

CS 253: Web Security

Authentication

Admin

- **Today:** Assignment 3 due Friday at 11:59pm
- **Thursday:** Guest Lecture on WebAuthn by Lucas Garron from GitHub security

**How can we build systems which are
secure even when **the attacker has
the user's password?****

What is authentication?

- **Idea:** Verify the user is who they say they are
- Authentication systems classically use three **factors**:
 - Something you know (e.g. a password)
 - Something you have (e.g. a phone, badge, or cryptographic key)
 - Something you are (e.g. a fingerprint or other biometric data)
- The more factors used, the more sure we are that the user is who they say they are



Authentication vs. Authorization

- **Authentication:** Verify the user is who they say they are
 - Login form
 - Ambient authority (e.g. HTTP cookies)
 - HTTP authentication
- **Authorization:** Decide if a user has permission to access a resource
 - Access control lists (ACLs)
 - Capability URLs

NIST Digital Identity Guidelines

Authentication establishes confidence that the claimant has possession of an authenticator(s) bound to the credential, and in some cases in the attribute values of the subscriber (e.g., if the subscriber is a U.S. citizen, is a student at a particular university, or is assigned a particular number or code by an agency or organization).

Authentication does not determine the claimant's authorizations or access privileges; this is a separate decision, and is out of these guidelines' scope. RPs can use a subscriber's authenticated identity and attributes with other factors to make authorization decisions...

Common implementation mistakes

- Usernames should be stored case insensitively
 - **feross** is the same user as **Feross**
- Usernames should be unique
 - Two users should not share the same username

Users choose weak passwords

2011	2012	2013	2014	2015
password	password	123456	123456	123456
123456	123456	password	password	password
12345678	12345678	12345678	12345	12345678
qwerty	abc123	qwerty	12345678	qwerty
abc123	qwerty	abc123	qwerty	12345
monkey	monkey	123456789	123456789	123456789
1234567	letmein	11111	1234	football
letmein	dragon	1234567	baseball	1234
trustno1	111111	iloveyou	dragon	1234567
dragon	baseball	adobe123	football	baseball

Source: SplashData

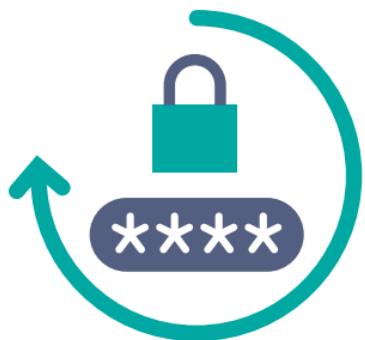
Designing password requirements

- Left on their own, users will choose weak passwords
- **Solution:** Let's enforce password requirements
 - What should the requirements be?

Payment Card Industry Data Security Standard (PCI DSS)

PASSWORD BEST PRACTICES

To minimize the risk of being breached, businesses should change vendor default passwords to strong ones, and never share them - each employee should have its own login ID and password.



Change your passwords regularly

Treat your passwords like a toothbrush. Don't let anyone else use them and get new ones every three months.



Don't share passwords

Insist on each employee having its own login ID and password - never share!

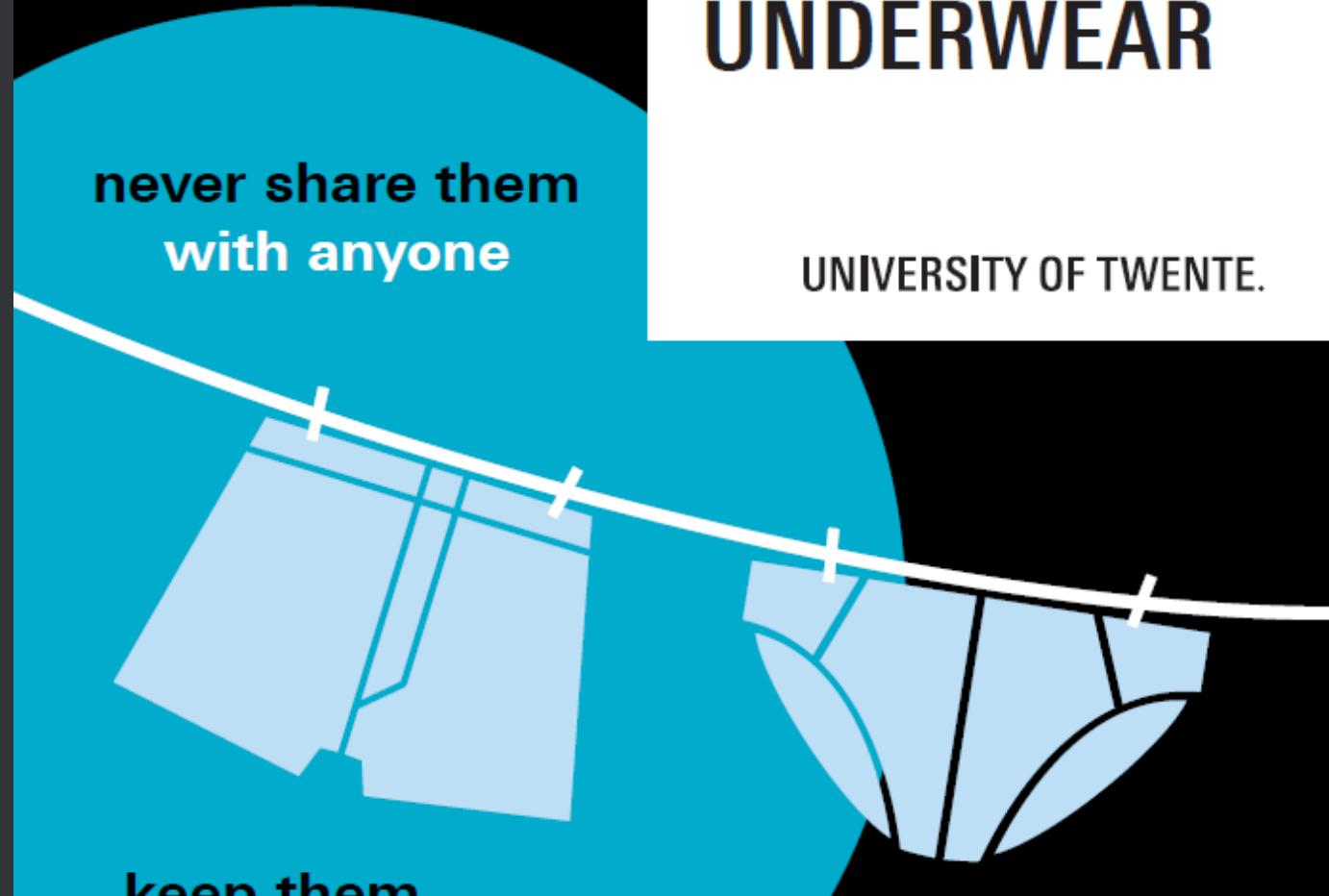


Make passwords hard to guess

The most common passwords are "password", "password1" and "123456." Hackers try easily-guessed passwords because they're used by half of all people. A strong password has seven or more characters and a combination of upper and lower case letters, numbers, and symbols (like !@#\$&*). A phrase that incorporates numbers and symbols can also be a strong password - the key is picking a phrase with specific meaning to you so it's easy to remember, like a favorite hobby, for example (like ILove2Fish4Trout!).

