

Cristina Nita-Rotaru



CY2550: Foundations of Cybersecurity

Section 03

Authentication Module



Authentication

Authentication

Definition:

Authentication is the process of verifying an actor's identity

- ▶ Critical for security of systems
 - ▶ Permissions, capabilities, and access control are all contingent upon knowing the identity of the actor
- ▶ Typically parameterized as a **username** and a **secret**
 - ▶ The secret attempts to limit unauthorized access
- ▶ Desirable properties of secrets include being *unforgeable*, *unguessable*, and *revocable*

Types of secrets

- ▶ Actors provide their secret to **log-in** to a system
- ▶ Three classes of secrets:
 1. Something you know
 - ▶ Example: a password
 2. Something you have
 - ▶ Examples: a smart card or smart phone
 3. Something you are
 - ▶ Examples: fingerprint, voice scan, iris scan




Password storage

Attacker goals and threat model

- ▶ Assume we have a system storing usernames and passwords
- ▶ The attacker has access to the password database/file

Database



User	Password
cbw	p4ssW0rd
sandi	puppies
amislove	3spr3ss0



I wanna login to those user accounts!

Cracked Passwords

User	Password
cbw	p4ssW0rd
sandi	puppies
amislove	3spr3ss0

Authentication

Checking passwords

- ▶ System must validate passwords provided by users
- ▶ Thus, passwords must be stored somewhere
- ▶ Basic storage: plain text

password.txt	
cbw	p4ssw0rd
sandi	i heart doggies
amislove	93Gd9#jv*0x3N
bob	security

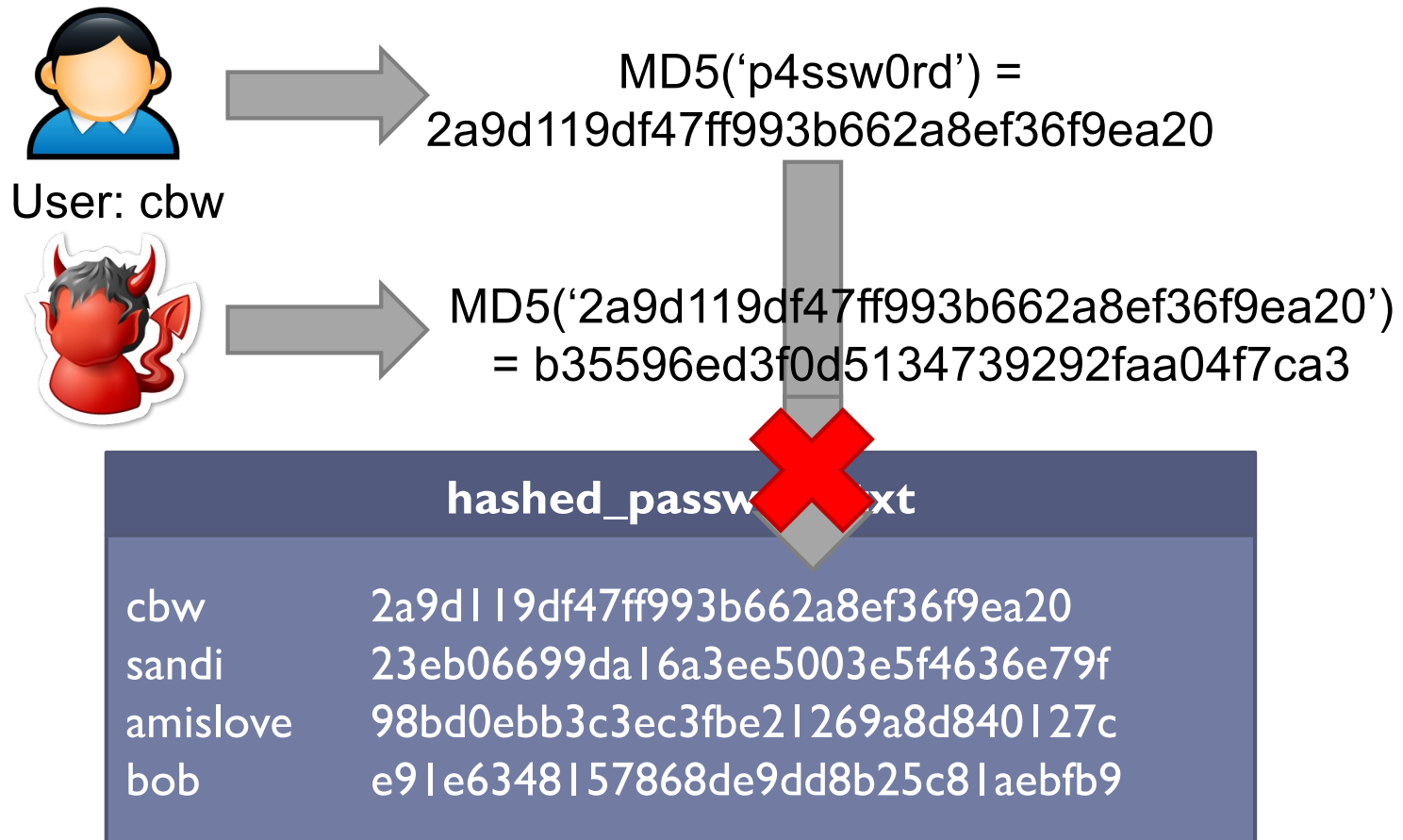
Problem: Password file theft

- ▶ Attackers often compromise systems
- ▶ They may be able to steal the password file
 - ▶ Linux: /etc/shadow
 - ▶ Windows: c:\windows\system32\config\sam
- ▶ If the passwords are plain text, what happens?
 - ▶ The attacker can now log-in as any user, including root/administrator
- ▶ **Passwords should never be stored in plain text**

Hashed passwords

- ▶ **Key idea: store hashed versions of passwords**
 - ▶ Use one-way cryptographic hash functions
 - ▶ Examples: MD5, SHA1, SHA256, SHA512, bcrypt, PBKDF2, scrypt
- ▶ **Cryptographic hash function transform input data into scrambled output data**
 - ▶ Deterministic: $\text{hash}(A) = \text{hash}(A)$
 - ▶ High entropy:
 - ▶ MD5('security') = e91e6348157868de9dd8b25c81aebfb9
 - ▶ MD5('security1') = 8632c375e9eba096df51844a5a43ae93
 - ▶ MD5('Security') = 2fae32629d4ef4fc6341f1751b405e45
 - ▶ Collision resistant
 - ▶ Locating A' such that $\text{hash}(A) = \text{hash}(A')$ takes a long time
 - ▶ Example: 2^{21} tries for md5

Hashed password example



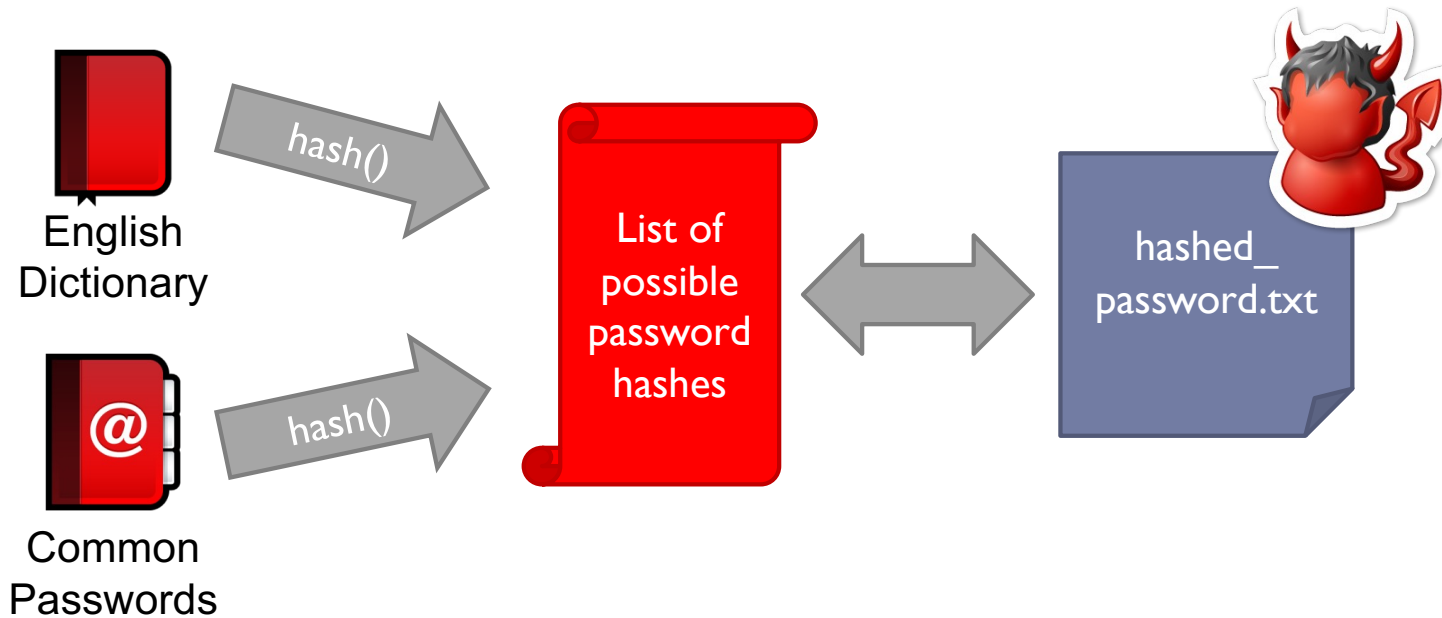
Attacking password hashes

- ▶ Recall: cryptographic hashes are collision resistant
 - ▶ Locating A' such that $\text{hash}(A) = \text{hash}(A')$ takes a long time
- ▶ Are hashed password secure from cracking?
 - ▶ **No!**
- ▶ Problem: users choose poor passwords
 - ▶ Most common passwords: 123456, password
 - ▶ Username: cbw, Password: cbw
- ▶ Weak passwords enable dictionary attacks

Most common passwords

Rank	2013
1	123456
2	password
3	12345678
4	qwerty
5	abc123
6	123456789
7	111111
8	1234567
9	iloveyou
10	adobe123

Dictionary attacks



- ▶ Common for 60-70% of hashed passwords to be cracked in <24 hours

Hardening password hashes

- ▶ **Key problem: cryptographic hashes are deterministic**
 - ▶ $\text{hash}(\text{'p4ssw0rd'}) = \text{hash}(\text{'p4ssw0rd'})$
 - ▶ This enables attackers to build lists of hashes
- ▶ **Solution: make each password hash unique**
 - ▶ Add a random **salt** to each password before hashing
 - ▶ $\text{hash}(\text{salt} + \text{password}) = \text{password hash}$
 - ▶ Each user has a unique, random salt
 - ▶ Salts can be stores in plain text
 - ▶ Salts should be as big as the output of the hash function

Example salted hashes

hashed_password.txt

cbw	2a9d119df47ff993b662a8ef36f9ea20
sakib	23eb06699da16a3ee5003e5f4636e79f
amislove	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

hashed_and_salted_password.txt

cbw	a8	af19c842f0c781ad726de7aba439b033
sakib	0X	67710c2c2797441efb8501f063d42fb6
amislove	hz	9d03e1f28d39ab373c59c7bb338d0095
bob	K@	479a6d9e59707af4bb2c618fed89c245



