Well, who are you? (Who are you? Who, who, who, who?)
I really wanna know (Who are you? Who, who, who, who?)
Tell me, who are you? (Who are you? Who, who, who, who?)
'Cause I really wanna know (Who are you? Who, who, who, who?)

**What is hashing?**

- Hashing is the generation of a fixed-size number from a variable-sized block of input
  - Many-to-one mapping
  - Even distribution: for random input, each output value is (approximately) equally likely
  - Efficient: hash function is easy to compute
- Cryptographic hash functions
  - One-way: given the hash function $H$, and a hash value $h$, it’s hard to find $x$ such that $H(x) = h$
  - Collision resistance
    - *Weak* collision resistance: given $x$, it’s hard to find $y \neq x$ such that $H(y) = H(x)$
    - *Strong* collision resistance: it’s hard to find *any* $x$ and $y \neq x$ such that $H(y) = H(x)$
Using cryptographic hashes

- Alice wants to send Bob an “I owe you” message
- Bob wants to be able to show the message to a judge to compel Alice to pay up
- Alice wants to prevent Bob from changing the contents of the message or making up his own IOU from Alice

IOU protocol, first try

Judge sez:
Hmmm — Bob could have just made up $M$ and $H(M)$!
IOU protocol, second try

Alice

Secret key $K_A$

Use Diffie-Hellman to establish shared secret $K_A$

Judge

Knows $K_A$

Can Bob cheat?
Can Alice cheat?
YES!
She can send Bob $M || junk$
The judge will think Bob cheated!

IOU protocol, third try

Alice

$\{K_{UA}, K_{RA}\}$

Public (KU) & private (KR) pair

Judge

Knows $K_{UA}$

Bob

Knows $K_{UA}$

Bob can verify $H(M)$ by decrypting, but can’t forge $M, E_{K_{RA}}[H(M)]$ pair without knowing $K_{RA}$
Weak collision resistance

- Suppose we use: \( H(\text{char } s[]) = (s[0] – ‘a’) \mod 10 \)
  - Alice sends Bob:
    “I, Alice, owe Bob $50.”, \( E_{KR_{A}}[H(M)] \)
  - Bob sends Judge:
    “I, Alice, owe Bob $500000.”, \( E_{KR_{A}}[H(M)] \)
  - Judge validates \( E_{KR_{A}}[H(M)] = H(“I, Alice, owe Bob $500000.”) \) and makes Alice pay
- Given \( x \), it should be hard to find \( y \neq x \) such that \( H(y) = H(x) \)
- Similar to a block cipher, but without the need for a secret key
  - Changing any bit of \( x \) should change most of \( H(x) \)
  - The mapping between \( x \) and \( H(x) \) should be confusing (complex and non-linear)

Is there a better hash function?

- Use \( H(x) = AES (x, 0) \)
- Weak collision resistance?
  - Given \( x \), it should be hard to find \( y \neq x \) such that \( H(y) = H(x) \)
  - This is the case for AES because AES is one-to-one
    - There is (by definition) no such \( y \)
- Is this a good hash function?
  - No!
    - Its output is as big as the message
    - Doesn’t summarize the message
- We need a hash function that
  - Produces a small number of bits (64–256)
  - Depends on the message in a confusing, non-linear way
Hash using cipher block chaining

- Cipher block chaining
  - Each ciphertext block depends on the previous one
  - Still too long!
- Use the last block from CBC mode
  - Still depends on all previous plaintext data
  - Fixed length
  - Problem: not good if you use both hashing and encryption: the hash value is the same as the last ciphertext value!

Actual hashing algorithms

- Based on cipher block chaining
- No need for
  - Secret key
  - Initialization vector
  - Use 0 for both
- Don’t use AES or other encryption algorithms
  - Somewhat slower
  - May “conflict” with encryption uses
- Two modern examples are
  - MD5: 512 bit blocks, produces 128-bit hash
  - SHA: 512 bit blocks, produces 160-bit hash
Why use big hashes?

