Review: Authentication of humans

Categories: [IBM, TR G520-2169, 1970]

• **Something you know**
  password, passphrase, PIN, answers to security questions

• **Something you have**
  physical key, ticket, {ATM,prox,credit} card, token

• **Something you are**
  fingerprint, retinal scan, hand silhouette, a pulse
Password lifecycle

1. **Create:** user chooses password
2. **Store:** system stores password with user identifier
3. **Use:** user supplies password to authenticate
4. **Change/recover/reset:** user wants or needs to change password
4. PASSWORD CHANGE
Password change

Motivated by...

• **User** forgets password (maybe just *recover* password)

• **System** forces password expiration
  – Naively seems wise
  – Research suggests otherwise [see Cranor 2016]:
    • When users do change passwords, they change them predictably
    • Foreknowledge of expiration causes users to choose weaker passwords
Digression: Password research

Where to get password corpus for research?

- Pay users to participate in experiments
  - Validity? low-stakes passwords might be different than high-stakes
- Use \textit{cracked} password databases posted by attackers
  - Validity? you get only the (more) easily cracked passwords
- Participate with IT departments to run approved code against plaintext passwords
Password change

Motivated by...

• **Administrator** forces password change
  – Perhaps intrusion or weak password detected

• **Attacker** learns password:
  – **Social engineering:** deceitful techniques to manipulate a person into disclosing information
  – **Online guessing:** attacker uses authentication interface to guess passwords
  – **Offline guessing:** attacker acquires password database for system and attempts to crack it
Change mechanisms

• Tend to be **more vulnerable** than the rest of the authentication system
  – Not designed or tested as well
  – Have to solve the authentication problem without the benefit of a password

• Two common mechanisms:
  – Security questions
  – Emailed passwords
Security questions

- **Something you know:** attributes of identity established at enrollment
- **Pro:** you are unlikely to forget answers
- **Assumes:** attacker is unlikely to be able to answer questions
- **Con:** might not resist targeted attacks
- **Con:** linking is a problem; same answers re-used in many systems
Emailed password

• Might be your old password or a new temporary password
  – one-time password: valid for single use only, maybe limited duration
• Something you know: emailed password
• **Assumes:** attacker is unlikely to have compromised your email account
• **Assumes:** email service correctly authenticates you
• Something you <?>: however you authenticated to email
3. PASSWORD USAGE
When authentication fails

- **Guiding principle:** the system might be under attack, so don't make the attacker's job any easier
- Don't leak valid usernames:
  - Prompt for username and password in parallel
  - Don't reveal which was bad
- Rate limit, and eventually disable
- Record failed attempts and review
  - Perhaps in automated way by administrators
  - Perhaps manually by user at next successful login
Mutual authentication

• Before entering their password, the user ought to be authenticating the system itself: mutual authentication

• Some mechanisms:
  – Secure attention key: key (or key sequence) that OS itself detects and handles
    • e.g., Ctrl+Alt+Del in Windows
    • Defends against login spoofing
    • Provides a trusted path
  – Visual secrets: user and system share a secret image
    • User enters username; system retrieves and displays image
    • User authenticates image before entering password
    • Makes phishing attacks harder but not impossible: if users can't or won't discern who is on the other side, man-in-the-middle attack will succeed anyway
2. PASSWORD STORAGE
Storage by humans

• To keep identities independent, humans should have separate password for every identity
• But humans have little memory capacity
• So we...
  – reuse passwords across systems
  – record passwords either physically or digitally
  – both introduce vulnerabilities (come back to this next lecture)
Storage by machines

• Passwords typically stored in a file or database indexed by username

• **Strawman idea:** store passwords in plaintext
  – requires perfect authorization mechanisms
  – requires trusted system administrators
  – ...

• In the real world, password files get stolen
Storage by machines

• **Want:** a function \( f \) such that...
  1. easy to compute and store \( f(p) \) for a password \( p \)
  2. hard given disclosed \( f(p) \) for attacker to recover \( p \)
  3. hard to trick system by finding password \( q \) s.t. \( q \neq p \) yet \( f(p) = f(q) \) [stated incorrectly during lecture; now fixed]

• Cryptographic hash functions suffice!
  – one-way property gives (1) and (2)
  – collision resistance gives (3)

• So would encryption, but then the key has to live somewhere
Hashed passwords

- Each user has:
  - username uid
  - password p
- System stores: uid, H(p)
- Assume: human Hu authenticating to a local machine L over trusted secure channel (e.g., keyboard)

To authenticate Hu to L:
1. Hu→L: uid, p
2. L: let h = stored hashed password for uid;
   if h = H(p)
   then uid is authenticated
Hashed passwords

To authenticate Hu to remote server S using local machine L:

1. Hu→L: uid, p
2. L and S: establish secure channel
3. L→S: uid, p
4. S: let h = stored hashed password for uid;
   if h = H(p)
   then uid is authenticated
Hashed passwords

• Why not 3’. \( L \rightarrow S: \text{uid, } H(p) \)?