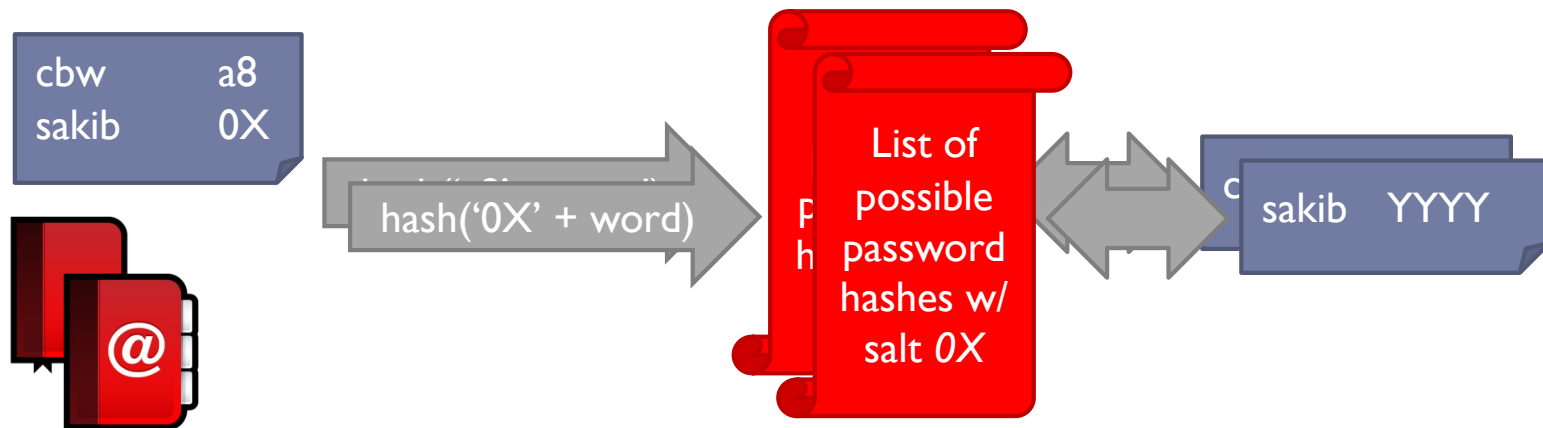
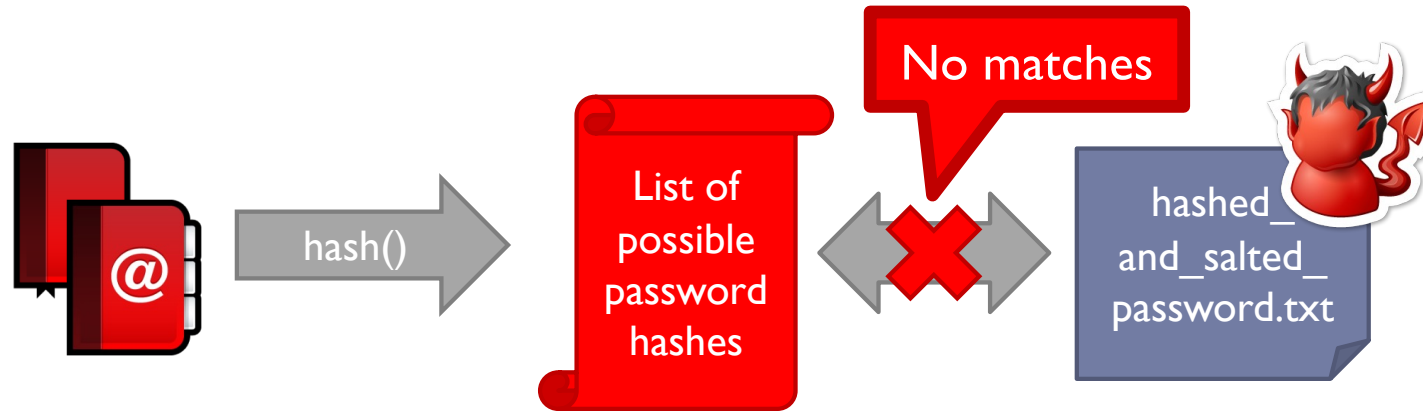
User: cbw



MD5('a8' + 'p4ssw0rd') =
af19c842f0c781ad726de7
aba439b033



Attacking salted passwords



Breaking hashed passwords

- ▶ **Stored passwords should always be salted**
 - ▶ Forces the attacker to brute-force each password individually
- ▶ **Problem: it is now possible to compute hashes very quickly**
 - ▶ GPU computing: hundreds of small CPU cores
 - ▶ nVidia GeForce GTX Titan Z: 5,760 cores
 - ▶ GPUs can be rented from the cloud very cheaply
 - ▶ \$0.9 per hour (2018 prices)

Examples of hashing speed

- ▶ A modern x86 server can hash all possible 6 character long passwords in 3.5 hours
 - ▶ Upper and lowercase letters, numbers, symbols
 - ▶ $(26+26+10+32)^6 = 690$ billion combinations
- ▶ A modern GPU can do the same thing in 16 minutes
- ▶ Most users use (slightly permuted) dictionary words, no symbols
 - ▶ Predictability makes cracking much faster
 - ▶ Lowercase + numbers $\rightarrow (26+10)^6 = 2B$ combinations

Hardening salted passwords

- ▶ Problem: typical hashing algorithms are too fast
 - ▶ Enables GPUs to brute-force passwords
- ▶ Old solution: hash the password multiple times
 - ▶ Known as *key stretching*
 - ▶ Example: *crypt* used 25 rounds of DES
- ▶ New solution: use hash functions that are designed to be **slow**
 - ▶ Examples: bcrypt, PBKDF2, scrypt
 - ▶ These algorithms include a *work factor* that increases the time complexity of the calculation
 - ▶ scrypt also requires a large amount of memory to compute, further complicating brute-force attacks

bcrypt example

```
[cbw@localhost ~] python
>>> import bcrypt
>>> password = "my super secret password"
>>> fast_hashed = bcrypt.hashpw(password, bcrypt.gensalt(0))
>>> slow_hashed = bcrypt.hashpw(password, bcrypt.gensalt(12))
>>> pw_from_user = raw_input("Enter your password:")
>>> if bcrypt.hashpw(pw_from_user, slow_hashed) ==
slow_hashed:
...     print "It matches! You may enter the system"
... else:
...     print "No match. You may not proceed"
```



Work factor

Dealing with breaches

- ▶ Suppose you build an extremely secure password storage system
 - ▶ All passwords are salted and hashed by a high-work factor function
- ▶ It is still possible for a dedicated attacker to steal and crack passwords
 - ▶ Given enough time and money, anything is possible
 - ▶ E.g. The NSA
- ▶ Question: is there a principled way to detect password breaches?

Honeywords

- ▶ **Key idea: store multiple salted/hashed passwords for each user**
 - ▶ As usual, users create a single password and use it to login
 - ▶ User is unaware that additional **honeywords** are stored with their account
- ▶ **Implement a **honeyserver** that stores the index of the correct password for each user**
 - ▶ Honeyserver is logically and physically separate from the password database
 - ▶ Silently checks that users are logging in with true passwords, not honeywords
- ▶ **What happens after a data breach?**
 - ▶ Attacker dumps the user/password database...
 - ▶ But the attacker does not know which passwords are honeywords
 - ▶ Attacker cracks all passwords and uses them to login to accounts
 - ▶ If the attacker logs-in with a honeyword, the honeyserver raises an alert!

Honeywords example



cbw

SHA512("fl" | "p4ssW0rd") → bHDJ8l



Cracked Passwords

User	PW 1	PW 2	PW 3
cbw	123456	p4ssW0rd	Turtles!
sandi	puppies	iloveyou	blizzard
amislove	coff33	3spr3ss0	qwerty

Database



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	Salt 3	H(PW 3)
cbw	aB	y4DvF7	fl	bHDJ8l	52	Puu2s7
sandi	0x	pIDS4F	K2	R/p3Y8	8W	S8x4Gk
amislove	9j	0F3g5H	/s	03d5jW	cV	lsRbj5

Honeyserver



User	Index
cbw	2
sandi	3
amislove	1

Authentication

Password storage summary

- 1. Never store passwords in plain text**
 - 2. Always salt and hash passwords before storing them**
 - 3. Use hash functions with a high work factor**
- ▶ These rules apply to any system that needs to authenticate users
 - ▶ Operating systems, websites, databases, etc.

Password recovery and reset

Password recovery/reset

- ▶ Problem: hashed passwords cannot be recovered (hopefully)



“Hi... I forgot my password. Can you email me a copy? Kthxbye”

- This is why systems typically implement password **reset**
 - Use out-of-band info to authenticate the user
 - Overwrite `hash(old_pw)` with `hash(new_pw)`
- Be careful: it is possible for an attacker to exploit password reset

Knowledge-based reset

- ▶ Typical implementations use **Knowledge Based Authentication (KBA)**
 - ▶ What was your mother's maiden name?
 - ▶ What was your prior street address?
 - ▶ Where did you go to elementary school
- ▶ **Problems?**
 - ▶ This information is widely available to anyone
 - ▶ Publicly accessible social network profiles
 - ▶ Background-check services like Spokeo
- ▶ **Experts recommend that services not use KBA**
 - ▶ When asked, users should generate random answers to these questions

Account-based reset

- ▶ **Idea: authenticate a user by sending a code to their contact address**
 - ▶ Typically e-mail address or phone number
- ▶ **Security rests on the assumption that the person's contact address is also secure**
 - ▶ E-mail account takeover
 - ▶ SIM hijacking

Challenges of password reset

- ▶ Password reset mechanisms are often targeted and are quite vulnerable
- ▶ **Best practice: implement a layered mechanism**
 - ▶ Knowledge-based
 - ▶ Secondary account
 - ▶ Second factor authentication: biometric or tokens
- ▶ **Warning: more secure = less usable**
 - ▶ Password loss is common, people will be frustrated by onerous reset mechanisms



Password cracking

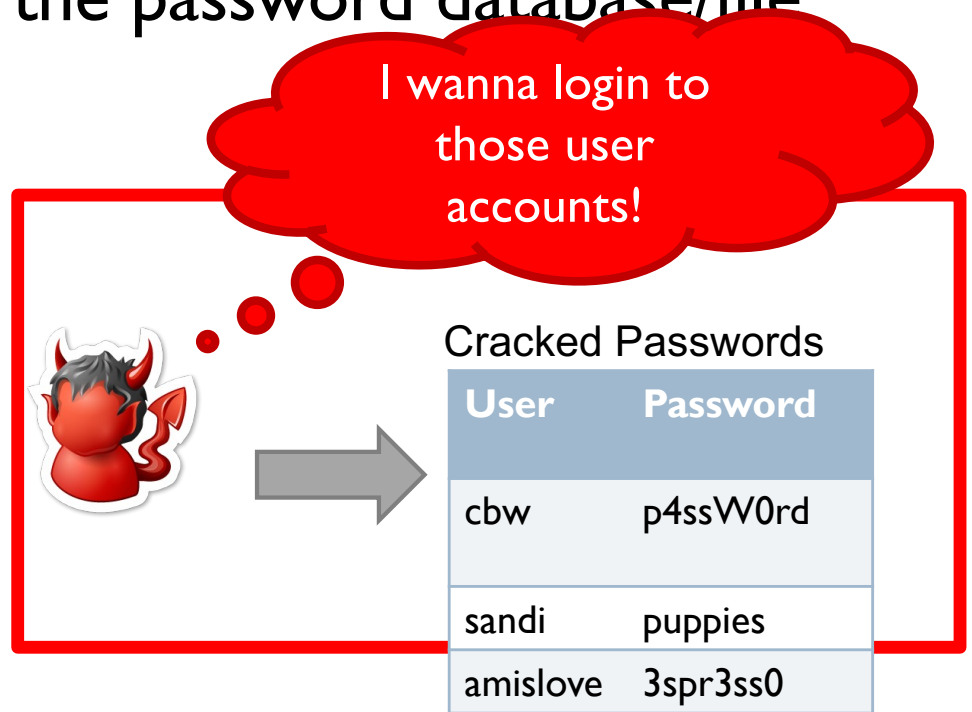
Attacker goals and threat model

- ▶ Assume we have a system storing usernames and passwords
- ▶ The attacker has access to the password database/file