TREAT YOUR PASSWORDS LIKE YOUR UNDERWEAR

UNIVERSITY OF TWENTE.



never share them
with anyone

keep them
off your desk

change
them often

utwente.nl/cyber-safety

Password requirement best practices (outdated)

- Ensure passwords are "complex", i.e. composed of numeric, alphabetic (uppercase and lowercase) characters in addition to special symbols and similar characters
- Force users to change passwords regularly
- Require new passwords not previously used by the user
- Example of a good password: P@ssw0rd1

Terrible password requirement practices

- Maximum length of 8-10 characters
- Minimum password age policy (to prevent password requirement dodging)
- Disable cut-and-paste
- Password hints which lack sufficient entropy
- Show an on-screen keyboard and make user click to enter password



Bank login "security images"

usbank

[◀ Return](#)

ID Shield Image / Sound and Phrase

[ID Shield FAQ](#)

Choose an image or sound category to identify your account.

Wild Animals

Enter a phrase for your account.
You will see this each time your ID Shield image or sound is displayed.

[Continue](#) [Cancel](#)

Studying the Effectiveness of Security Images in Internet Banking

Joel Lee

Carnegie Mellon University
Pittsburgh, PA
jlee@cmu.edu

Lujo Bauer

Carnegie Mellon University
Pittsburgh, PA
lbauer@cmu.edu

Abstract—Security images are often used as part of the login process on internet banking websites, under the theory that they can help foil phishing attacks. Previous studies, however, have yielded inconsistent results about users' ability to notice that a security image is missing and their willingness to log in even when the expected security image is absent. This paper describes an online study of 482 users that attempts to clarify to what extent users notice and react to the absence of security images. We also study the contribution of various factors to the effectiveness of security images, including variations in appearance and interactivity requirements, as well as different levels of user motivation. The majority of our participants (73%) entered their password when we removed the security image and caption. We found that features that make images more noticeable do not necessarily make them more effective at preventing phishing attacks, though some appearance characteristics succeed at discouraging users from logging in when the image is absent. Interestingly, we find that habituation, the level of financial compensation, and the degree of security priming, at least as explored in our study, do not influence the effectiveness of security images.

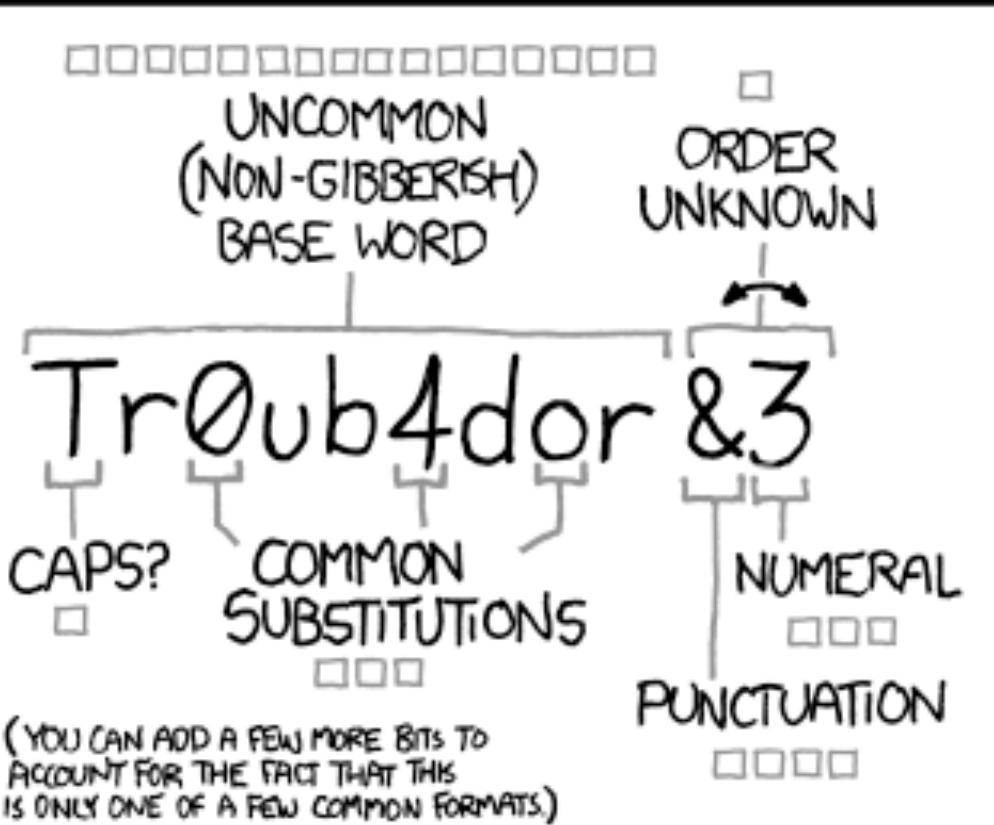
phishing site, users' ability to notice that an expected image is missing and then refuse to log in is not well understood.

Previous studies of the effectiveness of security images have reached divergent conclusions: in one, 92% of participants proceeded to log into their bank account even when the security image was absent [5]; in another, 60% of users of an online assignment-submission system noticed missing security images and refused to log in [6]. These previous studies used different methodologies, making it difficult to reconcile their results or isolate specific reasons for their divergence. Additionally, both studies were carried out in settings sufficiently different from real-world online banking scenarios that it is difficult to generalize from their results.

