50.020 Security
Lecture 5: Password and Rainbow Tables

Password-based Authentication

Terminology

- Access control
  - Allow/deny users access to resources
  - Sometimes, delegation is possible
- Authentication verifies correctness of data and source
  - In this context: verifying the *identity* of login request
  - *identification* itself does not include verification

Identification schemes

Identifaction only needs to provide an identity

- No direct security requirements
- Identifier should not be secret
  - E.g.: Social security numbers (bad)

Common Identification schemes:

Identification schemes

Identifaction only needs to provide an identity

- No direct security requirements
- Identifier should not be secret
    - E.g.: Social security numbers (bad)

Common Identification schemes:

- Email addresses
- Usernames/Real names
- Phone numbers
- Credit card numbers
- Bank cards (IDs on cards)
- National ID number, FIN, etc.
- Face recognition
    - Most other biometrics not good enough

Authentication schemes

- Which authentication schemes do you know?

Authentication schemes

- Which authentication schemes do you know?

- Knowledge-based authentication
  - Passwords
  - Patterns (Android lock screen)
  - Banking PINs
- Token-based authentication
  - Physical keys
  - Cryptographic tokens (e.g., from bank)
  - Certificate-based (e.g., TPM)
- Biometrics-based authentication
  - Fingerprint, voice, retina
- Multi-factor authentication

Password-based Authentication

Passwords are the most common way to authenticate users

Password-based Authentication

Passwords are the most common way to authenticate users

- Advantages:
  - Can be changed
  - User is free to choose
- Disadvantages
  - Can be forgotten
  - User can create bad passwords
  - Can be re-used

Password-based Authentication

Passwords are the most common way to authenticate users

- Advantages:
  - Can be changed
  - User is free to choose
- Disadvantages
  - Can be forgotten
  - User can create bad passwords
  - Can be re-used

- But: how to remember 30+ passwords?

Why is guessing passwords so easy (compared to keys)?

- Passwords use string.printable
- Passwords are somewhat short
- Some passwords are used more frequently
  - Or have frequently used components
- This enables semi-intelligent brute-forcing
  - dictionaries
  - hybrid attacks

Dictionary attacks

- Users prefer simple passwords
- Dictionary attacks produce lists of popular passwords
- Ordered by popularity
- This maximises likelyhood of success with minimal tries
- Often based on sets of passwords that became public

Rank	Password
1	123456
2	password
3	12345678
4	qwerty
5	abc123
6	123456789
7	111111
8	1234567
9	iloveyou
10	adob123
...	...

Popular Passwords 2013 (according to SplashData)

Hybrid attacks

- Users have heard about dictionary attacks
- "p4sSw0Rd" might not be in dictionary
  - But it is still pretty similar
- Hybrid attacks also try combinations and popular substitutions
  - E.g. replacements such as "$a->4$" and "$o->0$", case
- Interesting estimation of effort for attacks:

`https://www.bennish.net/password-strength-checker/`

Finding Passwords in practise

- Both dictionary attacks and hybrid attacks can be used to build long lists of likely passwords
- If there is an API to submit unlimited password attempts, this could be called to break into a system
  - In practise, accounts are quickly locked to prevent this
- In most cases, dictionary and hybrid attacks are used to attempt to find preimages of hashes
  - Password hashes were stolen in some attack
  - Attacker has *unlimited attempts* to find preimage
- Is your account compromised? Check at `https://haveibeenpwned.com/`

HaveIBeenPwned Example

Breaches you were pwned in

A "breach" is an incident where a site's data has been illegally accessed by hackers and then released publicly. Review the types of data that were compromised (email addresses, passwords, credit cards etc.) and take appropriate action, such as changing passwords.

Last.fm: In March 2012, the music website Last.fm was hacked and 43 million user accounts were exposed. Whilst Last.fm knew of an incident back in 2012, the scale of the hack was not known until the data was released publicly in September 2016. The breach included 37 million unique email addresses, usernames and passwords stored as unsalted MD5 hashes.

Compromised data: Email addresses, Passwords, Usernames, Website activity

LinkedIn: In May 2016, LinkedIn had 164 million email addresses and passwords exposed. Originally hacked in 2012, the data remained out of sight until being offered for sale on a dark market site 4 years later. The passwords in the breach were stored as SHA1 hashes without salt, the vast majority of which were quickly cracked in the days following the release of the data.

