15-410
"...What about gummy bears?..."

Security Applications
Apr. 8, 2011

Dave Eckhardt

PGP diagram shamelessly stolen from 15-441

SecurID picture clipped from rsa.com

Synchronization

Saturday night

- Computer Club/KGB “retro computer game” night
- 21:00
- University Center (Connan Room, black-chair lounge)
- Featured attractions
  - Amiga A1200, Sinclair ZX Spectrum, Atari 800XL, Commodore 64, Commodore 128, Mac Quadra, MS-DOS PC, NES, SNES

Disclaimer

- “Business attire” not recommended

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
  - "Protocols discussed in lecture are larger than they appear"

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?

```
alice : Whimsy33Fish/
bob   : secret
chas  : secret
```

Hashed Password File

Better

- Store `hash(key)`
  - `hash("Whimsy33Fish/") ⇒ X93f3ZaWhT`
  - `hash("secret") ⇒ fg8ReCFySk`
- User presents key
- Login program computes `hash(key)`, compares to file
  - Note: we use a collision-resistant (cryptographic) hash

```
alice : X93f3ZaWhT
bob   : fg8ReCFySk
chas  : fg8ReCFySk
```

Hashed Password File

Original Unix password file was made public

- Didn't contain keys, only key *hashes*

Still vulnerable to *dictionary attack*

- Cracker computes `hash("a")`, `hash("b")`, stores reverse
  - `unhash("54GtYuREbk")` ⇒ `"a"`
  - `unhash("PoLka67vab")` ⇒ `"b"`
- Once computed, hash ⇒ password list attacks *many users*
  - `unhash("fg8ReCFySk")` ⇒ `"secret"` hits Bob *and* Chas
  - Note: cracker may quit before `hash("Whimsy33Fish/")`

Hashed file is "arguably less wrong"

- Can we make the cracker's job even harder?

Salted Hashed Password File

Choose random number when user sets password

- Store `#`, `hash(#,key)`
  - `hash("Xz Whimsy33Fish/")` ⇒ `uiR34ExWmT`
  - `hash("p0 secret")` ⇒ `998ueTRvMx`
  - `hash("9Q secret")` ⇒ `opTkr7Sfh3`

User presents key

- Login looks up user, retrieves `#` and `hash(#,key)`
- Login computes `hash(#,typed-key)`, compares to file

```
alice : Xz : uiR34ExWmT
bob   : p0 : 998ueTRvMx
chas  : 9Q : opTkr7Sfh3
```

Salted Hashed Password File

Evaluation of “salt” extension

- *Zero* extra work for user
  - User still remembers just the password
  - Salt is invisible
- Trivial extra space & work for login program
  - Store a few more bytes
  - Hash a slightly-longer string
- Pre-computed dictionary must be *much larger*
  - Without salt: cracker must hash all “words”
  - With salt: cracker must hash (all “words”) X (all #'s)
    - » 2 random salt bytes [A-Za-z0-9] increases work 3844-fold
    - » Linear work for target, exponential work for cracker!

Can we do even better?

Shadow Salted Hashed Password File

Protect the password file after all

“Defense in depth” - Cracker must

- Either
  - Compute enormous all-word/all-salt dictionary
  - Break system security to get hashed password file
  - Scan through enormous all-word/all-salt dictionary
- Or
  - Break system security to get hashed password file
  - Run all-word 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 a password?

- **Can they undetectably impersonate you forever?**

“One-time passwords”

- **System (and user!) store key *list***
  - **User presents head of list, system verifies**
  - **User and system both *destroy key at head of list***
  - **Eavesdropper learns nothing with a future use**

One-time passwords

What if somebody *does* eavesdrop a password?

- Can they undetectably impersonate you forever?

"One-time passwords"

- System (and user!) store key *list*
  - User presents head of list, system verifies
  - User and system both *destroy key at head of list*
  - Eavesdropper learns nothing with a future use

Alternate approach

- Portable cryptographic clock
  - Sealed box which displays `E(time,key)`
  - Only box & server know the key
  - User types in displayed value as a password

Cryptography on One Slide

Symmetric / private-key cipher

```
ciphertext = E(cleartext, Key)
cleartext = E(ciphertext, Key)
Examples: DES, RC4, IDEA, Threefish, AES
```

Asymmetric / public-key cipher (aka “magic”)

```
ciphertext = E(cleartext, Key1)
cleartext = D(ciphertext, Key2)
Examples: RSA, ElGamal, Elliptic curve
```

Reminder: 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

Actually

- send E(hash(msg), $K_{private}$)

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

Authenticating to a "server"

- Client = de0u, server = "afs@ANDREW.CMU.EDU"

Client contacts server with a *ticket*

- Contains *identity* of holder
  - Server will use identity for access control checks
- Contains *ephemeral session key* for encryption
  - Roll dice to generate a key for today, then throw it away
  - Server will decrypt messages from client using this key
  - Also provides authentication – only client can encrypt with that key
- Contains time of issuance
  - Ticket "times out"
  - Client must get another one – re-prove it knows its key

Not Really Kerberos

Ticket format

- Ticket={client,time,$K_{session}$}K_s
  - {client, time, session key} DES-encrypted with server's 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 using K_s will be secure

How do clients get tickets?

- ?

Not Really Kerberos

Ticket format

- Ticket={client,time,$K_{session}$}K_s
  - {client, time, session key} DES-encrypted with server's 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 using K_s 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 when the ticket will expire
- $Ticket = \{client, time, K_{session}\}K_s$
  - 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?

- What could make the system fail?