With the study described in this paper, we seek to shed further light on the ability of users to notice and appropriately react to the absence of security images, and the factors that influence the effectiveness of security images. We study 482 participants in an online setting, as they interact with a simulated banking web site over a period of several days. Our

What we've learned about password requirements

- Complex isn't necessarily strong
 - Numeric, alphabetic, special symbols doesn't lead to stronger passwords
 - "Choosing multiple words from a suitably large dictionary of words may result in stronger passwords even if all of the words appear in dictionaries, are spelled with lowercase letters, and no punctuation is used"
 - Instead, check passwords against known leaked breach data
- Changing passwords regularly leads to weak passwords
- Length is the most important factor



~28 BITS OF ENTROPY

□□□□□□□

□□□□□□

□□□□

□□□□

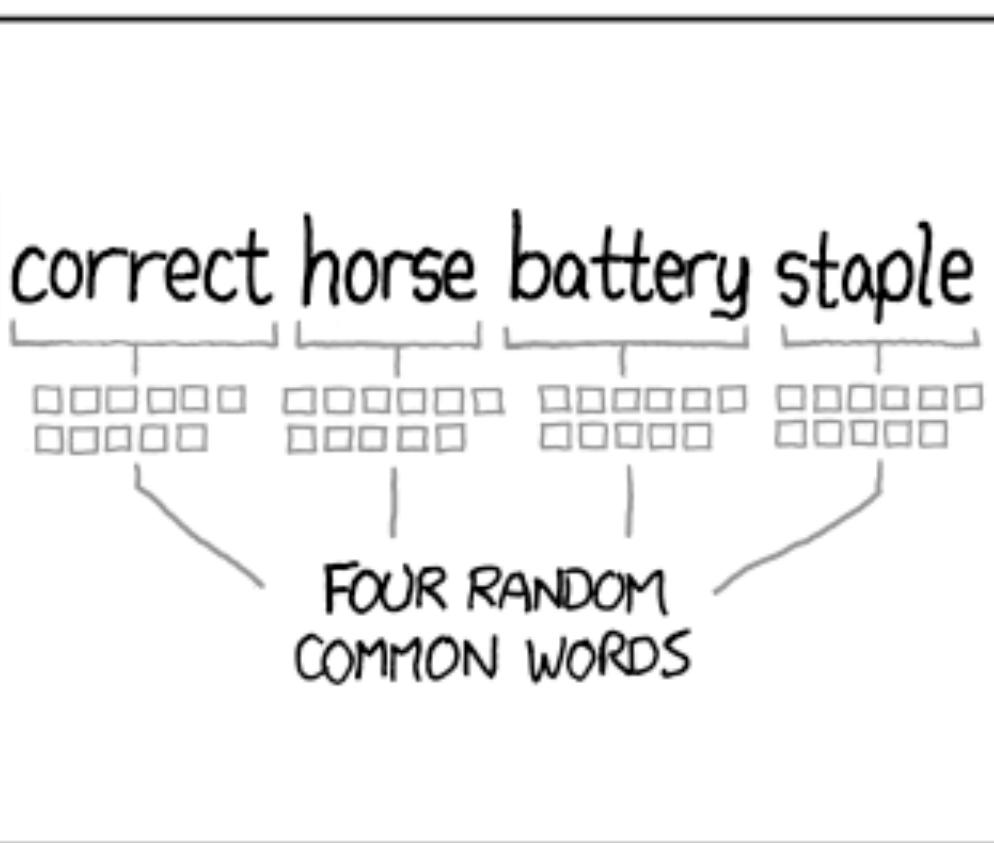
$2^{28} = 3 \text{ DAYS AT } 1000 \text{ GUESSES/SEC}$

(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.

User passwords are too short

Password length	Time to crack
9 characters	2 minutes
10 characters	2 hours
11 characters	6 days
12 characters	1 year
13 characters	64 years

Assuming a password comprised of uppercase, lowercase, and number characters

Demo: Check password strength

<https://www.grc.com/haystack.htm>

Password requirement best practices (updated)

- Minimum password length should be at least 8 characters
- Maximum password length should be at least 64 characters
 - Do not allow unlimited length, to prevent long password denial-of-service
 - Common gotcha: **bcrypt** has a max length of 72 ASCII characters
- Check passwords against known breach data
- Rate-limit authentication attempts
- Encourage/require use of a second factor

Common implementation mistakes

- Do not silently truncate long passwords
- Do not restrict characters
 - Unicode and whitespace characters should be allowed
- Do not include passwords in plaintext log files
- Obvious: Use TLS for all traffic

