6.858 Lecture 17
USER AUTHENTICATION

Core challenge: How can human users prove their identity to a program?

- Is there any solution that totally dominates passwords?
- At first glance, passwords seem wretched.
  - Low entropy --> easy for attackers to guess them.
  - Back-up security questions for passwords are low entropy too.
  - A user often employs a single password for multiple sites.
- As the paper for today's class states, "The continued domination of passwords over all other methods of end-user authentication is a major embarrassment to security researchers."
- But... is there actually an authentication scheme which totally dominates passwords?
- Plan for today's lecture:
  1) See how current password schemes work.
  2) Talk about desirable properties for an authentication scheme.
  3) See how other authentication schemes compare to passwords.

A password is a secret that is shared between the user and a server.

- Naive implementation: server has a table that maps usernames to plaintext passwords.
- Problem: If attacker compromises the server, can recover all user/password pairs.
- Improved solution: Server stores this table:
  - user_name --> hash(user_password)
- User client supplies cleartext password to server, the server hashes the cleartext and does a table lookup.
- Advantage: Hash functions are difficult to invert, so it's difficult for attacker to
- perform a brute force attack. However...
- Problem: Attacker doesn't have to launch an inefficient brute force search over all possible passwords---the set of strings that are actually used as passwords is quite small!
  - Skewed distribution: top 5000 password values cover 20% of users.
  - Yahoo password study: Rule-of-thumb passwords contain 10-20 bits of entropy.
  - Hash functions are optimized performance; this helps attackers!
    - Ex: A laptop can calcuate SHA1s over small blocks at ~2M SHA1 ops/sec. Even if password has 20 bits of entropy, can crack one account/sec.
- Server can use a computationally expensive key-derivation function (e.g., PBKDF2 or BCrypt).
  - These functions have adjustable costs, so they can be arbitrarily slow.
  - Ex: can make hash cost be 1 second -- O(1M) times slower than SHA1.
  - Internally, often performs repeated hashing using a slow hash.

- Problem: adversary can build "rainbow tables".
    - Table of password-to-hash mappings.
    - Expensive to compute, but allows the attacker to efficiently invert hashes afterwards.
    - To maximize cost/benefit trade-off, the attacker only need to build a rainbow table for dictionary of common passwords.
    - Roughly: 1-second expensive hash
        - |-> 1M seconds = 10 days
    - to hash common pws. After that, can very quickly crack common passwords in any password db.

- Better solution: server can use password salts.
    - Input some additional randomness into the password hash: H(salt, pw).
    - Where does the salt value come from? It's stored on the server in plaintext.
    - Q: Why is this better if the adversary can compromise the salt too?
    - A: The attacker cannot use a single rainbow table to check for hash matches---the same password with different salts will have a different hash value!
    - Best practices:
        - Choose a long random salt.
        - Choose a fresh salt each time user changes password.

How should a client transmit a password to a server?

- Bad idea: send the password in the clear.
- Slightly better: send password over an encrypted connection.
- Drawback: Connection may be intercepted by an attacker who pretends to be the server (encryption doesn't necessarily mean that the server has authenticated to the client!). MITM attacker can then use the stolen password to impersonate the user.
- Q: What if client sends the hash of the password instead of the raw password?
- A: Doesn't provide us with any extra power, since the hash can still be replayed by the attacker.
- Encryption and hashing do not automatically add security---you need to think about what security properties you want to achieve, and specific ways that encryption and hashing can achieve those goals.
- Better idea: Challenge/response protocol.

```
Client             Server
     Hi, I'm Alice.
    ---------------->

       Challenge C
    <----------------

     H(C || password)
```

```
          ----------------->
                          -Server checks whether the
                           response is H(C || password).
```

- Ignoring MITM, the server is now confident that the user is Alice.
- If server is an attacker and didn't already know password, the attacker still doesn't know password.
    - Q: How can we prevent server from brute-force guessing password based on H() value?
    - A1: Expensive hash + salting.
    - A2: Allow client to choose some randomness too: guard against rainbow tables.
- To avoid storing the real password on the server, use a protocol like SRP:
    - http://en.wikipedia.org/wiki/Secure_Remote_Password_protocol
- High-level idea: Given a security parameter g, the client computes this...
    - v = g^(hash(salt, password))
- ...and sends v and the salt to the server. The client and the server can then establish an ephemeral key with their shared knowledge of g and v (protocol takes advantage of the fact that it's difficult for the attacker to perform discrete logarithms modulo N; RSA also takes advantage of this
- observation).
- Implementing challenge/response often means changing the client and the server.