Database



User	H(PW)
cbw	iuafNas
sandi	23asZR
amislove	9xgGw/

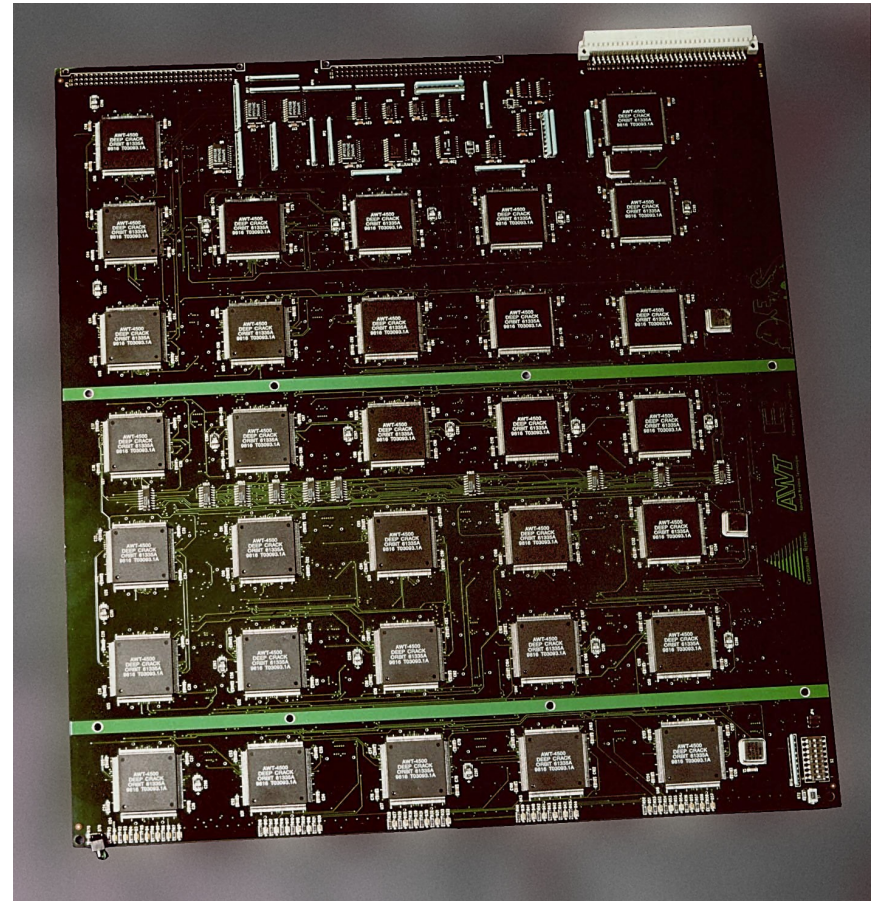


Basic password cracking

- ▶ Problem: humans are terrible at generating/remembering random strings
- ▶ Passwords are often weak enough to be brute-forced
 - ▶ Naïve way: systematically try all possible passwords
 - ▶ Slightly smarter way: take into account non-uniform distribution of characters
- ▶ Dictionary attacks are also highly effective
 - ▶ Select a baseline wordlist/dictionary full of **likely** passwords
 - ▶ Today, the best wordlists come from lists of breached passwords
 - ▶ Rule-guided word mangling to look for slight variations
 - ▶ E.g. password → Password → p4ssword → passw0rd → p4ssw0rd → passwordl → etc.
- ▶ Many password cracking tools exist (e.g. John the Ripper, hashcat)

“Deep Crack”: The EFF DES Cracker

- ▶ DES uses a 56-bit key
- ▶ \$250K in 1998, capable of brute-forcing DES keys in 56 hours
 - ▶ Uses 1856 custom ASIC chips
- ▶ Similar attacks have been demonstrated against MD5, SHA1
- ▶ Modern equivalent?
 - ▶ Bitcoin mining ASICs



Speeding up brute-force cracking

- ▶ Brute force attacks are slow because hashing is CPU intensive
 - ▶ Especially if a strong function (SHA512, bcrypt) is used
- ▶ Idea: why not pre-compute and store all hashes?
 - ▶ You would only need to pay the CPU cost once...
 - ▶ ... for a given salt
- ▶ Given a hash function H , a target hash h , and password space P , goal is to recover $p \in P$ such that $H(p) = h$
- ▶ Problem: naïve approach requires $\Theta(|P|n)$ bits, where n is the space of the output of H

Hash chains

- ▶ Hash chains enable time-space efficient reversal of hash functions
- ▶ Key idea: pre-compute **chains** of passwords of length k ...
 - ▶ ... but only store the start and end of each chain
 - ▶ Larger $k \rightarrow$ fewer chains to store, more CPU cost to rebuild chains
 - ▶ Small $k \rightarrow$ more chains to store, less CPU cost to rebuild chains
- ▶ Building chains require H , as well as a reduction $R : H \mapsto P$
 - ▶ Begin by selecting some initial set of password $P' \subset P$
 - ▶ For each $p' \in P'$, apply $H(p') = h', R(h') = p''$ for k iterations
 - ▶ Only store p' and p'^k
 - ▶ Example of R -- take only the first 8 characters of the hash output
- ▶ To recover hash h , apply R and H until the end of a chain is found
 - ▶ Rebuild the chain using p' and p'^k
 - ▶ $H(p) = h$ may be within the chain

Uncompressed hash chain example

Only these two columns get stored on disk

p	$H(p') = h'$	$R(h') = p''$	$H(p'') = h''$	$R(h'') = p'''$	$H(p''') = h'''$	$R(h''') = p^*$
abcde	\\WPNP_	vlsfqp	_QOZLR	eusrqv	CMRQ5X	cjldar
passw	VZDGEF	gfnxsk	ZLGEKV	yookol	EBOTHT	zvxscs
12345	SM-QK\9	sawtzg	RHKP_D	gvmdwm	BYE4LB	wjizbn
seprt	OKFTaY	btweoz	WA15HK	ttgovl	Q_4\6ZB	eivlac

K = 3

Hash chain example

p'	p^*
abcde	cjldar
passw	zvxscs
12345	wjizbn
seprt	eivlqc

$K = 3$

- Size of the table is dramatically reduced...
- ... but some computation is necessary once a match is found

p'	$H(p') = h'$	$R(h') = p''$	$H(p'') = h''$	$R(h'') = p'''$	$H(p''') = h'''$	$R(h''') = p^*$
12345						wjizbn

p	$H(p) = h$	$R(h) = p'$	$H(p') = h'$	$R(h') = p''$	$H(p'') = h''$	$R(h'') = p'''$	$H(p''') = h'''$
RHKP_D							

Desired password

Hash to recover

Problems with hash chains

- ▶ Hash chains are prone to collisions
 - ▶ Collisions occur when $H(p') = H(p'')$ or $R(h') = R(h'')$ (the latter is more likely)
 - ▶ Causes the chains to merge or overlap
- ▶ Problems caused by collisions
 - ▶ Wasted space in the file, since the chains cover the same password space
 - ▶ False positives: a chain may not include the password even if the end matches
- ▶ Proper choice of $R()$ is critical
 - ▶ Goal is to cover *likely* password space, not entire password space
 - ▶ R cannot be collision resistant (like H) since it has to map into likely plaintexts
- ▶ Difficult to select R under this criterion

Rainbow tables

- ▶ Rainbow tables improve on hash chains by reducing the likelihood of collisions
- ▶ Key idea: instead of using a single reduction R , use a family of reductions $\{R_1, R_2, \dots, R_k\}$
 - ▶ Usage of H is the same as for hash chains
 - ▶ A collisions can only occur between two chains if it happens at the same position (e.g. R_i in both chains)

Rainbow tables

- ▶ **Caveats**
 - ▶ Tables must be built for each hash function and character set
 - ▶ Salting and key stretching defeat rainbow tables
- ▶ **Rainbow tables are effective in some cases, e.g. MD5 and NTLM (Microsoft authentication)**
 - ▶ Precomputed tables can be bought or downloaded for free



Choosing passwords

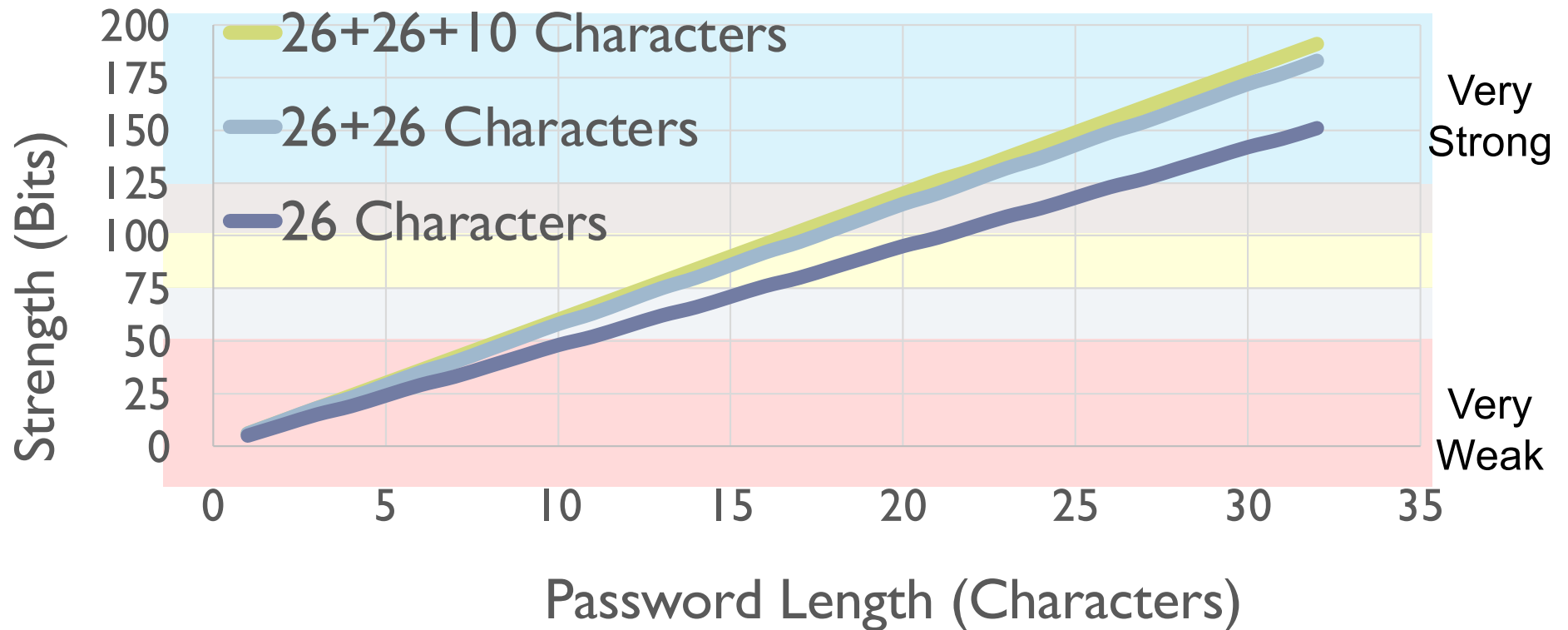
Password quality

$$S = \log_2 N^L$$

- ▶ How do we measure password quality? Entropy (captures unpredictability)
 - ▶ N – the number of possible symbols (e.g. lowercase, uppercase, numbers, etc.)
 - ▶ L – the length of the password
 - ▶ S – the strength of the password, in bits
- ▶ Formula tells you length L needed to achieve a desired strength S ...
 - ▶ ... for **randomly generated** passwords
- ▶ Is this a realistic measure in practice?