- Encryption is probably OK with 64 bit blocks
- Why do secure hash functions need at least 128 bit digests?
- Answer: strong collision resistance
  - 64 bits is OK for weak collision resistance
  - It’s too easy to find matches where we can build both messages we want to match

Strong collision resistance

- It is hard to find any \( x \) and \( y \neq x \) such that \( H(y) = H(x) \)
- Difference from weak collision resistance
  - Attacker gets to choose both \( x \) and \( y \), not just \( y \)
- Scenario
  - Bob writes IOU message and sends it to Alice for signing
  - Bob can use exploit weak collision to cheat Alice
IOU request protocol

Alice

Bob

{KU_A, KR_A}

M_1

E_{KRA}[H(M_1)]

M_2

E_{KRA}[H(M_1)]

Judge

knows KU_A

Bob picks M_1 and M_2 such that
H(M_1) = H(M_2)

Finding M1 and M2

- Bob generates $2^{10}$ different agreeable (to Alice) messages:
  - I, { Alice | Alice Hacker | Alice P. Hacker | Ms. A. Hacker },
  - { will pay | agree to pay } Bob { the sum of | the amount of }
    { $50 | $50.00 | 50 dollars | fifty dollars } { prior to | before }
    { July 1st | 1 July | 7/1 | 07/01 } { 2005 | 2005 AD}.

- Bob generates $2^{10}$ different agreeable (to Bob) messages:
  - I, { Alice | Alice Hacker | Alice P. Hacker | Ms. A. Hacker },
  - { will pay | agree to pay } Bob { the sum of | the amount of }
    { $5,000,000 | $5,000,000.00 | $5 million | five million dollars } { prior to | before }
    { July 1st | 1 July | 7/1 | 07/01 } { 2005 | 2005 AD}. 
Who wants to be a millionaire? Bob?

- For each message $M_1^i$ and $M_2^j$, Bob computes $hM_1^i = H(M_1^i)$ and $hM_2^j = H(M_2^j)$
- If $hM_1^i = hM_2^j$ for some $i$ and $j$
  - Bob sends Alice $M_1^i$
  - Alice produces $E_{KR_A}[h(M)]$ and returns it to Bob
  - Bob sends the judge $M_2^j || E_{KR_A}[h(M)]$
- The judge checks the signed hash value, and awards Bob $5$ million!
  - How could we avoid this?

Chances of success

- Hash function generates 64-bit digest ($n = 2^{64}$)
- Hash function is good (randomly distributed and diffuse)
- Chance a randomly chosen message maps to a given hash value: $1$ in $n = 2^{-64}$
- By hashing $m$ good messages, chance that a randomly chosen message maps to one of the $m$ different hash values: $m \times 2^{-64}$
- By hashing $m$ good messages and $m$ bad messages: $m \times m \times 2^{-64}$
Does Bob get wealthy?

- \( m = 2^{10} \)
- \( 2^{10} \times 2^{10} \times 2^{-64} = 2^{-44} \) (doesn’t look good...)
- Try \( m = 2^{32} \)
- \( 2^{32} \times 2^{32} \times 2^{-64} = 2^0 = 1 \) (yippee!)
- Flaw: some of the messages might hash to the same value, might need more than \( 2^{32} \) to find match
  - However, \( 2^{32} \) isn’t all that many messages
  - Just need a message with 32 pairs of words that are equivalent
  - This isn’t so hard to do...

How big should a hash value be?