To prevent brute force attacks, we can implement anti-hammering defenses.

- Ex: Rate-limit the number of password guesses; implement time-out periods after too many incorrect guesses.
- It's really important to throttle guess rate because passwords have so little entropy!
    - Many sites impose requirements on passwords (e.g., length, the use of special chars like punctuation).
    - In reality, what matters is entropy! Format requirements rarely translate into higher entropy.
    - A competent dictionary attacker can model password constraints and still generate rainbow tables; even with constraints, people will still pick passwords that adhere to a priori character distributions.
    - Telepathwords: https://telepathwords.research.microsoft.com/
        - As you type in a potential password letter, tries to guess the next letter using heuristics!
        - Common passwords (e.g., via leaks of password databases)
        - Popular phrases from web sites
        - Common user biases in selecting characters (e.g., using adjacent keys for adjacent password characters)
- Kerberos v4, and v5 without preauth were vulnerable to offline guessing
    - http://www.gnu.org/software/shishi/wu99realworld.pdf

    - Anyone could ask the KDC for a ticket encrypted with the user's password, i.e., the KDC did not authenticate requests (although the response would be encrypted with K_c).
    - Attacker can try brute-force to guess the user's password---this is easy to parallelize. Since the ticket-granting-ticket has a known format, the attacker can determine when a decryption is successful.
    - In Kerberos v5, ticket requestor must include { timestamp }_{K_c} along with request, to prove knowledge of K_c.

Password recovery is extremely important, but often overlooked.

- People often focus on the entropy of passwords, but if recovery questions can be used to reset passwords, the strength of a password authentication scheme is min(password_entropy, recovery_question_entropy).
- Recovery questions are often easily guessable. In one famous example, somebody got access to Sarah Palin's Yahoo address by guessing answers
- to her security questions.
    - Intrinsically low entropy ("What's your favorite color?" "What's the
- name of your best friend?")
    - Answers leaked via social media profiles ("What's your favorite movie?")
    - Self-generated questions are typically easy to answer ("What is 5 + 5?")

In the reading for today, the authors propose a bunch of factors that can be used to evaluate authentication schemes (the goal is to determine whether passwords are as bad as they seem). The authors consider three high-level metrics: usability, deployability, and security.

- Usability: How easy is it for users to interact with the authentication scheme?
    - Easy-to-Learn: "Users who don't know the scheme can figure it out and learn it without too much trouble."
        - This is a key reason why password schemes are so popular!
    - Infrequent errors: "The task that users must perform to log in usually succeeds when performed by a legitimate and honest user."
        - This is an important reason why users pick easy-to-guess passwords.
    - Scalable-for-Users: "Using the scheme for hundreds of accounts does not increase the burden on the user."
        - ...explains why people often reuse passwords or create a simple per-site uniquifying scheme for a base password.
    - Easy recovery from loss of the authentication token
        - A win for passwords--they're easy to reset.
    - Nothing to carry
        - Another win for passwords.

- Deployability: How easy is it to incorporate the authentication method into real systems?

    - Server-Compatible: "At the verifier's end, the scheme is compatible with text-based passwords. Providers don't have to change their existing authentication setup to support the scheme."
    - Browser-Compatible: "Users don't have to change their client to support the scheme . . . schemes fail to provide this benefit if they require the installation of plugins or any kind of software whose installation requires administrative privileges."
    - Accessible: "Users who can use passwords are not prevented from using the scheme by disabilities or other physical (not cognitive) conditions."

    - Deployability is extremely difficult: it's difficult to get users or servers to update en masse!
    - Passwords do well in this category by default, since the authors define "deployability" as how well a system integrates with current password infrastructure. However, passwords don't do very well in the next category…