Compromised data: Email addresses, Passwords

MySpace: In approximately 2008, MySpace suffered a data breach that exposed almost 360 million accounts. In May 2016 the data was offered up for sale on the "Real Deal" dark market website and included email addresses, usernames and SHA1 hashes of the first 10 characters of the password converted to lowercase and stored without a salt. The exact breach date is unknown, but analysis of the data suggests it was 8 years before being made public.

Compromised data: Email addresses, Passwords, Usernames

Patreon: In October 2015, the crowdfunding site Patreon was hacked and over 16GB of data was released publicly. The dump included almost 14GB of database records with more than 2.3M unique email addresses and millions of personal messages.

Compromised data: Email addresses, Payment histories, Private messages, Website activity

Strengthening Passwords

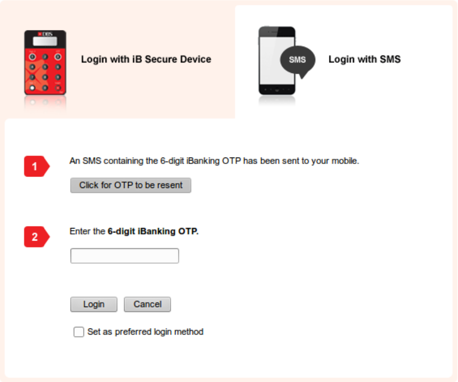
Multi-factor authentication (MFA)

- Simple, most commong form: Two-factor authentication (TFA)
- Combines username/password with second way, e.g. text messages
  - Example: DBS login into account

First factor login

Multi-factor authentication (MFA)

- Simple, most commong form: Two-factor authentication (TFA)
- Combines username/password with second way, e.g. text messages
  - Example: DBS login into account



Multi-factor authentication (MFA)

- Simple, most commong form: Two-factor authentication (TFA)
- Combines username/password with second way, e.g. text messages
  - Example: DBS login into account
- MFA is an application of "defense in depth"

Summary: "Best practises" for passwords

- Do not re-use the same password between two services
- Change passwords periodically (but not too often)
- Do not base your passwords on dictionary words
  - Dialects or made-up words are better
- How to find good passwords?
  - Find a good scheme to adapt your password to each site
  - Example: "p4ssw0rd(amazon)", "p4ssw0rd(sutd)" etc
- Use random passwords, and management software
- Use hashes/ your scheme here

Finding Hash Preimages

Brute Forcing Hashes

- Common cryptographic hashes have length 128 (e.g., MD5), 160 (SHA1), 224-512(SHA3)
- Brute-forcing SHA1 takes about $O(2^{160})$ computations
- How could we speed this up?
  - Precompute some/all values!
- If we precompute 2^{160} hashes, we can directly look up preimage
  - Unfortunately, this takes $160 * 2^{120}$ Terabyte of storage

Improving Brute Force

- So clearly, computing hashes on demand or full precomputation is infeasible
- Can we mix both?
  - Do precomputations, but only store a subset of the found hashes
  - Just store as many hash values as you have storage space
  - Ensure that you can recover preimage of the hash values
- This is the idea behind rainbow tables, which are based on hash chains

Improving Brute Force (continue)

- In the following, we ignore *hash collisions* to simplify things
- We also assume that the *input space* is smaller than the *output space*, e.g. if only 10 character inputs are considered. . .
- Note: Rainbow tables are not computed "on the fly" to look up one hash
  - Direct brute force would be more efficient in that case
  - Rainbow tables allow you to re-use brute force effort for many hashes

Hash Chains

- Hash chains trade storage space vs. computational effort
- General operations of hash chain:
  - H() is hashing function from input domain to hash domain
  - R() is some reduction function, mapping from hash to *input domain*
    - If you only care about string.printable of length < 10, then R() should map into that
- Lets initiate a hash chain with I_1 as first input

$$I_1 \xrightarrow{H()} O_1 \xrightarrow{R()} I_2 \xrightarrow{H()} O_2 \xrightarrow{R()} \cdots \xrightarrow{H()} O_t$$

- After t operations, we get hash output O_t
- If we store I_1 and O_t, how can we find I_t?