- \( P(n, k) = 1 - n!/(n-k)!n^k \)
- Given \( k \) random selections from \( n \) possible values, \( P(n, k) \) gives the probability of at least one duplicate
- We can derive an easier to calculate formula:
  \[ P(n, k) > 1 - e^{-k(k-1)/2n} \]
  - This was derived using \((1-x) ≤ e^{-x}\)
- Example: \( 2^{70} \) objects, 160 bit hash (like SHA)
  - For large \( k, k - 1 = k \)
  - For example, \( k/2n = 2^{-20} \)
  - \( P(2^{60}, 2^{70}) > 9.5 \times 10^{-7} \Rightarrow \) chance of at least 1 duplicate
  - \( 2^{70} \) objects is a trillion objects per second for 30 years!
- With big enough hash values and public key encryption, we get **digital signatures**!
Certificates

- Use hashing and public keys to build a tree of trust
  - Must trust at least one public key (certificate authority)
  - CA signs keys, allowing them to be trusted
  - If you trust a signed key, it too can be used to sign keys

How are certificates checked?

- TrustMe.com’s certificate is well-known
  - Incorporated into the browser!
- Decrypt the certificate using $K_U^T$ and check the hash value
What if a key is compromised?

• Since certificates are self-contained, it’s hard to revoke them
• Solution: certificates have times attached
  ◦ Times included in signed hash
  ◦ After time has expired, certificate is no good
  ◦ User has to get a new certificate
  ◦ If the key is compromised, no certificate will be reissued
• Advantages
  ◦ Certificates not good forever
  ◦ Certificate authorities can make more money!
• Disadvantages
  ◦ It takes time for a certificate to be disavowed
  ◦ Workaround: present certificate to CA for verification
    – Slow, so not done normally

Message Authentication Codes

• Problem: RSA signatures are too slow
  ◦ Not a problem for certificates: modified infrequently
  ◦ May be a problem for streams of data
• Solution: use Message Authentication Codes (MACs)
  ◦ Create a hash that includes all of the data and a secret shared between sender and receiver
  ◦ Intruder can see the data and the MAC, but has no idea of how to generate one
  ◦ Hashing is fast, so this method works quickly
  ◦ Drawback: relies upon shared secret (perhaps exchanged at the start of communication using RSA or DH)
• Data can be encrypted as well…
Sample MAC

- Prepend the key to the plaintext and hash the whole thing
- Hash the result along with the key
- Transmit the result along with the plaintext
- Can’t just send the result of step 1 (why not?)
- Impossible to regenerate the MAC without the secret key
  - Easy to tell if the plaintext (or hash) has been modified at the other end
  - Can’t tell who created the hash (must know key, though)

Passwords

- Authenticate a person to a computer
- Early (obvious scheme)
  - User sends password (in the clear) via direct link to system
  - System compares user ID & password against table
- Works fine with directly connected users?
  - Eve can’t see plaintext passwords in transit
Authentication problems

- Need to store the passwords somewhere
  - Must rely on the file remaining secure
  - Legitimate users could perhaps read the file, allowing them to impersonate someone else
  - Could be encrypted, but then where would the key be kept?
- Need to transmit passwords from user to host
  - Use a secure line: no remote logins (or difficult at best)
  - Encrypt the transmission?

Encrypted passwords

Terminal

Login: gwbush
Password: rangersfan

Login sends
<"gwbush", DES(0, “rangersfan”)>

Problem: anyone could simply look up DES(0, “rangersfan”) in the password file and send that value…

Trusted subsystem compares to stored value
Encrypted passwords redux

Terminal

Login: gw bush
Password: rangersfan

Login sends
<“gw bush”, “rangersfan”>

Trusted subsystem

Trusted subsystem computes
DES(0, “rangersfan”) and
compares to stored value

Making brute force attacks harder

- Use a slower encryption algorithm
  - DES is pretty slow
  - Limits the speed with which attackers can compute strings from passwords
- Even slower: run DES many times
  - Unix uses DES$^{25}$ (0, password)
  - Not more secure, but a lot slower…
- Require longer passwords
  - DES key is only 56 bits long, so it only uses the first 7.5 ASCII characters of the password
- Brute force is unlikely to work with DES$^{25}$ and all possible 8-letter passwords
  - Recent developments may have made this possible, though…
- Are all 8-letter passwords likely to occur?
Dictionary attacks

- Try a list of common passwords
  - All 1-4 letter words
  - List of common names (human, dog, cat)
  - Words from dictionary
  - Pairs of words
  - Phone numbers, license plates
  - Substitute numbers or symbols for letters
  - All of the above in reverse
- Simple dictionary attacks retrieve most user-selected passwords
  - 86% of users are dumb!