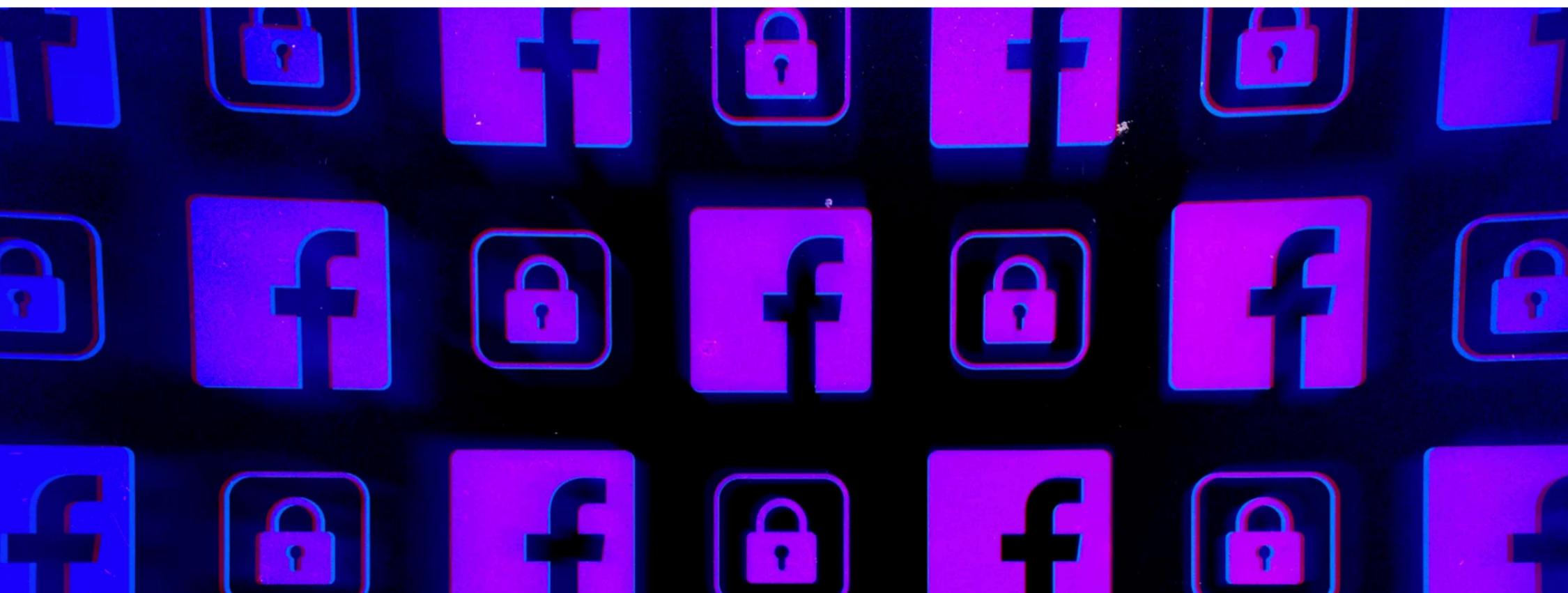
TECH \ FACEBOOK \ CYBERSECURITY

Facebook stored hundreds of millions of passwords in plain text

60

By Jacob Kastrenakes | @jake_k | Mar 21, 2019, 12:03pm EDT

SHARE



Network-based guessing attacks

Whereas systems choose keys at random, **users attempting to choose memorable passwords will often select from a very small subset of the possible passwords of a given length**, and many will choose very similar values. As such, whereas cryptographic keys are typically long enough to make network-based guessing attacks untenable, user-chosen passwords may be vulnerable, especially if no defenses are in place.

Network-based guessing attacks

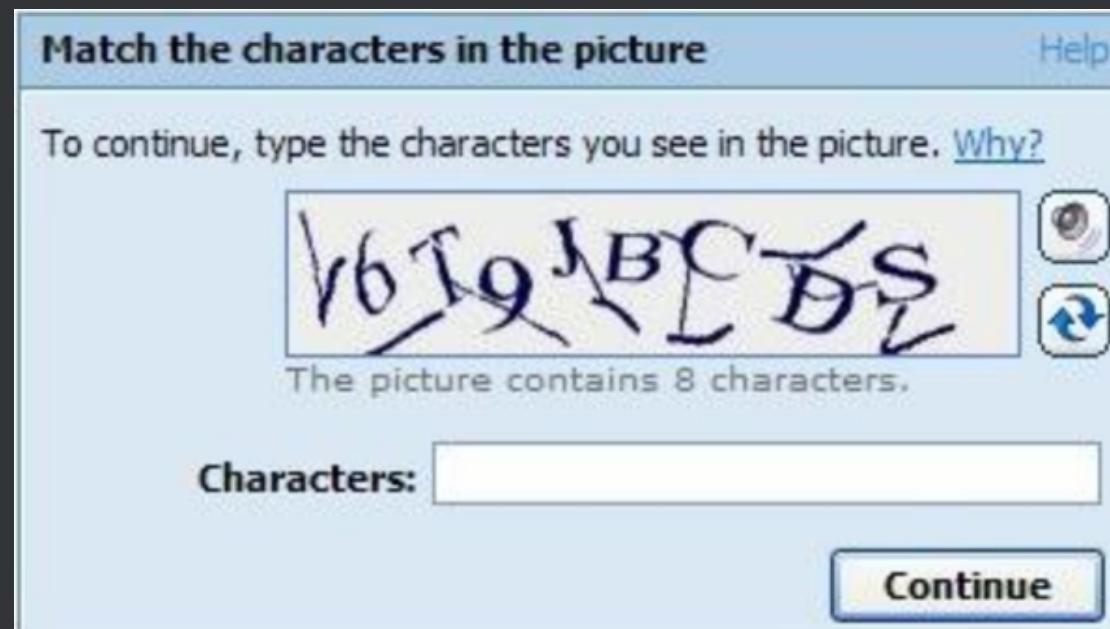
- Three primary types of attack
 - **Brute force:** Testing multiple passwords from dictionary or other source against a single account
 - **Credential stuffing:** Testing username/password pairs obtained from the breach of another site
 - **Password spraying:** Testing a single weak password against a large number of different accounts

Network-based guessing defenses

- Limit the rate at which an attacker can make authentication attempts, or delay incorrect attempts
- Keep track of IP addresses and limit the number of unsuccessful attempts
- Temporarily ban the user after too many unsuccessful attempts

CAPTCHA

- "Completely Automated Public Turing test to tell Computers and Humans Apart"
- Reverse Turing test



Recognizing Objects in Adversarial Clutter: Breaking a Visual CAPTCHA

Greg Mori
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
mori@cs.berkeley.edu

Jitendra Malik
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
malik@cs.berkeley.edu

Abstract

In this paper we explore object recognition in clutter. We test our object recognition techniques on Gimpy and EZ-Gimpy, examples of visual CAPTCHAs. A CAPTCHA (“Completely Automated Public Turing test to Tell Computers and Humans Apart”) is a program that can generate and grade tests that most humans can pass, yet current computer programs can’t pass. EZ-Gimpy (see Fig. 1, 5), currently used by Yahoo, and Gimpy (Fig. 2, 9) are CAPTCHAs based on word recognition in the presence of clutter. These CAPTCHAs provide excellent test sets since the clutter they contain is adversarial; it is designed to confuse computer programs. We have developed efficient methods based on shape context matching that can identify the word in an EZ-Gimpy image with a success rate of 92%, and the requisite 3 words in a Gimpy image 33% of the time. The problem of identifying words in such severe clutter provides valuable insight into the more general problem of object recognition in scenes. The methods that we present are instances of a framework designed to tackle this general problem.

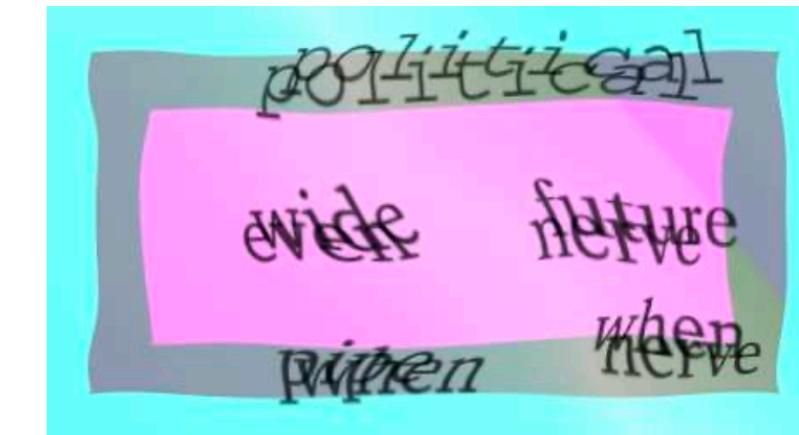


Figure 2: A Gimpy CAPTCHA. The task is to list 3 different words from the image.

2. Current computer programs can't pass

CAPTCHA stands for “Completely Automated Public Turing test to Tell Computers and Humans Apart”.

