

- Server knows identity of client
- Server knows how long to trust that identity
- Server has session key for encryption
  - Data which decrypt meaningfully must be from that client

Not Really Kerberos

Results (architecture)

- N users, M servers
- System has N+M keys
  - Like a public-key crypto system
  - But fast private-key ciphers are used
- Each entity remembers only one (small) key
  - “Single-sign on”: one password per user

Any weakness?

Securing a Kerberos Realm

KDC (Kerberos Distribution Center)

- Knows *all* keys in system
- Single point of failure
  - If it's down, clients can't get tickets to contact more servers...
- Single point of *compromise*
- *Very* delicate to construct & deploy
  - Turn off most Internet services
  - Maybe boot from read-only media
  - Unwise to back up key database to “shelf full of tapes”

Typical approach

- Multiple instances of server (master/slave)
- Deployed in *locked boxes* in (multiple) machine rooms

SSL

Goals

- **Fast, secure commnication**
- **Any client can contact any server on planet**

Problems

- **There is no single trusted party for the whole planet**
  - **Can't use Kerberos approach**
- **Solution: public-key cryptography?**
  - **Issue: public key algorithms are slow**
  - ***Big problem: there is no global public-key directory***

SSL Approach (Wrong)

Approach

- Use private-key/symmetric encryption for speed
- Swap symmetric session keys via public-key crypto
  - Temporary random session keys similar to Kerberos

Steps

- Client looks up server's public key in global directory
- Client generates random DES session key
- Client encrypts session key using server's RSA public key
- Now client, server both know session key
- Client knows it is talking to the desired server
  - After all, nobody else can do the decrypt...

SSL Approach (Wrong)

Problem

- *There is no global key directory*
- Would be a single point of compromise
  - False server keys enable server spoofing
- If you had a copy of one it would be out of date
  - Some server would be deployed during your download

Approach

- Replace global directory with *chain of trust*
- Servers present their own keys to clients
- Keys are signed by “well-known” certifiers

Not SSL

Server "certificate"

- "To whom it may concern, whoever can *decrypt* messages *encrypted* with public key AAFD01234DE34BEEF997C is www.cmu.edu"

Protocol operation

- Client calls server, requests certificate
- Server sends certificate
- Client generates private-key *session key*
- Client sends $\{K_{session}\}K_{server}$ to server
- If server can decrypt and use $K_{session}$, it must be legit

Any problem...?

SSL Certificates

How did we know to trust that certificate?

Certificates are signed by *certificate authorities*

- "Whoever can *decrypt* messages *encrypted* with public key AAFD01234DE34BEEF997C is www.cmu.edu
  - Signed, Baltimore CyberTrust
    - » SHA-1 hash of statement: 904ffa3bb39348aas
    - » Signature of hash: 433432af33551a343c143143fd11

Certificate verification

- Look up public key of Baltimore CyberTrust in global directory...oops!

SSL Certificates

How did we know to trust the server's certificate?

- Certificates signed by *certificate authorities*
- Browser vendor ships CA public keys in browser
  - Check your browser's security settings, see who you trust!
- "Chain of trust"
  - Mozilla.org certifies Baltimore Cybertrust
  - Baltimore Cybertrust certifies www.cmu.edu

PGP

Goal

- "Pretty Good Privacy" for the masses
- Without depending on a central authority

Approach

- Users generate public-key key pairs
- Public keys stored "on the web" (pgpkeys.mit.edu)
  - Global directory (untrusted, like a whiteboard)
- We have covered how to send & receive secret e-mail

Problem

- How do I *trust* a public key I get from "on the web"?

“On the Web”

PGP key server protocol

- ???: Here is de0u@andrew.cmu.edu's latest public key!
  - Server: “Great, I'll provide it when anybody asks!”
- Rahul: What is de0u@andrew.cmu.edu's public key?
  - Server: Here are 8 possibilities...decide which to trust!

How do I *trust* a public key I get “from the web”?

- “Certificate Authority” approach has issues
  - They typically charge $50-$1000 per certificate *per year*
  - They are businesses...governments can lean on them
    - » ...to present false keys...
    - » ...to delete your key from their directory...
    - » ...to refuse to sign your key...