Strength of random passwords

$$S = L * \log_2 N$$



Mental algorithms

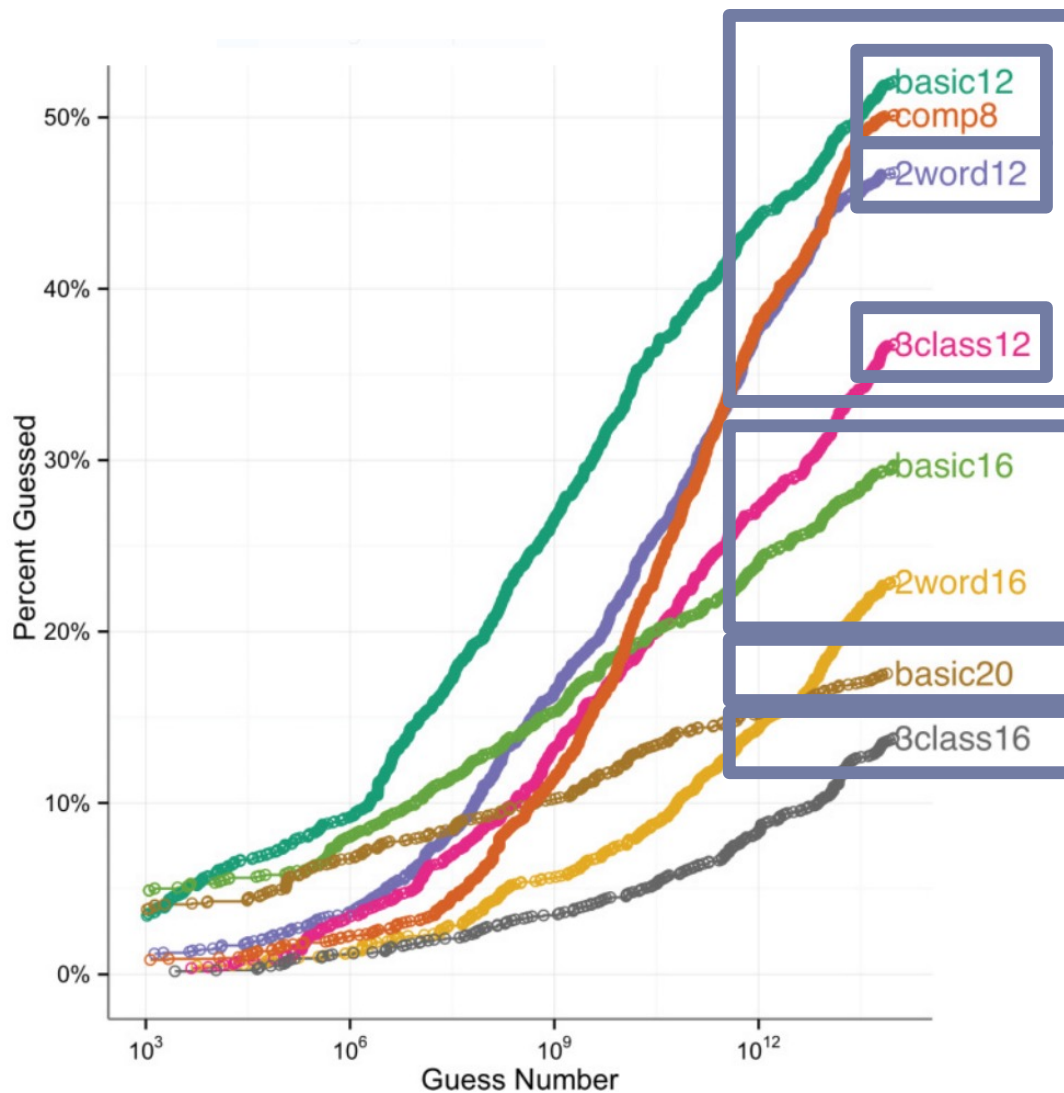
- ▶ Years of security advice have trained people to generate “secure” passwords
 1. Pick a word
 2. Capitalize the first or last letter
 3. Add a number (and maybe a symbol) to the beginning or end

- 1. Pick a word
- 2. Replace some of the letters with symbols (a → @, s → \$, etc.)
- 3. Maybe capitalize the first or last letter

Human generated passwords

Password
Computer3@
cl4ssr00m
7Dogsled*
TjwI989&6
B4nk0f4m3rIc4!

- Modern attackers are sophisticated
 - No need for brute force cracking!
 - Use dictionaries containing common words and passwords from prior leaks
 - Apply common “mental” permutations



Password Requirements

- $comp\ n$ and $basic\ n$: use at least n characters
- $k\ word\ n$: combine at least k words using at least n characters
- $d\ class\ n$: use at least d character types (upper, lower, digit, symbol) with at least n characters

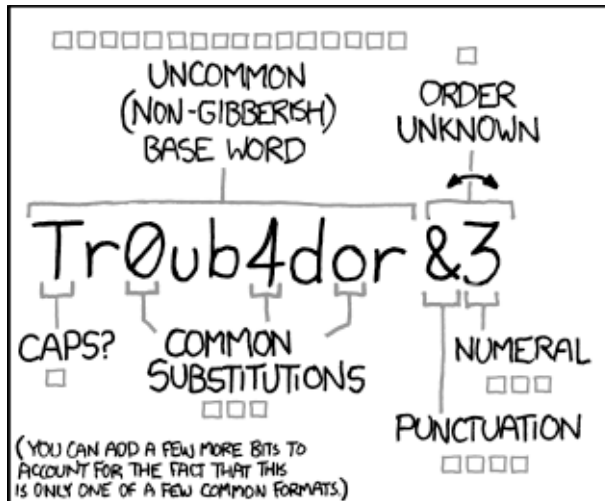
Plot from Shay et al.
https://www.blaseur.com/papers/tissec_1026.pdf

Better heuristics

- ▶ Notice that in $S = L * \log_2 N$, length matters more than symbol types
 - ▶ Choose longer passwords (16+ characters)
 - ▶ Do not worry about uppercase, digits, or symbols
- ▶ Use mnemonics
 - ▶ Choose a sentence or phrase
 - ▶ Reduce it to the first letter of each word
 - ▶ Insert random uppercase, digits, and symbols

I double dare you, say “what” one more time
i2Dy,s”w”omt





~ 28 BITS OF ENTROPY

$2^{28} = 3 \text{ DAYS AT } 1000 \text{ GUESSES/SEC}$

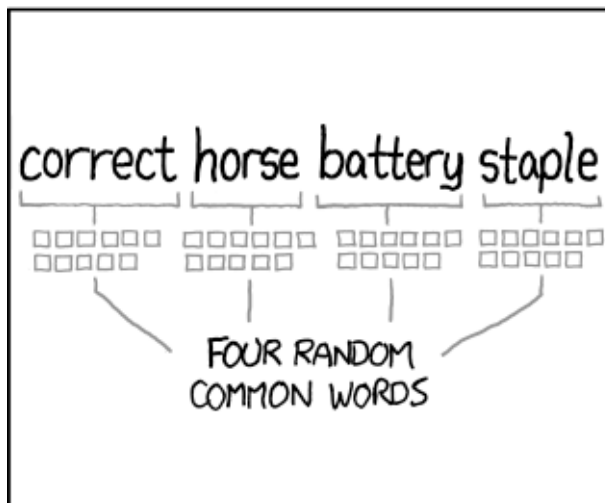
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)

DIFFICULTY TO GUESS: **EASY**

WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE 0s WAS A ZERO?

AND THERE WAS SOME SYMBOL...

DIFFICULTY TO REMEMBER: **HARD**



~ 44 BITS OF ENTROPY

$2^{44} = 550 \text{ YEARS AT } 1000 \text{ GUESSES/SEC}$

DIFFICULTY TO GUESS: **HARD**

THAT'S A BATTERY STAPLE.

CORRECT!

DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT

THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Password reuse

- ▶ People have difficulty remembering >4 passwords
 - ▶ Thus, people tend to reuse passwords across services
 - ▶ What happens if any one of these services is compromised?
- ▶ Service-specific passwords are a beneficial form of compartmentalization
 - ▶ Limits the damage when one service is inevitably breached
- ▶ Use a password manager
- ▶ Some service providers now check for password reuse
 - ▶ Forbid users from selecting passwords that have appeared in leaks

Sites

Favorites (8)

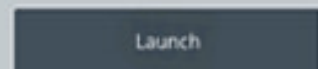
Sort By: Folder (a-z)



AirBnB
fan@lastpass.com



Amazon
fan@lastpass.com



Best Buy
fan@lastpass.com



Dropbox
fan@lastpass.com



Evernote
fan@lastpass.com



Facebook
fan@lastpass.com



Pocket
fan@lastpass.com



Twitter
fan@lastpass.com

Banking and Finance (3)

Read Only • Shared Folder



Bank of America
fan@lastpass.com



Fidelity
fan@lastpass.com



Mint
fan@lastpass.com



Have I Been Pwned (Troy Hunt) [AU] | https://haveibeenpwned.com

Home Notify me Domain search Who's been pwned Passwords API About Donate

';--have i been pwned?