- Security: What kinds of attacks can the authentication scheme prevent?
    - Resilient-to-Physical-Observation: "An attacker cannot impersonate a user after observing them authenticate one or more times. We grant Quasi-Resilient-to-Physical-Observation if the scheme could be broken only by repeating the observation more than, say, 10--20 times. Attacks include shoulder surfing, filming the keyboard, recording keystroke sounds, or thermal imaging of keypad."
        - Passwords fail this test, since, e.g., they can be captured by filming the keyboard or recording keystroke sounds.
    - Resilient-to-Targeted-Impersonation: "It is not possible for an acquaintance (or skilled investigator) to impersonate a specific user by exploiting knowledge of personal details (birth date, names of relatives etc.). Personal knowledge questions are the canonical scheme that fails on this point."
        - The authors say that passwords are "quasi-resistant" b/c they couldn't find any studies saying that your friends or acquaintances can easily guess your password.
    - Resilient-to-Throttled-Guessing: "An attacker whose rate of guessing is constrained by the verifier cannot successfully guess the secrets of a significant fraction of users… Lack of this benefit is meant to penalize schemes in which it is frequent for user-chosen secrets to be selected from a small and well-known subset."
        - Passwords fail because they have low entropy + skewed distributions.
    - Resilient-to-Unthrottled-Guessing: "An attacker whose rate of guessing is constrained only by available computing resources cannot successfully guess the secrets of a significant fraction of users. We might for example grant this benefit if an attacker capable of

attempting up to 2^40 or even 2^64 guesses per account could still only reach fewer than 1% of accounts. Lack of this benefit is meant to penalize schemes where the space of credentials is not large enough to withstand brute force search from a small and well-known subset."
    - Passwords fail because they have low entropy + skewed distributions.
- Resilient-to-Internal-Observation: "An attacker cannot impersonate a user by intercepting the user's input from inside the user's device (e.g., by keylogging malware) or eavesdropping on the cleartext communication between prover and verifier (we assume that the attacker can also defeat TLS if it is used, perhaps through the CA)…This penalizes schemes that are not replay-resistant, whether because they send a static response or because their dynamic response countermeasure can be cracked with a few observations. This benefit assumes that general-purpose devices like software-updatable personal computers and mobile phones may contain malware, but that hardware devices dedicated exclusively to the scheme can be made malware-free."
    - Passwords fail because they are static tokens: once you have one, you can use it until it expires or is revoked.
- Resilient-to-Phishing: "An attacker who simulates a valid verifier (including by DNS manipulation) cannot collect credentials that can later be used to impersonate the user to the actual verifier. This penalizes schemes allowing phishers to get victims to authenticate to look-alike sites and later use the harvested credentials against the genuine sites."
    - Passwords fail: phishing attacks are very common!
- *No-Trusted-Third-Party: "The scheme does not rely on a trusted third party (other than the prover and the verifier) who could, upon being attacked or otherwise becoming untrustworthy, compromise the prover's security or privacy."
    - This property makes an important point: a lot of authentication problems would become easier if we could just trust one party to store passwords, run the password servers, etc. However, single points of failure are bad, since attackers can focus all of their energy on that point!
- *Resilient-to-Leaks-from-Other-Verifiers: "Nothing that a verifier could possibly leak can help an attacker impersonate the user to another verifier. This penalizes schemes where insider fraud at one provider, or a successful attack on one back-end, endangers the user's accounts at other sites."
    - This property is related to No-Trusted-Third-Party. To avoid a central point of failure, we'd like to introduced some notion of distributed authentication: however, does this mean that the system is only as strong as its weakest link? [Think back to HTTPS, and how a bad certificate authority can convince a

browser to accept fake certificates for arbitrary sites. Security depends on the strength of the least secure CA!]

- Authors say that passwords fail because people often reuse passwords across sites.

Biometrics: Leverage the unique aspects of a person's physical appearance or behavior.

- How big is the keyspace?
    - Fingerprints: ~13.3 bits.
    - Iris scan: ~19.9 bits.
    - Voice recognition: ~11.7 bits.
- So, bits of entropy are roughly the same as passwords.

	Passwords	Biometrics
Easy-to-learn:	Yes	Yes
Infrequent errors:	Quasi-yes	No
Scalable for users:	No	Yes
Easy recovery:	Yes	No
Nothing to carry:	Yes	Yes

3.5 vs 3

	Passwords	Biometrics
Server-compatible:	Yes	No
Browser-compatible:	Yes	No
Accessible:	Yes	Quasi-yes

(entering biometrics is error-prone)

3 vs 0.5

	Passwords	Biometrics
Res-to-Phys-Obs:	No	Yes
Res-to-Trgtd-Imp:	Quasi-yes	No (e.g., replaying voice recording, lifting fingerprints from surfaces)
Res-to-Thrtld-Guess:	No	Yes
Res-to-UnThrtld-Guess:	No	No (key space isn't much bigger than that of passwords)
Res-to-Internal-Obv:	No	No (captured biometric data can be replayed)
Res-to-Phishing:	No	No
No-trusted-3rd-Party:	Yes	Yes
Res-Other-Ver-Leaks:	No	No (same biometrics are used by all verifiers)

1.5 vs 3

So, final score is 8 vs 6.5. Of course, one could assign non-unity weights to each category, but the point is that it's not obvious that biometrics are "better" than passwords!