Hash Chains

- Hash chains trade storage space vs. computational effort
- General operations of hash chain:
  - H() is hashing function from input domain to hash domain
  - R() is some reduction function, mapping from hash to *input domain*
    - If you only care about string.printable of length < 10, then R() should map into that
- Lets initiate a hash chain with I_1 as first input

$$I_1 \xrightarrow{H()} O_1 \xrightarrow{R()} I_2 \xrightarrow{H()} O_2 \xrightarrow{R()} \cdots \xrightarrow{H()} O_t$$

- After t operations, we get hash output O_t
- If we store I_1 and O_t, how can we find I_t?

- We re-compute the chain with I_1 until we hit I_t

Rainbow tables

- Rainbow tables trade time vs. space in hash reversal
- A rainbow table consists of m hash chains of length t
- For each chain, only the first input (i.e., plaintext) I_1 and the last hash O_t are stored.
- Overall space requirement: $(|I| + |O|) * m$
- Even more space-efficient:
  - Each of the m chains can use its index as starting input I
  - For each chain, only the last hash O_t is stored
  - Overall space requirement: $|O| * m$
- The product $m * t$ must be $\geq$ number of possible input values
  - E.g. if 10 characters [a-Z]: 26^{10}
- Runtime of hash lookup: $O(t/2 + t/2)$ if comparisons are free, and hashing is only expensive operation

Rainbow table operation

- How to use hash chain for lookup: We want to find $X : Y = H(X)$
- Check if Y is in list of last chain elements
  - if yes ($Y = O_z$): regenerate chain z using the input value I_z. Then find the I_z that was used to compute O_z, this is our X
  - if no: compute H(R(Y))=Y', see if this is in list of last chain elements
    - if yes ($Y' = O_z$): regenerate chain z using the input value I_z. Then find the I_z that was used to compute O_z, this is our X
    - if no: apply further iterations of reduction and hashing on Y

Rainbow table operation (continue)

- Estimated effort: (only counting hashing, as most expensive op)
  - expected $t/2$ reductions to find a matching last value + $t/2$ average effort to regenerate chain$\Rightarrow$ $O(t/2 + t/2)$
- Brute force effort
  - Either $O(n/2 = m * t/2)$ (on-the-fly computation) or
  - $O(1)$ computation and $O(m * t)$ space

RainbowCrack

- Rainbow tables can be generated using RainbowCrack tool for MD5, SHA1, NTLM, . . .
- If you have enough space and time, you can open a commercial rainbowtable:
  - https://www.cloudcracker.com/ [was still working in 2017, now offline]
  - $17 per hash lookup

Defending against Rainbow tables

To make rainbow tables infeasible, a salt (random number) is added to each hash

- E.g.: $x = H(m||s)$ with salt s
- The salt can be stored with hashed password (x,s)
- The salt should be different for each user
- What is the benefit?
- The attacker cannot just use the same rainbow table
- If attacker would want to pre-compute rainbow tables: n bit salt increases effort for attacker by 2^n
  - Each salt requires own rainbow table of same size as original one
- Some people say rainbow tables are dead. . .

Hashcat

- If salts are used, brute force might be the only solution to find preimages
  - In particular, if non-salt part of input is from small space
  - In particular, if hashing function is computationally cheap
- Effort for the attacker directly depends on cost of hash
  - If hashing can be done in 1% of time, attack is 100 times faster
  - Bitcoin caused a lot of specialized hashing hardware to appear
  - Modern GPUs can also be used for hashing (e.g. NVIDIA CUDA)
- Hashcat is an example tool to do such online attacks
  - http://hashcat.net/oclhashcat/
  - Hashcat leverages GPUs for hashing using OpenCL

Detecting compromise

Honeywords

- How to detect if an attacker is trying to break into your system?
- Create password entries for dummy users
- Trigger alarms when these users try to log in
- Trigger big alarm if these users log in with correct password!
- More reading by Rivest:

`http://people.csail.mit.edu/rivest/honeywords/`

Conclusions

- Password-based authentication
- Strengthening password
- Finding Hash pre-images (rainbow table, etc.)