Check if you have an account that has been compromised in a data breach

email address or username

264	4,859,717,682	61,081	59,268,789
pwned websites	pwned accounts	pastes	paste accounts

Biometric two-factor authentication

Types of Secrets

- ▶ Actors provide their secret to **log-in** to a system
- ▶ Three classes of secrets:
 1. Something you know
 - ▶ Example: a password
 2. Something you have
 - ▶ Examples: a smart card or smart phone
 3. Something you are
 - ▶ Examples: fingerprint, voice scan, iris scan

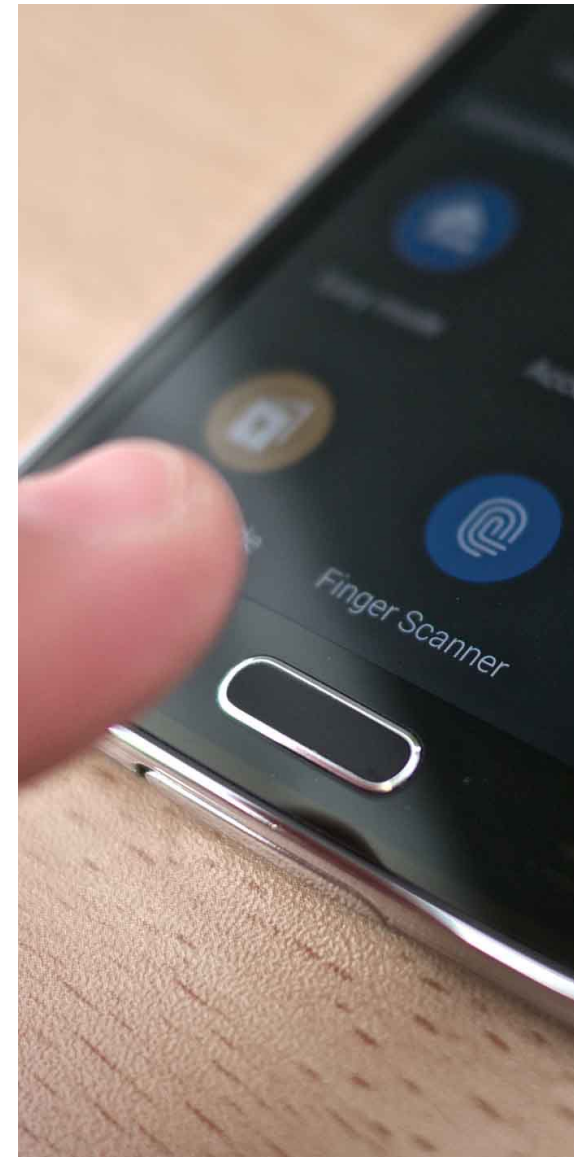
Biometrics

- ▶ ancient Greek: bios = "life", metron = "measure"
- ▶ **Physical features**
 - ▶ Fingerprints
 - ▶ Face recognition
 - ▶ Retinal and iris scans
 - ▶ Hand geometry
- ▶ **Behavioral characteristics**
 - ▶ Handwriting recognition
 - ▶ Voice recognition
 - ▶ Typing cadence
 - ▶ Gait

Fingerprints

- ▶ Ubiquitous on modern smartphones, some laptops
- ▶ Secure?
 - ▶ May be subpoenaed by law enforcement
 - ▶ Relatively easy to compromise
 1. Pick up a latent fingerprint (e.g. off a glass) using tape or glue
 2. Photograph and enhance the fingerprint
 3. Etch the print into gelatin backed by a conductor
 4. Profit ;)

https://www.theregister.co.uk/2002/05/16/gummi_bears_defeat_fingerprint_sensors/



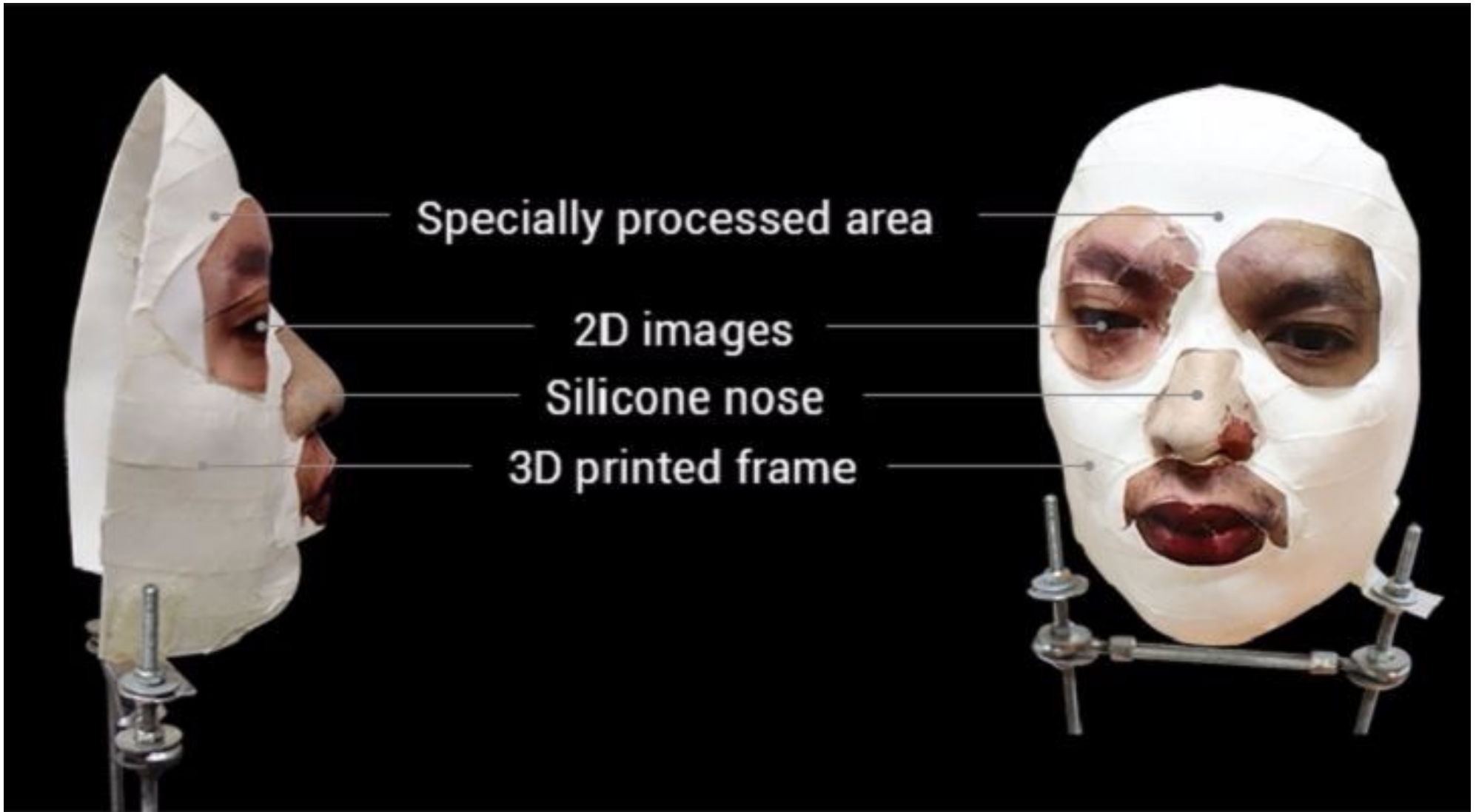
Authentication

Facial recognition

- ▶ Popularized by FaceID on the iPhone X
- ▶ Secure?
 - ▶ It depends
- ▶ Vulnerable to law enforcement requests
- ▶ Using 2D images?
 - ▶ Not secure
 - ▶ Trivial to break with a photo of the target's face
- ▶ Using 2D images + 3D depth maps?
 - ▶ More secure, but not perfect
 - ▶ Can be broken by crafting a lifelike mask of the target

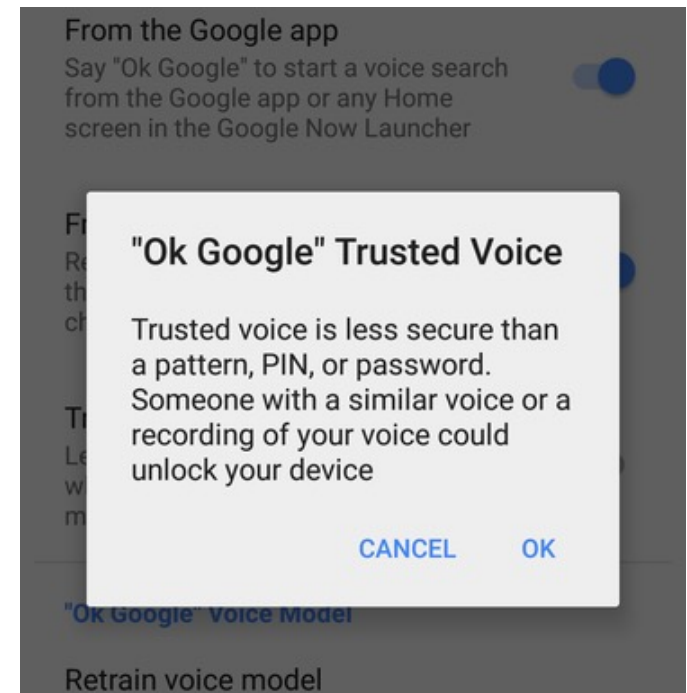


Authentication



Voice recognition

- ▶ **Secure?**
 - ▶ Very much depends on the implementation
- ▶ **Some systems ask you to record a static phrase**
 - ▶ E.g. say “unlock” to unlock
 - ▶ This is wildly insecure
 - ▶ Attacker can record and replay your voice
- ▶ **Others ask you to train a model of your voice**
 - ▶ Train the system by speaking several sentences
 - ▶ To authenticate, speak several randomly chosen words
 - ▶ Not vulnerable to trivial replay attacks, but still vulnerable
 - ▶ Given enough samples of your voice, an attacker can train a synthetic voice AI that sounds just like you



Fundamental issue with biometrics

- ▶ **Biometrics are immutable**
 - ▶ You are the password, and you cannot change
 - ▶ Unless you plan on undergoing plastic surgery?
- ▶ **Once compromised, there is no reset**
 - ▶ Passwords and tokens can be changed
- ▶ **Example: the Office of Personnel Management (OPM) breach**
 - ▶ US gov agency responsible for background checks
 - ▶ Had fingerprint records of all people with security clearance
 - ▶ Breached by China in 2015, all records stolen :(

Token based two-factor authentication

Types of Secrets

- ▶ Actors provide their secret to **log-in** to a system
- ▶ Three classes of secrets:
 1. Something you know
 - ▶ Example: a password
 2. Something you have
 - ▶ Examples: a smart card or smart phone
 3. Something you are
 - ▶ Examples: fingerprint, voice scan, iris scan

Something you have

- ▶ Two-factor authentication has become more commonplace
- ▶ Possible second factors:
 - ▶ SMS passcodes
 - ▶ Time-based one time passwords
 - ▶ Hardware tokens

SMS two factor


- ▶ Relies on your phone number as the second factor
 - ▶ Key assumption: only your phone should receive SMS sent to your number
- ▶ SMS two factor is deprecated. Why?
- ▶ Social engineering the phone company
 1. Call and pretend to be the victim
 2. Say “I got a new SIM, please activate it”
 3. If successful, phone calls and SMS are now sent to your SIM in your phone, instead of the victim
- ▶ Not hypothetical: successfully used against many victims



Authentication

One time passwords

- ▶ Generate ephemeral passcodes that change over time
- ▶ To login, supply normal password and the current one time password
- ▶ Relies on a shared secret between your mobile device and the service provider
 - ▶ Shared secret allows both parties to know the current one time password



Changes every few minutes

ACME INC

AUTHY

ACME INC TOKEN IS:

6883932

Duo Mobile

Lastpass Authenticator

Google Authenticator

The image shows a mobile application interface for 'Authy'. At the top, there is a red header bar with 'ACME INC' on the left and a menu icon on the right. Below the header, the 'Authy' logo is displayed. Underneath the logo, it says 'ACME INC TOKEN IS:' followed by a large blue number '6883932'. To the right of the number is a red circular icon with a white document symbol. A blue speech bubble points to the number with the text 'Changes every few minutes'. Below the main interface, there are three icons representing other authentication apps: Duo Mobile (green), Lastpass Authenticator (red), and Google Authenticator (grey).

Time-based one-time password algorithm

$T0$ = <the beginning of time, typically Thursday, 1 January 1970 UTC>

Tl = <length of time the password should be valid>

K = <shared secret key>

d = <the desired number of digits in the password>

$TC = \text{floor}((\text{unixtime}(\text{now}) - \text{unixtime}(T0)) / Tl),$

$\text{TOTP} = \text{HMAC}(K, TC) \% 10^d$

Specially formatted
SHA1-based signature


Given K , this algorithm can
be run on your phone and by
the service provider

Secret sharing for TOTP

Enable Two-Step Sign in

An authenticator app generates the code automatically on your smartphone. Free apps are available for all smartphone platforms including iOS, Android, Blackberry and Windows. Look for an app that supports time-based one-time passwords (TOTP) such as Google Authenticator or Duo Mobile.