Some sets of goals seem difficult to achieve at the same time.

- Memorywise-Effortless + Nothing-to-Carry.
- Memorywise-Effortless + Resilient-to-Theft.

Either the user remembers something, or it can be stolen (except for biometrics).

- Server-Compatible + Resilient-to-Internal-Observation.
- Server-Compatible + Resilient-to-Leaks-from-Other-Verifiers.

Server compatible means sending a password. Passwords can be stolen on user machine, replayed by one server to another.

Multi-factor authentication (MFA): defense using depth.

- Requires users to authenticate themselves using two or more authentication mechanisms.
- The mechanisms should involve different modalities!
    - Something you know (e.g., a password)
    - Something you possess (e.g., a cellphone, a hardware token)
    - Something you are (e.g., biometrics)
- Idea is that an attacker must steal/subvert multiple authentication mechanisms to impersonate a user (e.g., attacker might guess a password, but lack access to a user's phone).
- Example: Google's two-factor authentication requires a password plus a cellphone which can receive authorization codes via text
- message.
- Example: AWS two-factor authentication requires a password and an "MFA device" (a smartphone running an authentication app, or a special-purpose security token or security card).
    - http://aws.amazon.com/iam/details/mfa/
- MFA is a good idea, but empirical studies show that if users are given a second authentication factor in addition to passwords, users pick much weaker passwords!

What are potential answers to the homework questions? What factors matter?

- Logging into public Athena machine?
    - Resilient-to-Internal-Observation: easy to install malware on machine.
    - Resilient-to-Physical-Observation?
    - MIT IDs could be a good thing to leverage (use them as a smartcard).
    - Biometrics? Untrusted terminals, probably not a great plan.
- Accessing Facebook from Internet cafe?
    - Password managers not a good idea here.
    - How sensitive is the data?

        - Might be leveraged to authenticate to other sites! [Either "Login with Facebook" or by answering personal security questions to reset a password.]

- Withdrawing cash from ATM?
    - Security matters highly.
        - Resilient-to-Physical-Observation.
        - Resilient-to-Theft.
    - Possibly trusted terminal: biometrics might be worth considering. [However, in practice, bank may not want to trust the terminals.]
    - You also might care about authenticating individual transactions!
        - Prevent the adversary from using stolen credentials for different, attacker-chosen operations.
        - Ex: Maybe user can examine balance using just a password, but if she wants to withdraw money, she uses two-factor authentication using her phone.

Conclusion of paper: There is no authentication scheme which clearly dominates passwords! For example, according to the authors, the CAP reader has perfect scores on security!

- The CAP reader was designed by Mastercard to protect online banking transactions.
- Usage:
    1) Put your credit card into the CAP reader (which looks like a hand-held calculator).
    2) Enter PIN (bypassing keyloggers!).
    3) Reader talks to the card's embedded processor, outputs an 8-digit code which the user supplies to the web site.

```
                        CAP reader
    Easy-to-learn:      Yes
    Infrequent errors:  Quasi-yes
    Scalable for users: No (users require card+PIN per
verifier)
    Easy recovery:      No
    Nothing to carry:   No
                             1.5

                        CAP reader
   Server-compatible:   No
   Browser-compatible:  Yes
   Accessible:          No (blind people can't read 8-digit
code)
                              1

                        CAP reader
Res-to-Phys-Obs:        Yes\
Res-to-Trgtd-Imp:        Yes \__ One-time codes!
Res-to-Thrtld-Guess:     Yes /
Res-to-UnThrtld-Guess:  Yes/
Res-to-Internal-Obv:     Yes     Dedicated device
Res-to-Phishing:         Yes     One-time codes
No-trusted-3rd-Party:   Yes     Each site is its own
verifier
Res-Other-Ver-Leaks:    Yes     One-time codes
           8
```

- So, passwords=8 and CAP reader=10.5. However, there are reasons why CAP readers haven't taken over the world (see the low usability and deployability scores).
- In practice, deployability and usability are often more important than security.
    - o Migration costs (coding+debugging effort, user training) make developers nervous!
    - o The less usable a scheme is, the more that users will complain (and try to pick easier authentication tokens that are more vulnerable to attackers).
- Some situations may assign different weights to different evaluation metrics.
    - o Ex: On a military base, the security benefits of a hardware-based token might outweigh the problems with usability and deployability.