Securing a Kerberos Realm

KDC (Kerberos Distribution Center)

- **Single point of failure**
  - **If it's down, clients can't get tickets to contact more servers...**
    - » **Ok, fine, multiple instances of server (master/slave)**

Securing a Kerberos Realm

KDC (Kerberos Distribution Center)

- **Single point of failure**
  - **If it's down, clients can't get tickets to contact more servers...**
    - » **Ok, fine, multiple instances of server (master/slave)**
- **Each server knows *all* keys in system**
  - **Single point of *compromise***
    - » **Deployed in *locked boxes* in (multiple) machine rooms**
- ***Very* delicate to construct & deploy**
  - **Turn off most Internet services**
  - **Maybe boot from read-only media**
  - **Maybe booting requires entry of master password**
  - **Unwise to back up key database to "shelf full of tapes"**

SSL

Goals

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

Problems

- There is no single trusted key server for the whole planet
  - Can't use Kerberos approach
- Solution: public-key cryptography?

SSL

Goals

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

Problems

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

SSL Approach ("Not exactly")

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 (“Not exactly”)

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 directly 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

- **Compute SHA-1 hash of server's key statement**
- **Look up public key of Baltimore CyberTrust in global directory...oops!**

SSL Certificates

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

- **Certificates are 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, ex., www.cmu.edu**

SSL Certificates

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

- **Certificates are 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, ex., www.cmu.edu**
  - **Say, who actually certifies www.cmu.edu?**

SSL Certificates

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

- Certificates are 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, ex., www.cmu.edu
  - Say, who actually certifies www.cmu.edu?
    - » As of 2011-04-08: "UTN-USERFirst-Hardware"
    - » You've heard of them, right? Household name?

SSL Certificates

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

- Certificates are 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, ex., www.cmu.edu
  - Say, who actually certifies www.cmu.edu?
    - » As of 2011-04-08: “UTN-USERFirst-Hardware”
    - » You've heard of them, right? Household name?
    - » How about “NetLock Halozatbiztonsagi Kft.”???

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/sign 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!"
- Alice: What is de0u@andrew.cmu.edu's public key?
  - Server: Here are 8 possibilities...you 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 Joshua swap public keys (“key-signing party”)
- Joshua signs Dave's public key
  - “937022D7 is the fingerprint of de0u@andrew.cmu.edu's key” -- sincerely, 77432900
  - Publishes signature on one or more web servers
- Joshua and Sean swap public keys (at lunch)

Using the web of trust

- Sean fetches Dave's public key from the web
  - Verifies Joshua's signature on it
- Sean can safely send secret mail to Dave
- Sean 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

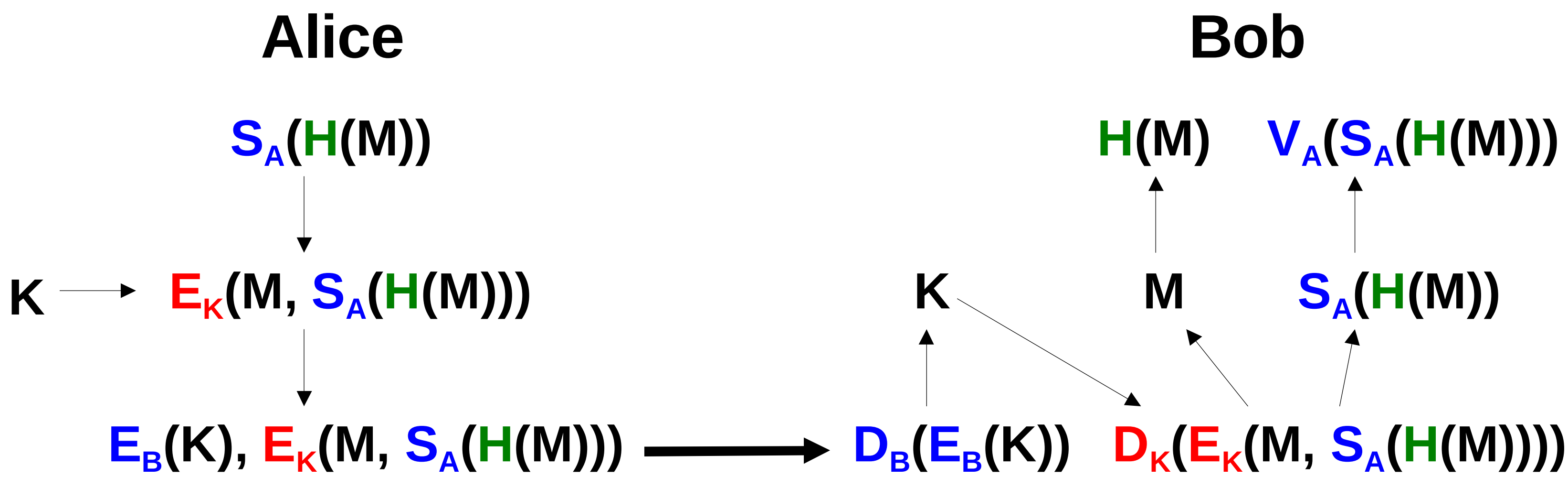
Message goals

- 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"...with K
(Notation closer to textbook's than to mine)

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?***

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?
- *What about carjackers?*

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

PGP Pathfinder

- http://pgp.cs.uu.nl/paths/3970227D/to/5B0358A2.html

Kerberos: An Authentication Service for Computer Networks

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

Further Reading

"Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL"

- http://files.cloudprivacy.net/ssl-mitm.pdf

"Creating a rogue CA certificate"

- http://www.phreedom.org/research/rogue-ca/

Impact of Artificial "Gummy" Fingers on Fingerprint Systems

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

Amputation hazards of biometrics

- http://www.theregister.co.uk/2005/04/04/fingerprint_merc_chop/