The concept behind such a program arose from real world problems faced by internet companies such as Yahoo and AltaVista. Yahoo offers free email accounts. The intended users are humans, but Yahoo discovered that various

Problems with CAPTCHAs

- It takes the average person approximately 10 seconds to solve a typical CAPTCHA
- Difficult for users with visual impairment to use
 - Security of the CAPTCHA is only as strong as the weakest form of CAPTCHA offered
- Attackers can proxy CAPTCHA requests to another user in real-time
- Dark market services offer cheap CAPTCHA solving services powered by humans



CAPTCHA solving service

✓ **Cheapest price on the market**

Starting from 0.5USD per 1000 images,
depending on your daily upload volume

✓ **Pay as you go**

Pay-per-captcha payment basis. Minimum refill is
1 USD, no recurring charges

✓ **99.99% uptime since 2007**

Vast amount of workers and premium
infrastructure allows us to provide highly reliable
24/7/365 service

✓ **Solving Google Recaptcha since 2016**

You may fully rely on our stable solution and
forget about browser emulation

[Create Account](#)[Customers Area](#)

The End is Nigh: Generic Solving of Text-based CAPTCHAs

Elie Bursztein
Google
elieb@google.com

Jonathan Aigrain
Stanford University
jonathan.aigrain@gmail.com

Angelika Moscicki
Google
moscicki@google.com

John C. Mitchell
Stanford University
jcm@cs.stanford.edu

Abstract

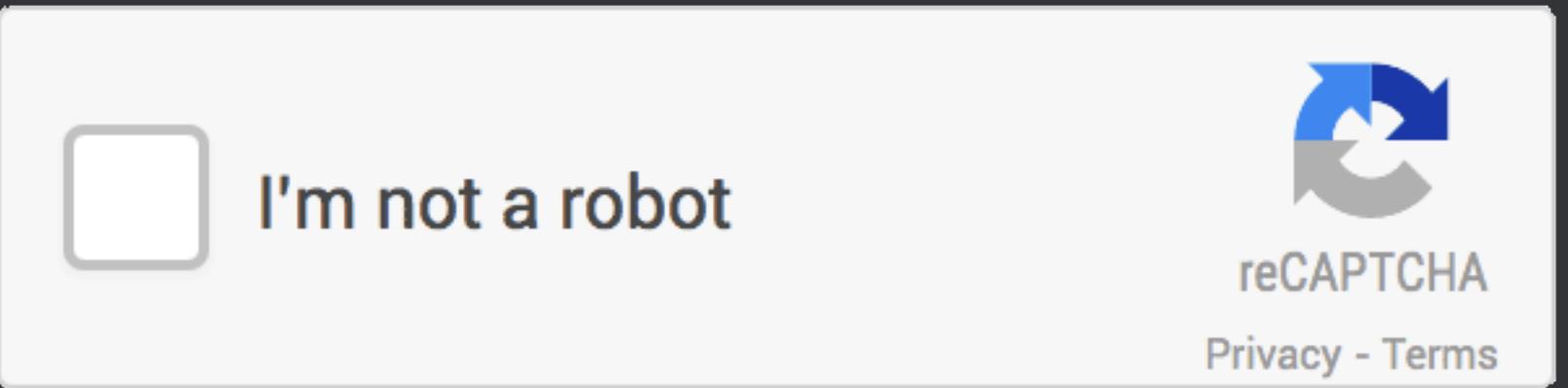
Over the last decade, it has become well-established that a captcha's ability to withstand automated solving lies in the difficulty of segmenting the image into individual characters. The standard approach to solving captchas automatically has been a sequential process wherein a segmentation algorithm splits the image into segments that contain individual characters, followed by a character recognition step that uses machine learning. While this approach has been effective against particular captcha schemes, its generality is limited by the segmentation step, which is hand-crafted to defeat the distortion at hand. No general algorithm is known for the character collapsing anti-segmentation technique used by most prominent real world captcha schemes.

This paper introduces a novel approach to solving captchas in a single step that uses machine learning to attack the segmentation and the recognition problems simultaneously. Performing both operations jointly allows our algorithm to exploit information and context that is not available when they are done sequentially. At the same time, it removes the need for any hand-crafted component, making our approach generalize to new captcha schemes where the previous approach can not. We were able to solve all the real world captcha schemes we evaluated accurately enough to consider the scheme insecure in practice, including Yahoo (5.33%) and ReCaptcha (33.34%), without any adjustments to the algorithm or its parameters. Our success against the Baidu (38.68%) and CNN (51.09%) schemes that use occluding lines as well as character collapsing leads us to believe that our approach is able to defeat occluding lines in an equally general manner. The effectiveness and universality of our results

1 Introduction

Many websites use CAPTCHAs [39], or *Completely Automated Public Turing tests to tell Computers and Humans Apart*, to block automated interaction with their sites. For example, GMail uses captchas ¹ to block access by automated spammers, eBay uses captchas to improve its marketplace by blocking bots from flooding the site with scams, and Facebook uses captchas to limit creation of fraudulent profiles used to spam honest users or cheat at games. The most widely used captcha schemes use combinations of distorted characters and obfuscation techniques that humans can recognize but that may be difficult for automated scripts. Captchas are sometimes called *reverse Turing tests*, because they are intended to allow a computer to determine whether a remote client is human or machine.

Due to the proficiency of machine learning algorithms at recognizing single letters, it has become well-established that a captcha's ability to withstand automated solving lies in the difficulty of segmenting the image into individual characters [12, 10]. The standard approach to solving captchas automatically has been a sequential process wherein a segmentation algorithm splits the image into segments that contain individual characters, followed by a character recognition step that uses machine learning [13]. This is known as the *segment then recognize* approach. While this approach has been effective against particular captcha schemes [15, 4], its generality is limited by the segmentation step, which is hand-crafted to defeat the distortion at hand. No general algorithm is known for the character collapsing anti-segmentation technique used by most prominent real



Reauthenticate for sensitive features

- Defense-in-depth against XSS, CSRF, session hijacking, physical access
- Before: change password, change email, add new shipping address



Confirm password to continue

Password

[Forgot password?](#)

[Confirm password](#)

Tip: You are entering [sudo mode](#). We won't ask for
your password again for a few hours.

Response discrepancy information exposure

- **Information exposure:** Information is leaked to an attacker that should not be leaked
- **Response discrepancy:** "The software provides different responses to incoming requests in a way that allows an actor to determine system state information that is outside of that actor's control sphere."

Response discrepancy: error messages

- Respond with a generic error message regardless of whether:
 - The username or password was incorrect
 - The account does not exist
 - The account is locked or disabled
- Don't forget password reset and account creation!