To set up your mobile app, add a new service and scan the QR code.



If you can't scan the code, enter this secret key manually: fvxo [blurred]

[USE SMS INSTEAD](#) [CANCEL](#) [NEXT STEP](#)

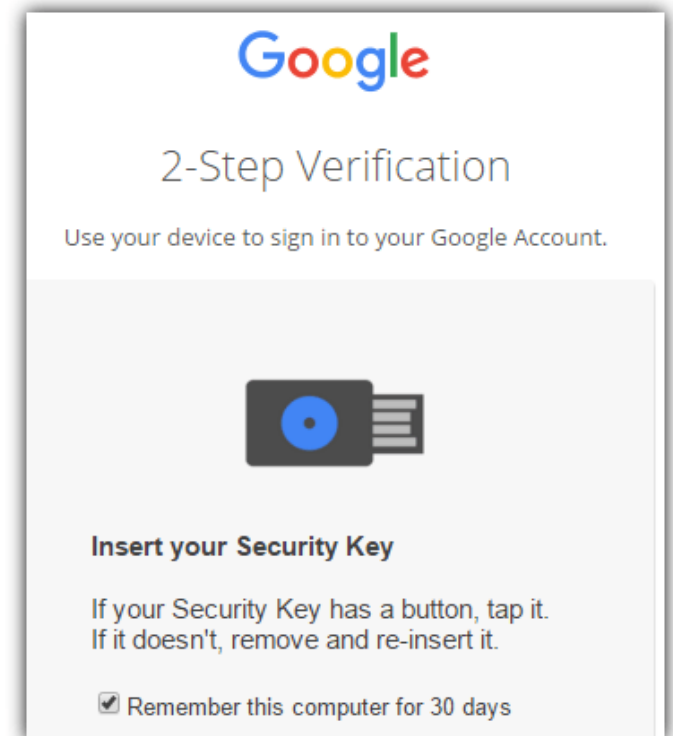
Hardware two factor

- ▶ Special hardware designed to hold cryptographic keys
- ▶ Physically resistant to key extraction attacks
 - ▶ E.g. scanning tunneling electron microscopes
- ▶ Uses:
 - ▶ 2nd factor for OS log-on
 - ▶ 2nd factor for some online services
 - ▶ 2nd factor for password manager
 - ▶ Storage of PGP and SSH keys



Universal 2nd Factor (U2F)

- ▶ Supported by Chrome, Opera, and Firefox
- ▶ Works with Google, Dropbox, Facebook, Github, Gitlab, etc.
- ▶ Pro tip: always buy 2 security keys
 - ▶ Associate both with your accounts
 - ▶ Keep one locked in a safe, in case you lose your primary key ;)





Authentication in Linux

Status check

- ▶ **At this point, we have discussed:**
 - ▶ How to securely store passwords
 - ▶ Techniques used by attackers to crack passwords
 - ▶ Biometrics and 2nd factors
- ▶ **Next topic: building authentication systems**
 - ▶ Given a user and password, how does the system authenticate the user?
 - ▶ How can we perform efficient, secure authentication in a distributed system?

Authentication in Unix/Linux

- ▶ Users authenticate with the system by interacting with *login*
 - ▶ Prompts for username and password
 - ▶ Credentials checked against locally stored credentials
- ▶ By default, password policies specified in a centralized, modular way
 - ▶ On Linux, using Pluggable Authentication Modules (PAM)
 - ▶ Authorizes users, as well as environment, shell, prints MOTD, etc.

Example PAM Configuration

```
# cat /etc/pam.d/system-auth
#%PAM-1.0
```

```
auth required pam_unix.so try_first_pass nullok
auth optional pam_permit.so
auth required pam_env.so
```

```
account required pam_unix.so
account optional pam_permit.so
account required pam_time.so
```

```
password required pam_unix.so try_first_pass
password optional pam_permit.so
```

```
session required pam_limits.so
session required pam_unix.so
session optional pam_permit.so
```

- Use SHA512 as the hash function
- Use /etc/shadow for storage

Unix Passwords

- ▶ **Traditional method: *crypt***
 - ▶ First eight bytes of password used as key (additional bytes are ignored)
 - ▶ 12-bit salt
 - ▶ 25 iterations of DES on a zeroed vector
- ▶ **Modern version of *crypt* are more extensible**
 - ▶ Full password used
 - ▶ Up to 16 bytes of salt
 - ▶ Support for additional hash functions like MD5, SHA256, and SHA512
 - ▶ Key lengthening: defaults to 5000 iterations, up to $10^8 - 1$



Password files

- ▶ Password hashes used to be in */etc/passwd*
 - ▶ World readable, contained usernames, password hashes, config information
 - ▶ Many programs read config info from the file...
 - ▶ But very few (only one?) need the password hashes
- ▶ Turns out, world-readable hashes are **Bad Idea**
- ▶ Hashes now located in */etc/shadow*
 - ▶ Also includes account metadata like expiration
 - ▶ Only visible to root



Password storage on Linux

`/etc/passwd`

`username:x:UID:GID:full_name:home_directory:shell`

`cbw:x:1001:1000:Christo Wilson:/home/cbw/~/bin/bash`

`amislove:x:1002:1000:amislove:/home/amislove/~/bin/sh`

`$<algo>$<salt>$<hash>`

Algo: 1 = MD5, 5 = SHA256, 6 =
SHA512

`/etc/shadow`

`username:password:last:may:must:warn:expire:disable:reserved`

`cbw:$1$0nSd5ewF$0df/3G7iSV49nsbAa/5gSg:9479:0:10000:::`

`amislove:1I3RxU5F1$:8172:0:10000:::`



Distributed authentication

Distributed authentication

- ▶ Early on, people recognized the need for authentication in distributed environments
 - ▶ Example: university lab with many workstations
 - ▶ Example: file server that accepts remote connections
- ▶ Synchronizing and managing password files on each machine is not scalable
 - ▶ Ideally, you want a centralized repository that stores policy and credentials

The Yellow Pages

- ▶ Network Information Service (NIS), a.k.a. the Yellow Pages
 - ▶ Developed by Sun to distribute network configurations
 - ▶ Central directory for users, hostnames, email aliases, etc.
 - ▶ Exposed through *yp** family of command line tools
- ▶ For instance, depending on */etc/nsswitch.conf*, hostname lookups can be resolved by using
 - ▶ */etc/hosts*
 - ▶ DNS
 - ▶ NIS
- ▶ Superseded by NIS+, LDAP

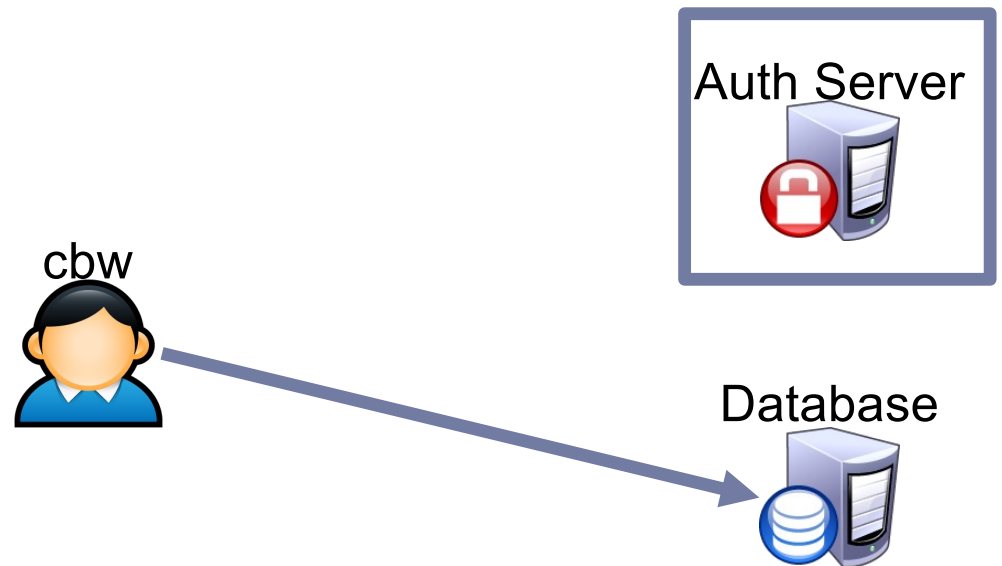
NIS Password Hashes

- *Crypt* based password hashes
- Can easily be cracked
- Many networks still rely on insecure NIS

```
[cbw@workstation ~] ypcat passwd
afbjune:qSAH.evuYFHAM:14532:65104::/home/afbjune:/bin/bash
philowe:T.yUMej3XSNAM:13503:65104::/home/philowe:/bin/bash
bratus:2omkwsYXWiLDo:6312:65117::/home/bratus:/bin/tcsh
adkap:ZfHdSwSz9WhKU:9034:65118::/home/adkap:/bin/zsh
amitpoon:i3LjTqgU9gYSc:8198:65117::/home/amitpoon:/bin/tcsh
kcole:sgYtUs0tyk38k:14192:65104::/home/kcole:/bin/bash
david87:vA06wxjJEUgBE:13055:65101::/home/david87:/bin/bash
loch:6HgIQrVkcBeiw:13729:65104::/home/loch:/bin/bash
ppkk315:s6CTSAkqqr/nU:14061:65101::/home/ppkk315:/bin/bash
haynesma:JYWaQUARSqDQE:14287:65105::/home/haynesma:/bin/bash
ckubicek:jYpwYhqqr3tA:10937:65117::/home/ckubicek:/bin/tcsh
mwalz:wPIa5Bv/tFVb2:9103:65118::/home/mwalz:/bin/tcsh
sushma:G6XNe18GpeQj.:13682:65104::/home/sushma:/bin/bash
guerin1:n0Da2Tm09MDBI:14512:65105::/home/guerin1:/bin/bash
```