<table>
<thead>
<tr>
<th>Password Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single ASCII character</td>
<td>0.5%</td>
</tr>
<tr>
<td>Two characters</td>
<td>2%</td>
</tr>
<tr>
<td>Three characters</td>
<td>14%</td>
</tr>
<tr>
<td>Four alphabetic letters</td>
<td>14%</td>
</tr>
<tr>
<td>Five same-case letters</td>
<td>21%</td>
</tr>
<tr>
<td>Six lowercase letters</td>
<td>18%</td>
</tr>
<tr>
<td>Names or words in dictionaries</td>
<td>15%</td>
</tr>
<tr>
<td>Other (possibly good) passwords</td>
<td>14%</td>
</tr>
</tbody>
</table>

[Morris/Thompson 79]

Making dictionary attacks harder

- Slower encryption (already covered)
- Force (or convince) users to choose better passwords
  - Test selected passwords against a known dictionary
  - Require numeric or non-alphanumeric characters
  - Require at least $n$ character passwords
  - Better software that allows longer passwords
  - Salt…
- Problem: users don’t like rules
  - They get annoyed
  - They may write down passwords if they’re too hard to remember
Multiple users with the same password

- What if two people have the same password?
  - DES\(^{25}\) will result in the same value!
  - Must guard against this!
- Solution: add “salt”
  - Extra random bits (12 in the case of Unix) added to password before encryption
  - Same password results in different outcome because of different salt value
- Unix uses this scheme, but with DES+ (salt-dependent E-tables)

Salt and brute-force password cracking

- Salt makes it harder to use brute-force to crack passwords
  - More possible hash values for each password
  - Makes it harder to use static tables
- Unix uses 12-bit salt, which only makes it 4096x harder
- Consider an alternate system that uses
  - MD5
  - 30-bit (5 character) salt
  - This requires building \(2^{30}\) tables
  - Alternately, could just try to brute-force a single password for which the salt is known (it’s in the password file entry)
  - Either way, salt makes the password file harder to crack
- None of this helps unless users pick good passwords!
Problem: what about Eve?

- Eve can spy on transmissions between user and system
- Eve can capture user name and password and replay them
  - No need to even see password file
- How can we guard against this?
  - Nonces
  - Prevent replay attacks
  - Ask Martin Abadi…

Authentication isn’t that simple!

- Computers can only verify a “token”
  - Password
  - Cardkey
  - Fingerprint
  - Other stuff
- Computers can’t ensure that the bearer of the token is authorized to use it!
  - Stolen token
  - Forged token
- Assuming that token holder is authorized may lead to problems….
Ways to get those tokens...

- Tokens can be obtained in many ways
- Technical
  - Crack the encryption: this can be difficult
  - Break the protocol: this might not be so hard
- Semi-technical
  - Guess the password
  - Use technical solutions to help narrow down choices
- Non-technical
  - Spying (could use technology, though)
  - Stealing
  - Other social engineering techniques
    - Convincing user to give up his password
    - Blackmail
    - Torture
    - Use your imagination!