Response discrepancy: Login

- **Incorrect response examples:**
 - "Login for User foo: invalid password"
 - "Login failed, invalid user ID"
 - "Login failed; account disabled"
 - "Login failed; this user is not active"
- **Correct response example:**
 - "Login failed; Invalid user ID or password"

Response discrepancy: Password recovery

- **Incorrect response examples:**
 - "We just sent you a password-reset link"
 - "This email address doesn't exist in our database"
- **Correct response example:**
 - "If that email address is in our database, we will send you an email to reset your password"

Response discrepancy: Account creation

- **Incorrect response examples:**
 - "This user ID is already in use"
 - "Welcome! You have signed up successfully"
- **Correct response example:**
 - "A link to activate your account has been emailed to <input email address>"

Response discrepancy: HTTP status codes

- Any difference will leak info to the attacker, even HTTP status codes
- **Incorrect response examples:**
 - *Sometimes* HTTP 200: "Login failed; Invalid user ID or password"
 - *Sometimes* HTTP 403: "Login failed; Invalid user ID or password"
- **Correct response example:**
 - *Always* HTTP 403: "Login failed; Invalid user ID or password"

Response discrepancy: Timing

Bad:

```
const userExists = await lookupUserExists(username)
if (userExists) {
  const passwordHash = hash(password)
  const isValid = await lookupCredentials(username, passwordHash)
  if (!isValid) {
    throw Error('Invalid username or password')
  }
} else {
  throw Error('Invalid username or password')
}
```

Response discrepancy: Timing

Good:

```
const passwordHash = hash(password)
const isValid = await lookupCredentials(username, passwordHash)
if (!isValid) {
  throw Error('Invalid username or password')
}
```

- Beware of using **early returns** in authentication code

Response discrepancy: Mitigation tradeoffs

- Mitigations make user experience worse
 - Generic error messages are less useful to the user
 - Can frustrate legitimate users
- Rate-limiting authentication attempts will prevent user enumeration at scale, while allowing friendly error messages to remain

Data breaches

TECH

Credit reporting firm Equifax says data breach could potentially affect 143 million US consumers

PUBLISHED THU, SEP 7 2017 4:34 PM EDT | UPDATED FRI, SEP 8 2017 3:25 PM EDT

**Todd Haselton**
@ROBOTODD

SHARE



KEY POINTS

- Equifax said data on 143 million U.S. customers was obtained in a breach.
- The breach was discovered July 29.
- Personal data including birth dates, credit card numbers and more were obtained in the breach.
- Three Equifax executives sold shares in the company days after the breach was discovered.

NEWS

[Home](#) | [Video](#) | [World](#) | [US & Canada](#) | [UK](#) | [Business](#) | [Tech](#) | [Science](#) | [Stories](#) | [More](#) ▾[Business](#) | [Market Data](#) | [Global Trade](#) | [Companies](#) | [Entrepreneurship](#) | [More](#) ▾

'One billion' affected by Yahoo hack

🕒 15 December 2016

 [f](#) [m](#) [t](#) [e](#) [Share](#)

REUTERS

Yahoo has said more than one billion user accounts may have been affected in a hacking attack dating back to 2013.

The internet giant said it appeared separate from a 2014 breach disclosed in September, when Yahoo revealed 500 million accounts had been accessed.

Yahoo said names, phone numbers, passwords and email addresses were

Biggest data breaches

Yahoo - 3 billion

Twitter - 330 million

Canva - 137 million

Rambler - 91 million

Aadhaar - 1.1 billion

NetEase - 234 million

Apollo - 126 million

Facebook - 87 million

Verifications.io - 763 million

LinkedIn - 165 million

Badoo - 112 million

Dailymotion - 85 million

Yahoo - 500 million

Dubsmash - 162 million

Evite - 101 million

Dropbox - 69 million

Marriott/Starwood - 500 million

Adobe - 152 million

Quora - 100 million

tumblr - 66 million

Adult Friend Finder - 412.2 million

MyFitnessPal - 150 million

VK - 93 million

MySpace - 360 million

Equifax - 148 million

MyHeritage - 92 million

Exactis - 340 million

eBay - 145 million

Youku - 92 million

Were you in a breach?

- The answer is almost certainly "Yes"
- **HavelBeenPwned.com** service
 - Check for your email in breaches
 - Check for your password in breaches
 - Websites or password managers can check passwords against the list

Demo: Were you in a breach?

<https://haveibeenpwned.com/>

How are passwords stored?

- **Important:** Never, ever, ever store passwords in plaintext
- In a data breach, the attacker will learn all users' passwords and be able to attack their accounts on other sites, assuming the user has re-used their password across sites (very likely)

TWITTER LEAKED DATABASE: 32 MILLION ACCOUNTS

Price is 0.5 BTC

Send bitcoin to this address REDACTED

When its done, send me your transaction details to REDACTED and I give you download link in 2 hours.

This leak includes the emails and passwords for every single Twitter account registered before 2015.

Just open up the database in your favorite text editor and Ctrl + F for the email or username you want to hack.

Yes, this means that for 0.5 BTC you can hack ANY twitter user, and if they use the same password on other sites you can hack into there too. Their iCloud with all their personal photos, their email accounts, facebook and instagram are all vulnerable to being hacked once you have this database.

Enjoy and please help keep this leak private by not sharing it after you've purchased.

Proof of content, first 100 lines of accounts:

(Format is email:password)

```
root@kali-linux$ head -n 100 twitter-1.txt
REDACTED@yahoo.com:REDACTED
REDACTED@hotmail.com:REDACTED
REDACTED@gmail.com:REDACTED
REDACTED@yahoo.com:REDACTED
REDACTED@yahoo.com:REDACTED
REDACTED@hotmail.com:REDACTED
REDACTED@tele2.nl:REDACTED
REDACTED@gmail.com:REDACTED
```

User table (plaintext)

Username	Password
alice	password
bob	hunter2
charlie	correct-battery-horse-staple
dakotah	hunter2

Never store plaintext passwords

- **Important:** Hash the plaintext password, then store the hash in the database
- **Cryptographic hash function:** Algorithm that maps data of arbitrary size (the "message") to a bit string of a fixed size (the "hash value")
 - **One-way function:** infeasible to invert
 - **Deterministic:** same message always results in the same hash value
 - **Quick to compute:** we often call hash functions thousands of times
 - **No collisions:** infeasible to find different messages with same hash value
 - **Avalanche effect:** small change to message changes hash value extensively

Example: Hashing passwords

```
const crypto = require('crypto')
const sha256 = s => crypto.createHash('sha256').update(s).digest('hex')

const user = await createUser(username, sha256(password))

// later...

const isValid = sha256(otherPassword) === passwordHash
```