• Counterintuitive: From user’s perspective, sending plaintext password is better!
  – When password database leaked, 3’ immediately enables attacker to authenticate, whereas 3 forces attacker to invert hash

• From the two machines’ perspectives, about the same: one hash computation

• From DY adversary’s perspective, the same: can replay either message if security of channel is broken
Hashed passwords are still vulnerable

**Assume:** attacker does learn password file (offline guessing attack)

- Hard to invert: i.e., given $H(p)$ to compute $p$
- But what if attacker didn't care about inverting hash on arbitrary inputs?
  - i.e., only have to succeed on a small set of $p$'s: $p_1$, $p_2$, ..., $p_n$
- Then attacker could build a dictionary...
Dictionary attacks

Dictionary:
- p1, H(p1)
- p2, H(p2)
- ...
- pn, H(pn)

- **Dictionary attack:** lookup H(p) in dictionary to find p

- And *it works* because most passwords chosen by humans are from a relatively small set
Typical passwords

[Schneier quoting AccessData in 2007]:

• 7-9 character root plus a 1-3 character appendage
  — Root typically pronounceable, though not necessarily a real word
  — Appendage is a suffix (90%) or prefix (10%)

• Dictionary of 1000 roots plus 100 suffixes (= 100k passwords) cracks about 24% of all passwords
Typical passwords

[Schneier quoting AccessData in 2007]:

• More sophisticated dictionaries crack about 60% of passwords within 2-4 weeks

• Given biographical data (zip code, names, etc.) and other passwords of a user...
  – success rate goes up a little
  – time goes down to days or hours
Typical passwords

[Schneier quoting AccessData in 2007]:

• For comparison: a scan of every printable character string on your hard drive (including free space, swap files, etc.) breaks >50% of passwords
  – OS and applications leave secrets sitting around

...defense against offline guessing?
Defense 1: slow down

• **Vulnerability:** hashes are easy to compute

• **Countermeasure:** hash functions that are slow to compute
  
  – Slow hash wouldn't bother user: delay in logging hardly noticeable
  
  – But would bother attacker constructing dictionary: delay multiplied by number of entries
  
  – Ideally, enough to make constructing a large dictionary prohibitively expensive

• Examples: crypt, bcrypt, scrypt, PBKDF2, Argon2, ...
Slowing down fast hashes

• Given a fast hash function...
• Slow it down by iterating it many times:

\[ z_1 = H(p); \]
\[ z_2 = H(p, z_1); \]
\[ \ldots \]
\[ z_{1000} = H(p, z_{999}); \]
\[ \text{output } z_1 \text{ XOR } z_2 \text{ XOR } \ldots \text{ XOR } z_{1000} \]

• Number of iterations is a parameter to control slowdown
  – originally thousands
  – current thinking is 10s of thousands
• Aka key stretching
Defense 2: add salt

- **Vulnerability:** one dictionary suffices to attack every user
- **Vulnerability:** passwords chosen from small space
- **Countermeasure:** include a *unique system-chosen nonce* as part of each user's password
  - make every user's stored hashed password different, even if they chose the same password
  - make passwords effectively be from larger space
Salted hashed passwords

- Each user has:
  - username uid
  - unique salt $s$
  - password $p$
- System stores: $uid$, $s$, $H(s, p)$

To authenticate Hu to L:
1. $Hu \rightarrow L$: $uid$, $p$
2. $L$: let $h =$ stored hashed password for $uid$;
   
   let $s =$ stored salt for $uid$;
   
   if $h = H(s, p)$
   
   then $uid$ is authenticated
Salt

• Salt confidentiality:
  – Can be as public as username, though typically users don’t see it
  – Does not need to be secret, whereas password must be

• Salt needs to be unique even across systems; easiest way to achieve is to choose randomly

• Length of salt should be related to strength of cryptography employed in rest of system
Salt

To combine with iterated hashing, include salt in first hash:

\[ z_1 = H(p, s); \]
\[ z_2 = H(p, z_1); \]
\[ \ldots \]
\[ \ldots \]
\[ z_{1000} = H(p, z_{999}); \]
\[ \text{output } z_1 \text{ XOR } z_2 \text{ XOR } \ldots \text{ XOR } z_{1000} \]

this idea used in widely-deployed algorithm for deriving encryption keys from passwords... (next time)
Upcoming events

• [Wed] A3 due
• See today’s exercises for a way to win a free coffee

*Treat your password like your toothbrush. Don’t let anybody else use it.* – Clifford Stoll
SLOWING DOWN HASHES WITH SPACE

(we didn’t get to this in lecture)
Costly hashes

• Time is no longer *the* limiting factor
  – Custom ASICs
  – GPUs
  – Parallelize across the hardware

• Relevant to cryptocurrencies
**Costly hashes**

- **Space** is another scarce resource
  - Idea: provide configurable tradeoff of time vs. space required to compute hash
  - Technique: large number of computationally-expensive-to-produce random elements accessed in random order
    - user computing a single hash is okay with spending a lot of time and little space
    - attacker computing billions of hashes to construct dictionary wants to minimize time but would need large space for every hash, hence **hard to parallelize**