Authentication protocols

- Goal of authentication: convince the other party that you are who you say you are
- We now have a set of tools for managing passwords and authenticating locally
  - Works over trusted links
  - Real world is full of insecure links!
- Lots of potential problems with insecure links
  - Eavesdropper can listen to transmissions
  - Transmissions can be replayed
  - Authentication sessions can be faked
**Authentication protocols**

- Two principals, which may be
  - Computers (clients / servers)
  - Users
- Secrets, possibly shared
  - Often encryption keys
- Encryption
  - Symmetric key
  - Public / private key
- Trusted servers
- Proofs of timeliness
  - Timestamps
  - Nonces
- End result: secure channel established

**Simple protocol**

- Each principal has a public / private key pair
- S is a *certification authority* for public keys
- Alice and Bob agree on a session key $K_{AB}$ and send data using it

1. A,B
2. Public keys for A,B
3. $E_{K_{UA}}(K_{AB})$
4. $E_{K_{AB}}(M)$
Pitfalls in authentication protocols

- Replay attacks
  - Simple replay (replay the message)
  - Repetition within a time window
  - Undetectable replay (original message suppressed)
  - Backward replay (message replayed to sender)

- Avoid replay with
  - Timestamps
  - Challenge / response, often using a nonce

- Include sufficient information to guarantee protocol can’t be faked
  - Never make any assumptions about messages!
  - Information in messages must be tamper-proofed

Denning-Sacco protocol

1. A, B

2. $E_{KRS}(A, KU_A, T)\quad E_{KRS}(B, KU_B, T)$

3. $E_{KRS}(A, KU_A, T)\quad E_{KRS}(B, KU_B, T)\quad E_{KUB}(E_{KRA}(K_{AB}, T))$

4. $E_{KAB}(data, T)$

- Encryption with a private key is typically done to sign a message.
- Encryption with a public key is done for secrecy
- $T$ is a timestamp
Is one protocol enough?

- One protocol isn’t enough
- We may want
  - Different crypto systems
  - Two-way authentication
  - Fewer messages
  - Central authority (or lack of it)
  - Anonymity
- Typically, one protocol can’t meet all of these criteria
  - Example: difficult to use only symmetric-key encryption without a central authority

Needham-Schroeder shared-key protocol

1. A, B, N_A
2. E_{K_{AS}}(N_A, B, K_{AB}, E_{K_{BS}}(K_{AB}, A))
3. E_{K_{BS}}(K_{AB}, A)
4. E_{K_{AB}}(N_B)
5. E_{K_{AB}}(N_B^{-1})

N_A, N_B are nonces
K_{XS} is a symmetric key known only to X and S
Problems with this protocol?

- What if attacker (much) later discovers $K_{AB}$?
  - Possible to impersonate A!
  - No guarantee that $E_{E_{KS}}(K_{AB}, A)$ is fresh
  - Replay this value to B, and pick up protocol from there

- Potential solutions
  - Make sure $K_{AB}$ is never discovered (strong key)
  - Allow B and S to interact
  - Use timestamps

Descendants of Needham-Schroeder

- Kerberos protocol derived from Needham-Schroeder
- Kerberos added several improvements and extensions
  - Use of passwords in the protocol
  - Multiple authentication servers
  - Timestamps
- More on Kerberos later…
How can protocols go wrong?

- **Replay**
  - Attacker gets a copy of the messages exchanged
  - Attacker replays the messages
    - Might not be able to decrypt them!
  - Attacker Mallory tricks Alice into believing that Mallory is Bob
- To guard against this, make sure messages are complete
  - Guard all necessary data with encryption!
  - Never make *any* assumptions…

Examples of protocol problems

- What does the message $A, B, E_{KUA}(N_A)$ mean?
  - Message contains a nonce signed by A
  - Message is supposedly from A to B, but…
  - No signature guarding values A and B!
  - Intruder could replace B with C!
- Possible solutions
  - Encrypt (sign) entire message with $K_{UA}$
- Another problem: keeping track of nonces
  - How do you know when you’ve seen one before?
Kerberos

- Goal: scalable authentication services
  - Tens of thousands of users
  - Dozens (hundreds?) of servers
  - Make sure this won’t run too slow
- Problem: can’t use public-key crypto!
  - Relatively new (at the time Kerberos was designed)
  - Very slow
  - The protocol was designed without it…
- Build up protocol by starting with simple scenarios
- Provide additional protection to guard against potential security holes
- Eventual protocol is somewhat complex…