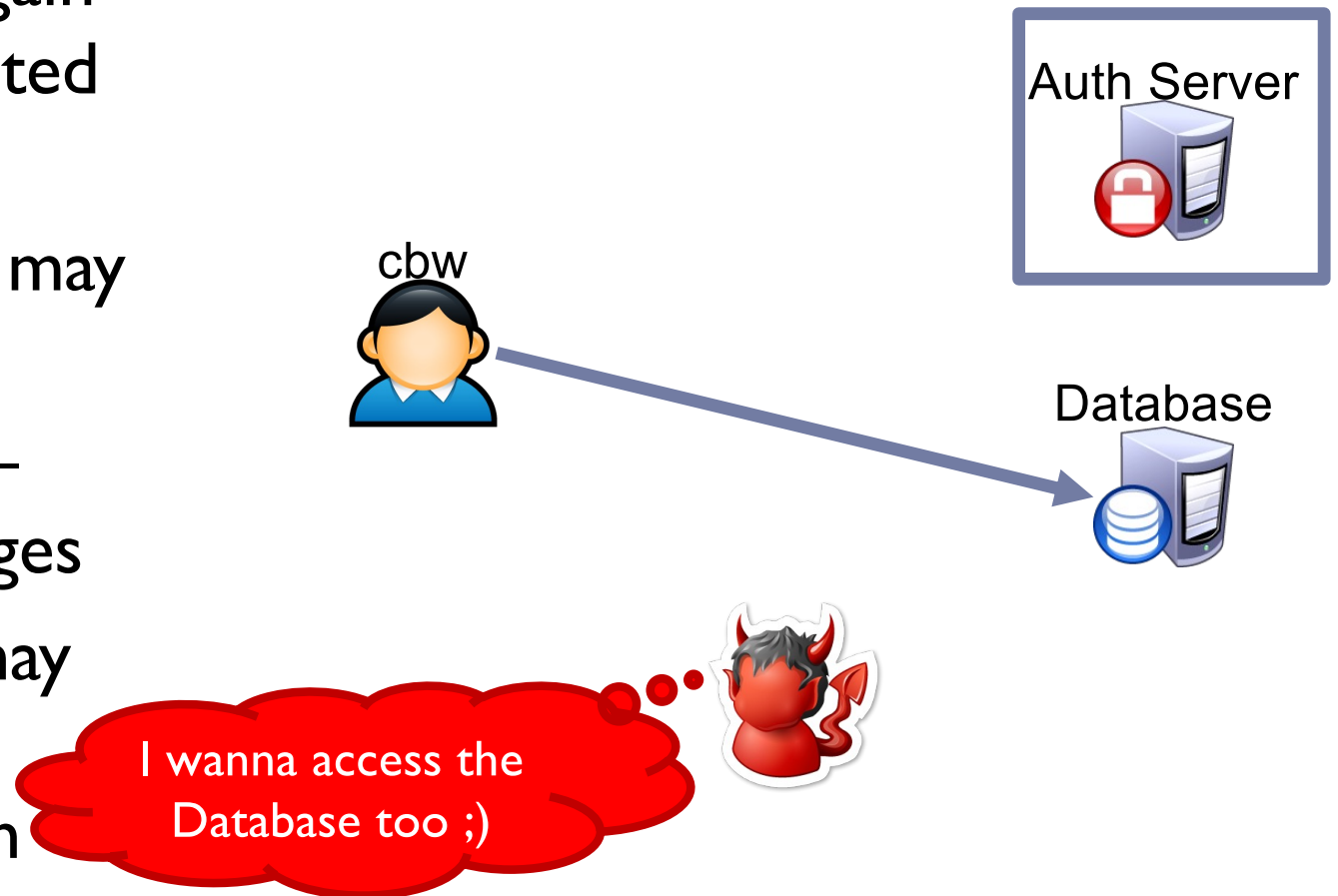
Distributed authentication revisited

- ▶ **Goal:** a user would like to use some resource on the network
 - ▶ File server, printer, database, mail server, etc.
- ▶ **Problem:** access to resources requires authentication
 - ▶ Auth Server contains all credential information
 - ▶ You do not want to replicate the credentials on all services



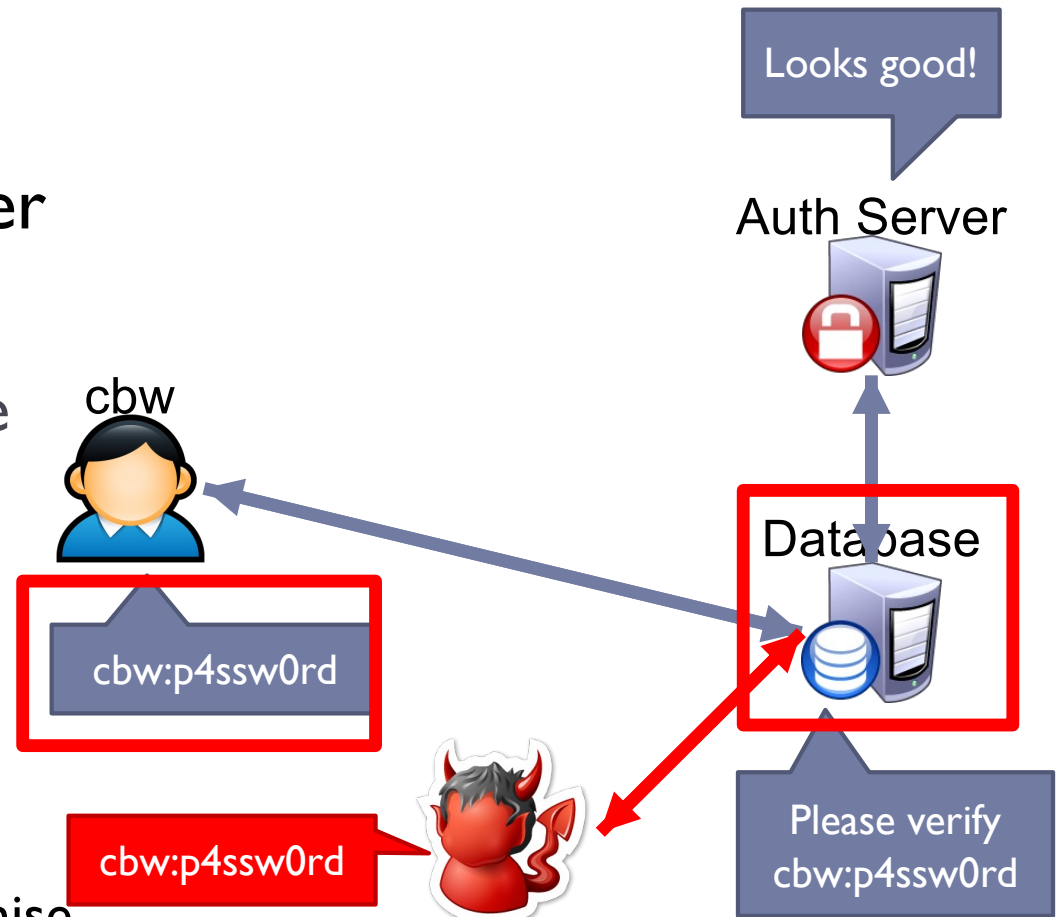
Attacker goals and threat model

- ▶ Goal: steal credentials and gain access to protected resources
- ▶ Local attacker – may spy on traffic
- ▶ Active attacker – may send messages
- ▶ In some cases, may be able to steal information from users



(Bad) Distributed Auth Example

- ▶ Idea: client forwards user/password to service, service queries Auth Server
- ▶ Problems:
 - ▶ Passwords being sent in the clear
 - ▶ Attacker can observe them!
 - ▶ Clearly we need encryption
 - ▶ Database learns about passwords
 - ▶ Additional point of compromise
 - ▶ Ideally, only the user and the Auth Server should know their password



Reflection attack example

- ▶ A and B authenticate to each other, authentication comes from knowing the secret key K , r_A , r_B are random numbers (nonces)

(1) $A \rightarrow B : r_A$

(2) $B \rightarrow A : E_k(r_A, r_B)$

(3) $A \rightarrow B : r_B$

- ▶ **Attack:** E wants to trick A to accept them as B, thus needs to obtain something like $E_k(r_A, r_x)$, where r_A was selected by A, and r_x can be selected by E, without knowing secret key K

(1) $A \rightarrow E : r_A$

(2) $E \rightarrow A : r_A$: Starting a new session, note that here E is the initiator

(3) $A \rightarrow E : E_k(r_A, r_{A'})$: Reply of (2)

(4) $E \rightarrow A : E_k(r_A, r_{A'})$: Reply of (1)

(5) $A \rightarrow E : r_{A'}$; this concludes session started with (1)

Interleaving attack example

- ▶ A and B authenticate to each other based on their secret keys used for digital signature

(1) $A \rightarrow B : r_A$

(2) $B \rightarrow A : r_B, S_B(r_B, r_A, A)$

(3) $A \rightarrow B : r_{A'}, S_A(r_{A'}, r_B, B)$

- ▶ Attack: E wants to pass as A to B without knowing A's private key, so E needs to get A to sign a message

(1) $E \rightarrow B : r_A$

(2) $B \rightarrow E : r_B, S_B(r_B, r_A, A)$ E can not finish now, needs A to sign something

(1) $E \rightarrow A : r_B$ E starts a parallel session with A

(2) $A \rightarrow E : r_{A'}, S_A(r_{A'}, r_B, B)$ this is what E needs for the session with B

(3) $E \rightarrow B : r_{A'}, S_A(r_{A'}, r_B, B)$ E can complete the session with B

Needham-Schroeder Protocol

- Let Alice A and Bob B be two parties that trust server S
- K_{AS} and K_{BS} are shared secrets between $[A, S]$ and $[B, S]$
- K_{AB} is a negotiated session key between $[A, B]$
- N_i and N_j are random **nonces** generated by A and B

1) $A \rightarrow S: A, B, N_i$

K_{AS} is not sent in the clear, authenticates S and A

2) $S \rightarrow A: \{N_i, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$