User table (hashed)

Username	Password (Hashed)
alice	XohImNooBHFR00VvjcYpJ3NgPQ1qq73WKhHvch0VQt g=
bob	9S+9MrKzuG/4jvbEkGKChfSCrxXdyyIUH5S89Saj9sc=
charlie	0mk89QsPD4FIJQv8IcHnoSe6qjOzKvcNuTevydeUxWA=
dakotah	9S+9MrKzuG/4jvbEkGKChfSCrxXdyyIUH5S89Saj9sc=

Problems with just hashing

1. Users who have identical passwords are easy to spot
2. Pre-computed lookup attacks are easy
 - SHA256 is quite fast to compute
 - Rainbow tables are easy to generate
 - **Rainbow table:** a precomputed table for reversing cryptographic hash functions

Password salts

- **Goal:**
 - Prevent two users who use identical passwords from being revealed
 - Add entropy to weak passwords to make pre-computed lookup attacks intractable
- **Solution:** A **salt** is fixed-length cryptographically-strong random value
 - No need to keep the salt secret; can be stored alongside the password (salt is usually 16, 32, or 64 bytes)
 - Concatenate the salt and the password before hashing it

Example: Hashing and salting passwords

```
const crypto = require('crypto')
const sha256 = s => crypto.createHash('sha256').update(s).digest('base64')

const salt = crypto.randomBytes(16).toString('base64')
const passwordHash = sha256(salt + password)
const user = await createUser(username, salt, passwordHash)

// later...

const salt = await getSalt(username)
const otherPasswordHash = sha256(salt + otherPassword)
const isValid = otherPasswordHash === passwordHash
```

User table (hashed and salted)

Username	Salt	Password (Hashed & Salted)
alice	ciMTj87Q5Ti/ PDfSUM4jcAT6cFJWVwJFjEbMc2sqAn0=	AQAiFDIbEUk5Wdoe6tTL+bnCBOlsectOW2Sf ftG0je8=
bob	NB9zdy/ OIvnGHkPK7fK01saCcIpXrWV5rdtW8i5k/XY=	uxIXXvfrQ8/gTwrbTtgnsqsZCAw/ y2408nU3qlho5GE=
charlie	hetbWcTifseB9K3lQQPr6c/eMJyj3kVTqq/ l+FqYf78=	FykuFcJVOAjBLyxMuQWr vuSTjRXyXStitVteW UJmPlM=
dakotah	lZu5hPamBS/ QY4ILZzTcyVY8TK17Dt9hmXW7bC4XbCc=	ydVe+vA56bKbAOoXzRfYtkABUXaxgkF4ngB0 xNJRvA4=

Just use bcrypt

- Password hashing function designed by Niels Provos and David Mazières
- Expensive key setup algorithm
 - You don't want speed in a password hash function
- Automatically handles all password salting complexity and includes it in the hash output

Example: Just use bcrypt

```
const bcrypt = require('bcrypt')
const HASH_ROUNDS = 10

const passwordHash = bcrypt.hashSync(password, HASH_ROUNDS)
const user = await createUser(username, passwordHash)

// later...

const isValid = bcrypt.compareSync(otherPassword, passwordHash)
```

User table (bcrypt)

Username	Password
alice	\$2b\$10\$aQNe4MK0HDhrkus8GZGQL.Nj11nsx12VTMTDBkykiL/jRbb.fJuGC
bob	\$2b\$10\$TSbaMNCCq6.xNkDVszwh09Fpb.eeW6aUSIFzGkPoQrs5RahskOUO
charlie	\$2b\$10\$.5KcQQNEfnkPBYxeiqS2ZeePXT5J30HG7zngfesyGuc0js37X41e
dakotah	\$2b\$10\$l8n7ZLsq13ygE0m3cQ8oEuBjPnGcGBUA4zvJhnsKgyDEZdEd2EFXa

How attackers use a breach database

- Machine capable of cracking 100B+ passwords per second against SHA256 can be built for \$20,000 (as of July 2019)
- Try every password which has been disclosed in a breach (>500M passwords). Think of this as “every password anyone has ever thought of, ever.” **Statistically, this will break >70% of user passwords**
- The complete list just takes 5ms to try, so an attacker can run the complete list against 200 accounts every second
- Build a list of popular phrases, song lyrics, news headlines to pick up another 5-7% of user passwords

Multi-factor authentication

- Microsoft: "Based on our studies, your account is more than 99.9% less likely to be compromised if you use MFA"
- Common additional factors:
 - Something you have (e.g. a phone, badge, or cryptographic key)
 - Something you are (e.g. a fingerprint or other biometric data)