Kerberos Version 4, first try

- Authentication server (AS) knows all users’ passwords
  - Stored in a central database
- AS shares a unique secret key with each server
  - Keys distributed in a secure manner
- Ticket: proof of identity usable for server V
  - Frees servers from having to authenticate users
- Problem: user has to authenticate for each server
- Problem: password sent in the clear

C

AS

Ticket

V

C

IDc || Pc || IDv

C

IDc || Ticket

V

E{KV}[IDc || ADc || IDv]
Kerberos v4: second try

- Ticket granting server (TGS) issues tickets to those who have been authenticated by the AS
  - AS has lower load
  - Multiple TGS possible
- Lifetime & timestamp limit use of ticket
  - If lifetime is too short, AS and TGS will be deluged with requests
  - If lifetime is too long, replay attacks become a problem
- Problem: how can server authenticate itself to users?
  - Otherwise, fake servers could capture tickets and act as users...

\[
\text{Ticket}_{\text{X}} = E^{K_X}[\text{ID}_C || \text{AD}_C || \text{ID}_X || \text{TS} || \text{Lifetime}]
\]

\[
\text{TS} = \text{timestamp}
\]

\[
\text{C} \xrightarrow{\text{ID}_C \| \text{ID}_{\text{tgs}} \| E^{K_C}[\text{Ticket}_{\text{tgs}}]} \text{AS}
\]

\[
\text{C} \xrightarrow{\text{ID}_C \| \text{ID}_V \| \text{Ticket}_V} \text{TGS}
\]

\[
\text{C} \xrightarrow{\text{ID}_C \| \text{Ticket}_V} \text{V}
\]

Kerberos v4: actual protocol

- Additional security
  - Service using a ticket must be able to prove that person using a ticket is the one it was issued to
  - Solution: AS securely gives secret to both TGS and client
    - Client securely reveals secret to TGS
    - Mechanism: session keys
- Server verification
  - Client must be able to authenticate server

\[
\text{Ticket}_{\text{tgs}} = E^{K_{\text{tgs}}}[\text{K}_C, \text{ID}_C \| \text{AD}_C \| \text{ID}_{\text{tgs}} || \text{TS}_1 || \text{Lifetime}_2]
\]

\[
\text{Ticket}_V = E^{K_V}[\text{K}_C \| \text{ID}_V \| \text{AD}_C || \text{ID}_V \| \text{TS}_4 || \text{Lifetime}_4]
\]

\[
\text{Auth}_{C,X} = E^{K_{C,X}}[\text{ID}_C \| \text{ID}_X \| \text{AD}_C || \text{ID}_X \| \text{TS}_1]
\]

\[
\text{C} \xrightarrow{\text{ID}_C \| \text{ID}_{\text{tgs}} || \text{TS}_1 \| E^{K_C}[\text{Ticket}_{\text{tgs}}]} \text{AS}
\]

\[
\text{C} \xrightarrow{\text{Auth}_{C,X} \| \text{Ticket}_V} \text{TGS}
\]

\[
\text{C} \xrightarrow{\text{Auth}_V \| \text{Ticket}_V} \text{V}
\]
Kerberos v4 -> v5

- **Environmental differences**
  - Different encryption algorithms allowed
  - Different address specifications allowed
  - Lifetimes may be longer
  - Authentication forwarding allowed
  - Inter-realm operation streamlined

- **Technical differences**
  - Double encryption eliminated
  - Standard CBC encryption allowed
  - Subsession keys to defeat replay attacks

- **Remaining issues**
  - Password detection and replay

Summary

- **Authentication is a difficult problem**
  - We’ve only scratched the surface!
- **Use many tools for authentication**
  - Trees (or web) of trust
  - Public-key encryption
  - Message authentication codes
  - Passwords
- **Be careful!**
  - Passwords are easy to crack!
  - Good passwords are hard to choose!