3) $A \rightarrow B: \{K_{AB}, A\}_{K_{BS}}$

K_{BS} is not sent in the clear, authenticates B

4) $B \rightarrow A: \{N_j\}_{K_{AB}}$

Challenge nonce forces A to acknowledge they have K_{AB}

5) $A \rightarrow B: \{N_j - 1\}_{K_{AB}}$

Needham-Schroeder Example

1) $A \rightarrow S: A, B, N_i$

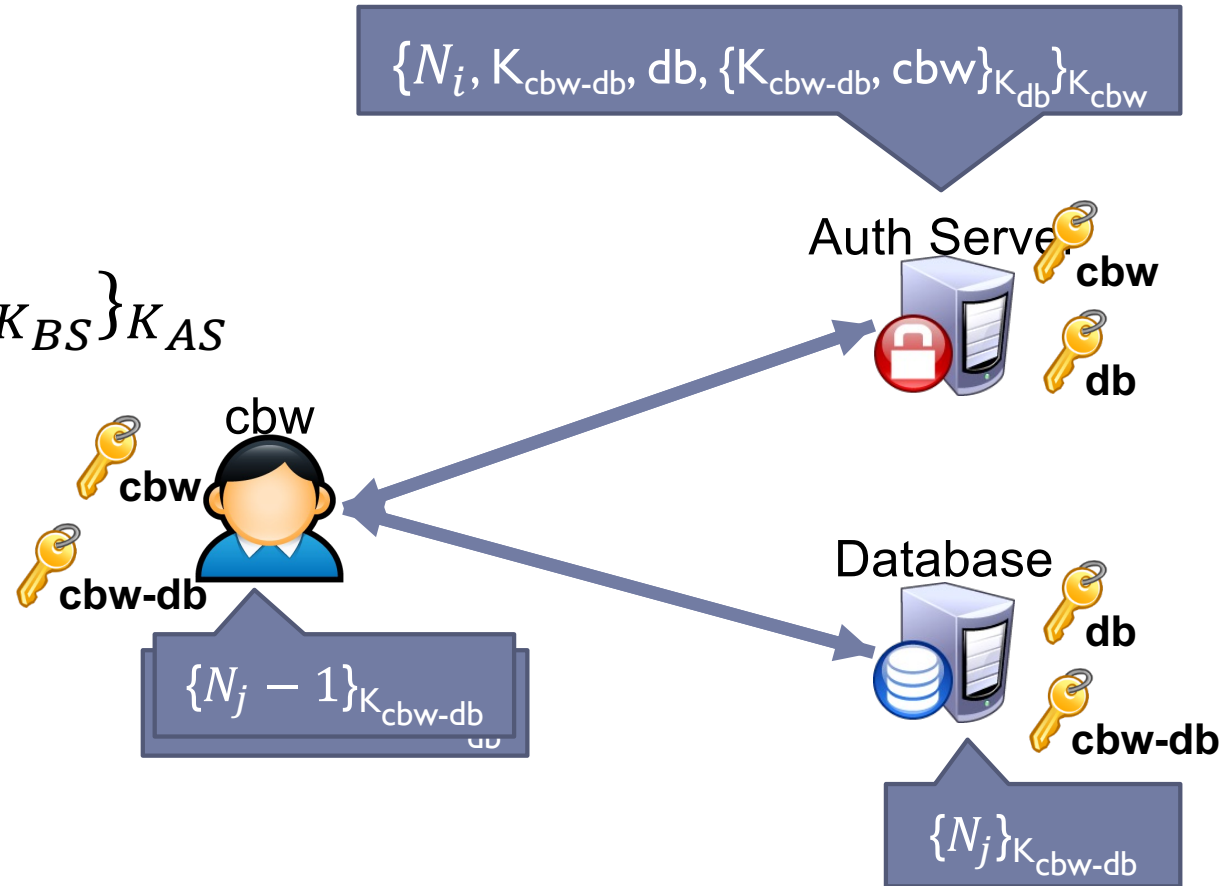
2) $S \rightarrow$

$A: \{N_i, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$

3) $A \rightarrow B: \{K_{AB}, A\}_{K_{BS}}$

4) $B \rightarrow A: \{N_j\}_{K_{AB}}$

5) $A \rightarrow B: \{N_j - 1\}_{K_{AB}}$



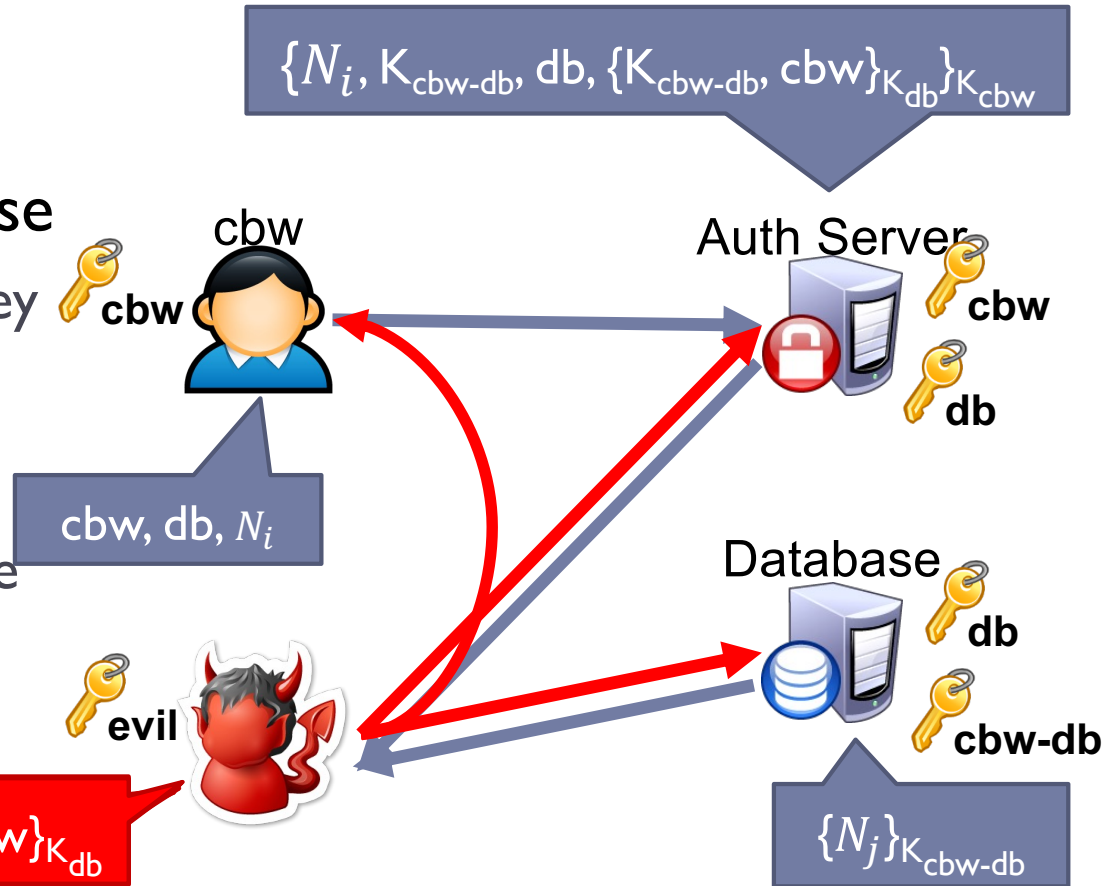
Attacking Needham-Schroeder

- ▶ Spoof the client request
 - ▶ Fail! Client key is needed to decrypt

- ▶ Spoof the Auth Server response
 - ▶ Fail! Need to know the client key

- ▶ Spoof the client-server interaction
 - ▶ Fail! Need to know the database key

- ▶ Replay the client-server interaction
 - ▶ Fail! Need to know the session key



Replay Attack

Typical, Benign Protocol

- 1) $A \rightarrow S: A, B, N_i$
- 2) $S \rightarrow A: \{N_i, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
- 3) $A \rightarrow B: \{K_{AB}, A\}_{K_{BS}}$
- 4) $B \rightarrow A: \{N_j\}_{K_{AB}}$
- 5) $A \rightarrow B: \{N_j - 1\}_{K_{AB}}$

Replay Attack

- 1) $M \rightarrow B: \{K_{AB}, A\}_{K_{BS}}$
- 2) $B \rightarrow M: \{N_j\}_{K_{AB}}$
- 3) $M \rightarrow B: \{N_j - 1\}_{K_{AB}}$

- Attacker must hack A to steal K_{AB}
 - So the attacker can also steal K_{AS}
- However, what happens after A changes K_{AS}
 - Attacker can still conduct the replay attack! Only K_{AB} is necessary!

Fixed Needham-Schroeder Protocol

- Let Alice A and Bob B be two parties that trust server S
- K_{AS} and K_{BS} are shared secrets between $[A, S]$ and $[B, S]$
- K_{AB} is a negotiated session key between $[A, B]$
- N_i and N_j are random nonces generated by A and B
- T is a timestamp chosen by S

1) $A \rightarrow S: A, B, N_i$

2) $S \rightarrow A: \{N_i, K_{AB}, B, \{K_{AB}, A, T\}_{K_{BS}}\}_{K_{AS}}$

3) $A \rightarrow B: \{K_{AB}, A, T\}_{K_{BS}}$

*B only accepts requests
with fresh timestamps*

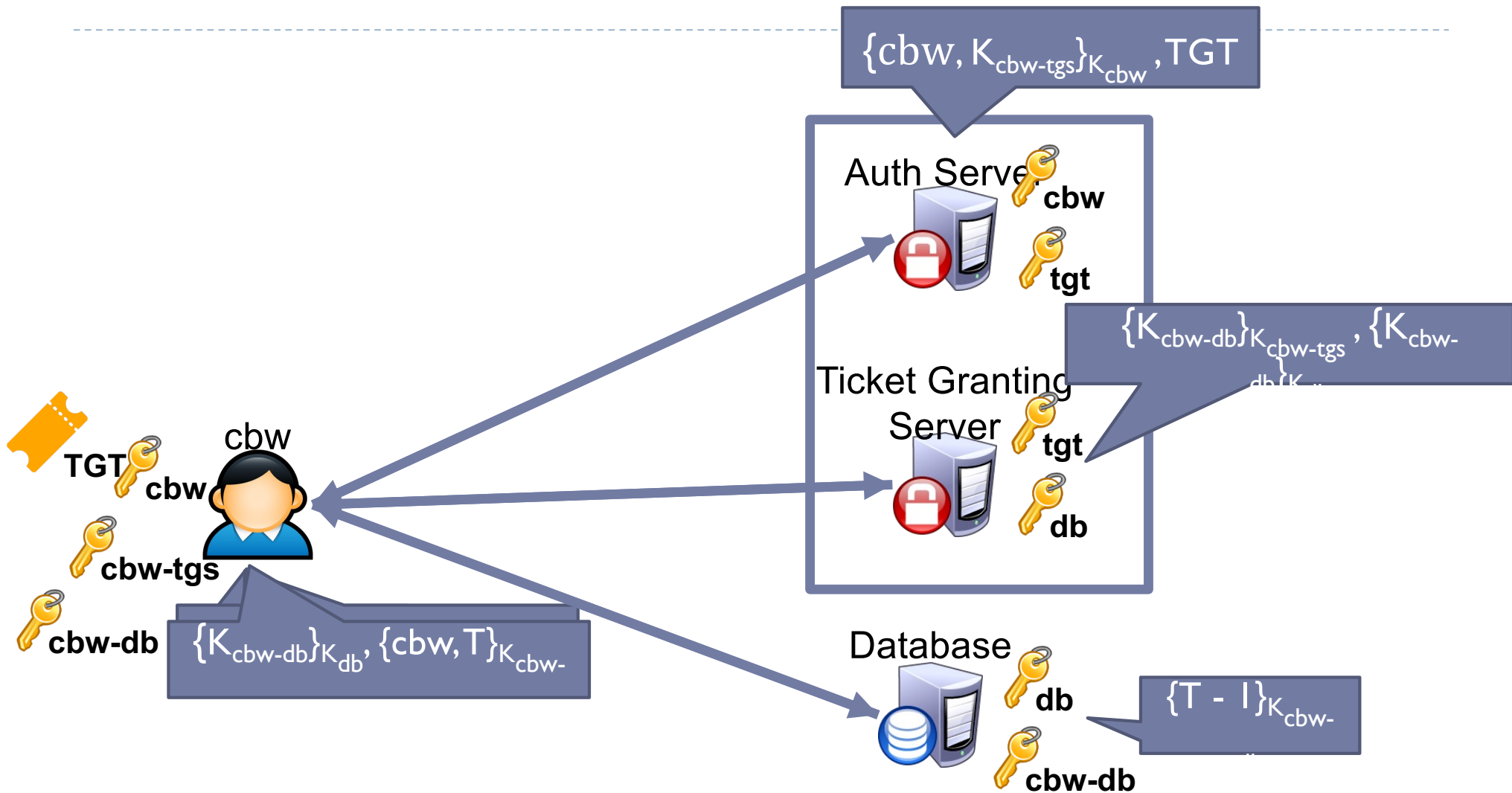
4) $B \rightarrow A: \{N_j\}_{K_{AB}}$

5) $A \rightarrow B: \{N_j - 1\}_{K_{AB}}$

Kerberos

- ▶ Created as part of MIT Project Athena
 - ▶ Based on Needham-Schroeder
- ▶ Provides mutual authentication over untrusted networks
 - ▶ Tickets as assertions of authenticity, authorization
 - ▶ Forms basis of Active Directory authentication
- ▶ Principals
 - ▶ Client
 - ▶ Server
 - ▶ Key distribution center (KDC)
 - ▶ Authentication server (AS)
 - ▶ Ticket granting server (TGS)

Kerberos Example



Attacking Kerberos

- ▶ **Don't put all your eggs in one basket**
 - ▶ The Kerberos Key Distribution Server (KDS) is a central point of failure
 - ▶ DoS the KDS and the network ceases to function
 - ▶ Compromise the KDS leads to network-wide compromise
- ▶ **Time synchronization**
 - ▶ Inaccurate clocks lead to protocol failures (due to timestamps)
 - ▶ Solution?
 - ▶ Use NTP ;)

Sources

1. Many slides courtesy of Wil Robertson: <https://wkr.io>
 2. Honeywords, Ari Juels and Ron Rivest: <http://www.arijuels.com/wp-content/uploads/2013/09/JR13.pdf>
- ▶ For more on generating secure passwords, and understanding people's mental models of passwords, see the excellent work of Blas Ur: <http://www.blaseur.com/pubs.htm>