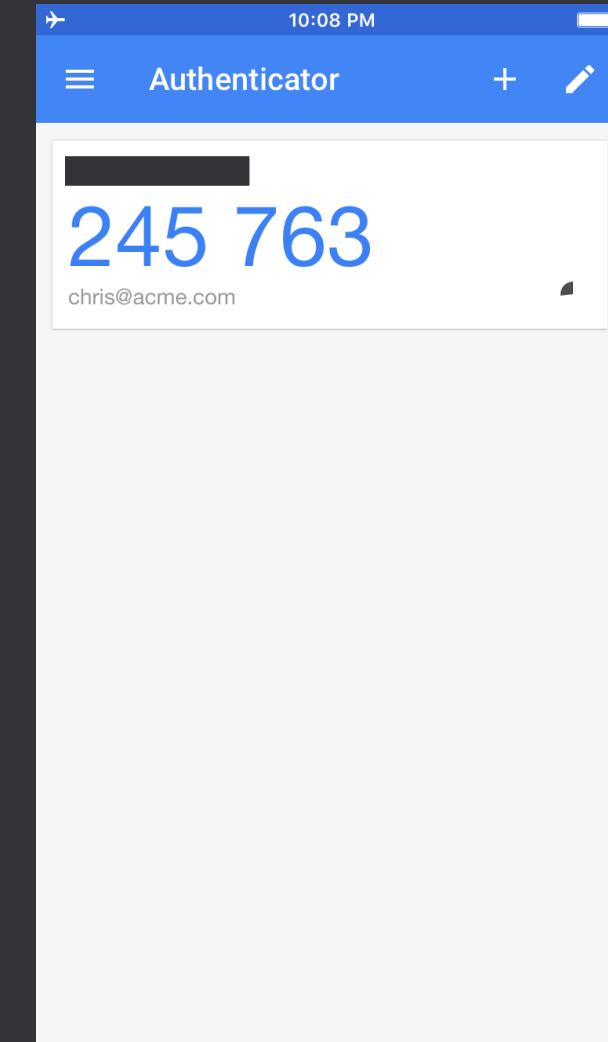
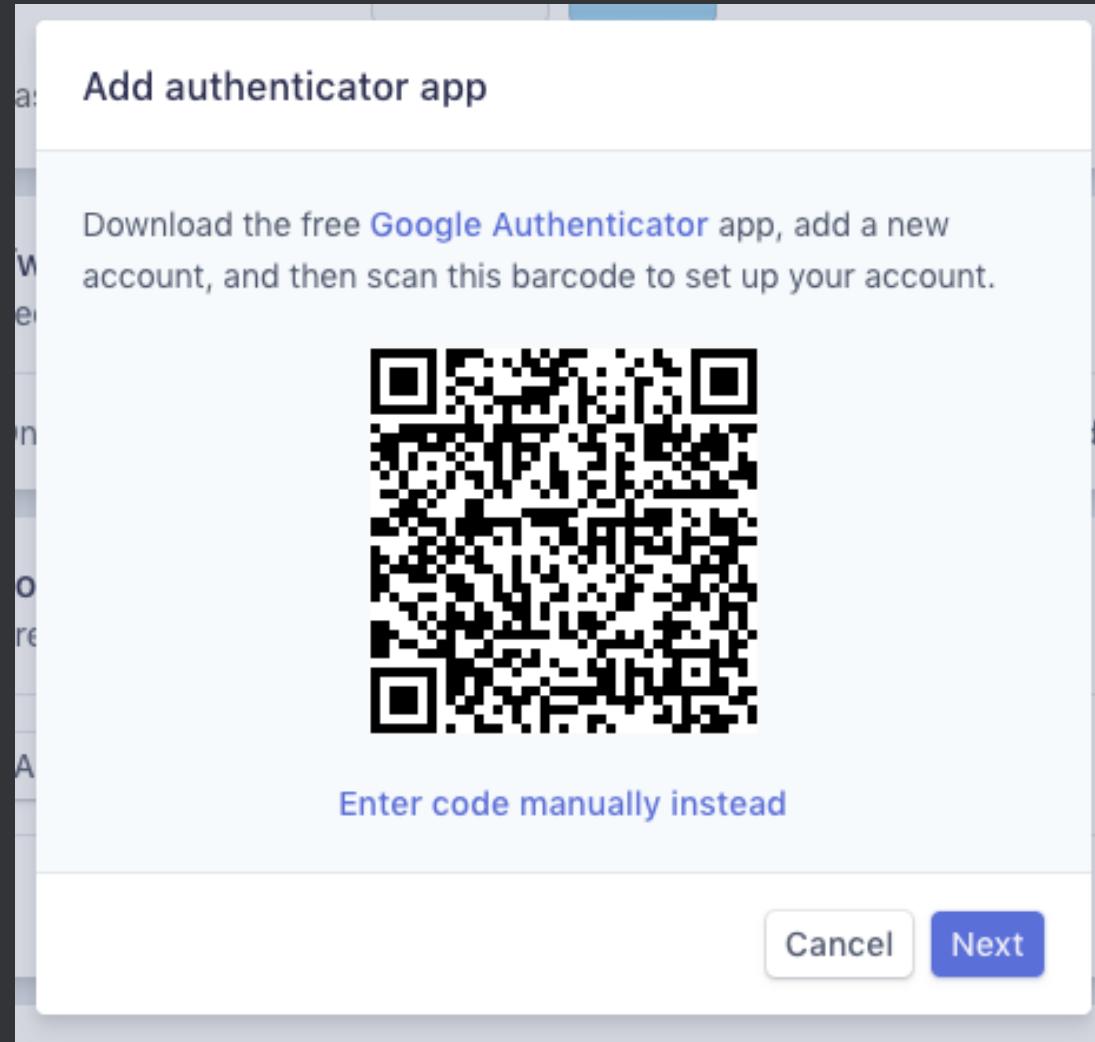
Multi-factor authentication

- Choosing a strong password **can** prevent these attacks:
 - **Password spray:** Guessing, hammering, low-and-slow
 - **Brute force:** Database extraction, cracking
- Choosing a strong password **cannot** prevent these attacks:
 - **Credential stuffing:** Breach replay, list cleaning
 - **Phishing:** Man-in-the-middle, credential interception
 - **Keystroke logging:** Malware, sniffing
 - **Local discovery:** Dumpster diving, physical recon, network scanning
 - **Extortion:** Blackmail, insider threat

Selectively requiring MFA

- To preserve **user experience**, consider only requiring MFA for:
 - A new browser/device or IP address
 - An unusual country or location
 - An IP address that appears on known blocklists
 - An IP address that has tried to login to multiple accounts
 - A login attempt that appears to be scripted rather than manual

Time-based One-Time Passwords (TOTP)



Time-based One-Time Passwords (TOTP)

1. Server creates a secret key for specific user
2. Server shares secret key with the user's phone app
3. Phone app initializes a counter
4. Phone app generates a one time password using secret key and counter
5. Phone app changes the counter after a certain interval and regenerates the one time password

Time-based One-Time Passwords (TOTP)

Server generates unique secret key for user:

```
const secretKey = crypto.randomBytes(160)
```

```
await addSecretKeyForUser(username, secretKey)
```

```
// Now give secret key to user via QR code...
```

Time-based One-Time Passwords (TOTP)

Generate the current one-time password:

```
const counter = Math.floor(Date.now() / (30 * 1000))
const hmacHash = crypto.createHmac('sha1', secretKey).update(counter).digest()

const offset = hmacHash[19] & 0xf
const truncatedHash = (hmacHash[offset++] & 0x7f) << 24 |
  (hmacHash[offset++] & 0xff) << 16 |
  (hmacHash[offset++] & 0xff) << 8 |
  (hmacHash[offset++] & 0xff)

const finalOTP = truncatedHash % (10 ^ 6)
```

Final thoughts

- Always hash and salt your passwords
- "Just use bcrypt"
- Consider how to protect users even when attackers know their password

END

Credits:

<https://www.upguard.com/blog/biggest-data-breaches>

<https://www.utwente.nl/en/news/2018/8/308711/treat-your-password-like-its-underwear>

<https://blog.codinghorror.com/your-password-is-too-damn-short/>

<https://cheatsheetseries.owasp.org/cheatsheets/AuthenticationCheatSheet.html>

[https://techcommunity.microsoft.com/t5/Azure-Active-Directory-Identity/Your-Password-doesn't-matter/ba-p/731984](https://techcommunity.microsoft.com/t5/Azure-Active-Directory-Identity/Your-Password-doesn-t-matter/ba-p/731984)