PGP

"*Web* of trust"

- Dave and Bruce swap public keys ("key-signing party")
- Bruce signs Dave's public key
  - Publishes signature on one or more web servers
- Rahul and Bruce swap public keys (at lunch)

Using the web of trust

- Rahul fetches Dave's public key
  - Verifies Bruce's signature on it
- Rahul can safely send secret mail to Dave
- Rahul can verify digital signatures from Dave

PGP “key rings”

Private key ring

- All of your private keys
- Each encrypted with a “pass phrase”
  - Should be longer, more random than a password
  - If your private keys leak out, you can't easily change them

Public key ring

- Public keys of various people
  - Each has one or more signatures
  - Some are signed by you – your PGP will use without complaint

PGP Messages

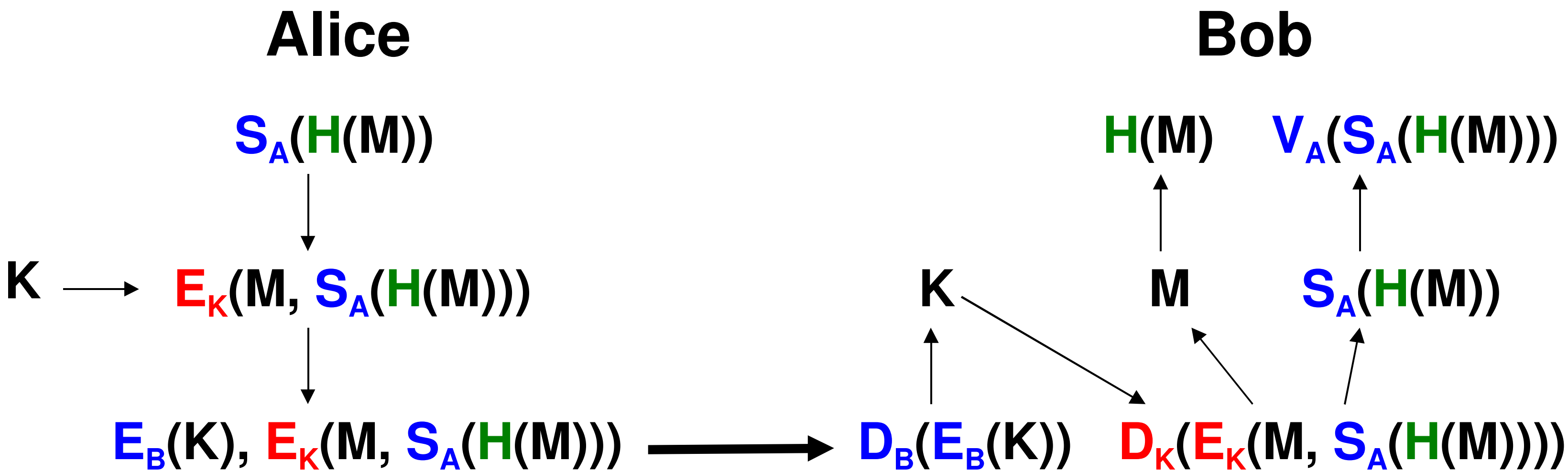
Message goals

- Decryptable by multiple people (recipients of an e-mail)
- Large message bodies decryptable quickly
- Message size not proportional to number of receivers

Message structure

- One message body, encrypted with a symmetric cipher
  - Using a random “session” key
- N key packets
  - Session key public-key encrypted with one recipient's key

Not PGP



Note: on this slide, E_K(a, b) means ..."a and b"...

Biometrics

Concept

- Tie authorization to ***who you are***
  - Not what you know – can be copied
- Hard to impersonate a retina
  - Or a fingerprint

Biometrics

Concept

- Tie authorization to ***who you are***
  - Not what you know – can be copied
- Hard to impersonate a retina
  - Or a fingerprint

Right?

Biometrics

Concept

- Tie authorization to *who you are*
  - Not what you know – can be copied
- Hard to impersonate a retina
  - Or a fingerprint

Right?

What about gummy bears?

Summary

Many threats

Many techniques

“The devil is in the details”

Just because it “works” doesn't mean it's right!

Open algorithms, open source

Further Reading

Kerberos: An Authentication Service for Computer Networks

- B. Clifford Neuman, Theodore Ts'o
- USC/ISI Technical Report ISI/RS-94-399

Impact of Artificial "Gummy" Fingers on Fingerprint Systems

- Matsumoto et al
- http://cryptome.org/gummy.htm